A Distributed Evolutionary Method to Design Scheduling Policies for Volunteer Computing

Trilce Estrada
University of Delaware
Dept. of Computer & Information Sciences
Newark, DE, 19716 USA
estrada@udel.edu

Olac Fuentes
University of Texas at El Paso
Dept. of Computer Science
El Paso, TX, 79968 USA
ofuentes@utep.edu

Michela Taufer
University of Delaware
Dept. of Computer & Information Sciences
Newark, DE, 19716 USA
taufer@udel.edu

ABSTRACT
Volunteer Computing (VC) is a paradigm that uses idle cycles from computing resources donated by volunteers and connected through the Internet to compute large-scale, loosely-coupled simulations. A big challenge in VC projects is the scheduling of work-units across heterogeneous, volatile, and error-prone computers. The design of effective scheduling policies for VC projects involves subjective and time-demanding tuning that is driven by the knowledge of the project designer. VC projects are in need of a faster and project-independent method to automate the scheduling design.

To automatically generate a scheduling policy, we must explore the extremely large space of syntactically valid policies. Given the size of this search space, exhaustive search is not feasible. Thus in this paper we propose to solve the problem using an evolutionary method to automatically generate a set of scheduling policies that are project-independent, minimize errors, and maximize throughput in VC projects. Our method includes a genetic algorithm where the representation of individuals, the fitness function, and the genetic operators are specifically tailored to get effective policies in a short time. The effectiveness of our method is evaluated with SimBA, a Simulator of BOINC Applications. Contrary to manually-designed scheduling policies that often perform well only for the specific project they were designed for and require months of tuning, our resulting scheduling policies provide better overall throughput across the different VC projects considered in this work and were generated by our method in a time window of one week.

Categories and Subject Descriptors
I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—Dynamic programming, Heuristic methods, Scheduling; I.6.3 [Computing Methodologies]: Simulation and Modeling—Applications

General Terms
Design, Management, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF’08, May 5–7, 2008, Ischia, Italy.
Copyright 2008 ACM 978-1-60558-077-7/08/05 ...$5.00.

1. INTRODUCTION
Volunteer Computing (VC) is a paradigm that uses heterogeneous computing resources (e.g., desktops, notebooks) connected through the Internet and owned by volunteers to provide computing power needed by computationally expensive, loosely-coupled applications. For such applications, VC systems represent an effective alternative to traditional High Performance Computing (HPC) systems because they can provide higher throughput at a lower cost.

Many VC projects involve simulating phenomena in nature. Examples of VC projects are: Predictor@Home [21], which predicts protein structures; Folding@Home [16], which explores the physical processes of protein folding; and climateprediction.net1, which predicts climate phenomena such as El Niño. The major challenge in VC projects is that the computers, also called workers or hosts, are non-dedicated, volatile, error-prone, and unreliable. Indeed, workers donate idle cycles and therefore are not fully dedicated to the execution of VC applications. Their volatility is due to the fact that they may suddenly leave the project without returning any results (or returning partial results) for the assigned computation. Network and execution errors can take place at any time and cannot be predicted. Last but not least, the results returned may be affected by malicious attacks, hardware malfunctions, or software modifications and therefore can be invalid [22]. When any of these conditions happens, a VC system should be ready to discard and redistribute the affected computation at the cost of decreasing the overall throughput and increasing the replicated computation.

VC projects rely on a scheduler to lessen the impact of these conditions. The scheduler makes decisions about the amount of computation or work-units (WUs) that should be assigned to a worker in terms of number of worker-unit replicas or instances (WUIs). Because of the heterogeneity of the worker community, in terms of performance and availability, the scheduler cannot apply the same distribution criteria to every machine. Moreover, both performance and availability of a worker can suddenly change between two requests for computation. Ideally, schedulers should be able to intelligently adapt to the characteristics of the worker community as the simulation evolves.

The creation of an effective scheduling policy is usually a time-demanding and subjective tuning process based on the knowledge that the project developer has about the specific application and the workers participating in the VC project. Project administrators of-

1http://climateprediction.net
ten spend months observing the behavior of a VC project before changing any parameter setting. Measuring the effectiveness of a scheduling policy and tuning its parameters may also take several months. Finally, extensive tuning does not guarantee that the optimal scheduling policy will be found, since the volunteer community is constantly changing. We argue that VC projects are in need of a fast, automated, project-independent method to scheduling design. To address this need, we propose the use of a distributed, evolutionary method to search over a wide space of possible scheduling policies for a small subset of policies that minimizes both errors and invalid results while maximizing the project throughput. Note that the contribution of this paper and its novelty is in the application of our method to effectively generate scheduling policies for VC projects.

To assess the effectiveness of our method in capturing the most important characteristics of applications and workers across different VC projects, we consider four BOINC (Berkeley Open Infrastructure for Network Computing) projects: Predictor@Home, FightAIDS@Home, Human Proteome Folding, and Genome Comparison. BOINC [1] is a well-known middleware that enables VC. The four projects have widely varying features such as application size as well as number and type of workers. We compare and contrast their default scheduling policies with the policies resulting from our method. To quickly estimate the performance of the different policies in terms of throughput and replication of computation, we use SimBA, a simulator of BOINC Applications [23]. SimBA is used to accurately design and tune scheduling policies in BOINC projects without affecting the volunteers. Our results show that manually-designed scheduling policies are project dependent, i.e., they perform well for one project but not for a set of projects. Our method was able to automatically identify four scheduling policies that provide better throughput across the different projects and have competitive results in terms of replicated computation with respect to the best manually-designed scheduling policies. These results were achieved over one week of simulations on a non-dedicated cluster of 64 nodes.

The rest of this paper is organized as follows: Section 2 gives a short overview of BOINC, SimBA, scheduling policies for VC, and Genetic Algorithms (GAs); Section 3 presents our distributed, evolutionary method for the design of scheduling policies in VC; Section 4 shows the major achievements of our method for a set of heterogeneous BOINC projects; Section 5 provides a short overview of related work; and Section 6 summarizes the paper and describes future work.

2. BACKGROUND

2.1 BOINC

BOINC is a well-known representative of VC systems [1]. It is an open-source middleware that enables the computing power and storage capacity of thousands of PCs (called workers) connected to the Internet for scientific purposes. A BOINC project comprises hundreds of thousands of independent work-units (WUs). For fault-tolerance and trust reasons, a WU is replicated in several work-unit instances (WUIs) that are distributed to different workers. Each time a worker requests for computation, BOINC builds a package of WUIs for that machine. Network and computation errors or invalid results due to malicious attacks, hardware malfunctions, or software modifications, cause the generation and distribution of new WUIs to replace those faulty WUs. The computing resources available in a BOINC project are heterogeneous: the workers have different processors, memory, and network connections. Some workers connect to a project via modem a few times per day, others are permanently connected. Every BOINC project has its own worker community, which is the set of active workers that donate computation for that specific project, with their own performance features. These features can change unpredictably and dynamically.

In this paper we consider four BOINC projects: Predictor@Home, and three projects from the IBM initiative, World Community Grid. Predictor@home (P@H) is a BOINC project for large-scale protein structure prediction [21]. The protein structure prediction algorithm in P@H is a multi-step pipeline that includes: (a) a conformational search using a Monte Carlo simulated-annealing approach using MFold [19]; and (b) protein refinement, scoring, and clustering using the CHARMM Molecular Dynamics simulation package [14]. World Community Grid (WCG) is an initiative supported by IBM that makes grid technology available to the public and not-for-profit organizations. WCG currently supports several VC projects. In our work we consider FightAIDS@Home, Human Proteome Folding, and Genome Comparison. FightAIDS@Home searches for drugs to disable HIV-1 Protease. Proteome Folding computes simulation of folding for unstudied proteins. Because similar genes usually have similar functions, Genome Comparison identifies already studied genes and compares similarities with unstudied genes. Table 1 summarizes the main features of these projects and their heterogeneity in terms of size of the worker community (number of workers) and application size (flop).

Table 1: Main features of the performance traces of the BOINC projects used in this paper

<table>
<thead>
<tr>
<th>Project</th>
<th>Size of traces (days)</th>
<th>Number workers</th>
<th>Average size of WUI (flop)</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>P@H CHARMM</td>
<td>8</td>
<td>5093</td>
<td>8000000000000000</td>
<td>2004</td>
</tr>
<tr>
<td>P@H MFold</td>
<td>15</td>
<td>7810</td>
<td>1000000000000000</td>
<td>2004</td>
</tr>
<tr>
<td>FightAIDS@Home Genome Comp.</td>
<td>25</td>
<td>35583</td>
<td>35474100205931</td>
<td>2007</td>
</tr>
<tr>
<td>Proteome Folding</td>
<td>24</td>
<td>36590</td>
<td>46978950664183</td>
<td>2007</td>
</tr>
<tr>
<td>Genome Comparison</td>
<td>50</td>
<td>30540</td>
<td>98006404441865</td>
<td>2007</td>
</tr>
</tbody>
</table>

2.2 SimBA

SimBA or Simulator of BOINC Applications [23] is a discrete event simulator that accurately models the main functions of BOINC, i.e., generation, distribution, collection, and validation of WUs in a general VC project. The generation and characterization of simulated workers are driven by traces obtained directly from real BOINC projects. Currently SimBA supports the following scheduling policies: First-Come-First-Served, fixed- and variable thresholds based on availability and reliability of workers [8], and the scheduling policy currently used by the World Community Grid projects. In [23] we showed that SimBA’s predictions of Predictor@Home and World Community Grid performance are within approximately 5% of the performance reported by these projects.

2.3 Volunteer Computing Scheduling

Existing policies that schedule WUs in VC projects are based on heuristics and can be classified in two classes: naive and knowledge-based. Naive policies assign computation without taking into account the history of the workers while knowledge-based scheduling policies look at the history of the worker applying for computation and the whole community. Examples of naive scheduling policies are: (1) First-Come-First-Served (FCFS): WUs are sent to any host that applies for computation [1]; (2) Locality scheduling policy: WUs are preferentially sent to hosts that already have

http://worldcommunitygrid.org
the necessary data to accomplish the work [2]; and (3) Random assignment: WUIs are selected randomly.

Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds (FTR) checks the availability and reliability values of the requesting hosts; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds (VTR) is similar to the fixed thresholds, but the scheduler varies the thresholds at runtime; if the number of WUIs waiting for distribution is greater than the predefined threshold, the scheduler assigns the requested work to that host [8]; and (3) Random assignment: WUIs are selected randomly.

Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds (FTR) checks the availability and reliability values of the requesting hosts; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds (VTR) is similar to the fixed thresholds, but the scheduler varies the thresholds at runtime; if the number of WUIs waiting for distribution is greater than the predefined threshold, the scheduler assigns the requested work to that host [8]; and (3) Random assignment: WUIs are selected randomly.

Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds (FTR) checks the availability and reliability values of the requesting hosts; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds (VTR) is similar to the fixed thresholds, but the scheduler varies the thresholds at runtime; if the number of WUIs waiting for distribution is greater than the predefined threshold, the scheduler assigns the requested work to that host [8]; and (3) Random assignment: WUIs are selected randomly.

Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds (FTR) checks the availability and reliability values of the requesting hosts; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds (VTR) is similar to the fixed thresholds, but the scheduler varies the thresholds at runtime; if the number of WUIs waiting for distribution is greater than the predefined threshold, the scheduler assigns the requested work to that host [8]; and (3) Random assignment: WUIs are selected randomly.

Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds (FTR) checks the availability and reliability values of the requesting hosts; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds (VTR) is similar to the fixed thresholds, but the scheduler varies the thresholds at runtime; if the number of WUIs waiting for distribution is greater than the predefined threshold, the scheduler assigns the requested work to that host [8]; and (3) Random assignment: WUIs are selected randomly.

Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds (FTR) checks the availability and reliability values of the requesting hosts; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds (VTR) is similar to the fixed thresholds, but the scheduler varies the thresholds at runtime; if the number of WUIs waiting for distribution is greater than the predefined threshold, the scheduler assigns the requested work to that host [8]; and (3) Random assignment: WUIs are selected randomly.

Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds (FTR) checks the availability and reliability values of the requesting hosts; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds (VTR) is similar to the fixed thresholds, but the scheduler varies the thresholds at runtime; if the number of WUIs waiting for distribution is greater than the predefined threshold, the scheduler assigns the requested work to that host [8]; and (3) Random assignment: WUIs are selected randomly.
Table 2: Factors used to characterize worker community and applications

<table>
<thead>
<tr>
<th>Factors</th>
<th>Dependency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>os</td>
<td>worker</td>
<td>Operating system: (1) Windows, (2) Linux, (3) Darwin</td>
</tr>
<tr>
<td>processor</td>
<td>worker</td>
<td>Vendor: (1) AMD, (2) Intel, (3) PowerPC Mac, (4) Intel Mac</td>
</tr>
<tr>
<td>availability</td>
<td>worker/application</td>
<td>WUI completed without error or timeout over WUI distributed since last worker connection</td>
</tr>
<tr>
<td>reliability</td>
<td>worker/application</td>
<td>WUI valid over WUI collected since the last worker connection</td>
</tr>
<tr>
<td>availabilityC</td>
<td>worker/application</td>
<td>WUI completed without error or timeout over WUI distributed since worker joined the project</td>
</tr>
<tr>
<td>reliabilityC</td>
<td>worker/application</td>
<td>WUI valid over WUI since worker joined the project</td>
</tr>
<tr>
<td>tps</td>
<td>worker</td>
<td>Integer operations per second</td>
</tr>
<tr>
<td>flops</td>
<td>worker</td>
<td>Floating point operations per second</td>
</tr>
<tr>
<td>lifespan</td>
<td>worker</td>
<td>Number of hours since the worker joined the project</td>
</tr>
<tr>
<td>claimed_credit</td>
<td>worker</td>
<td>Amount of credits claimed in the last connection</td>
</tr>
<tr>
<td>granted_credit</td>
<td>worker</td>
<td>Total amount of credits granted for valid results</td>
</tr>
<tr>
<td>ravg_credit</td>
<td>worker</td>
<td>Recent average of granted credits for valid results</td>
</tr>
<tr>
<td>last_rac_update</td>
<td>worker</td>
<td>Last time the worker received credits</td>
</tr>
<tr>
<td>avg_turnaround</td>
<td>worker</td>
<td>Average time to return a WUI for a given worker</td>
</tr>
<tr>
<td>acquired_wui</td>
<td>worker/application</td>
<td>Number of WUIs given to the worker in the last connection</td>
</tr>
<tr>
<td>avg_dedicated_time</td>
<td>worker/application</td>
<td>Average time dedicated to VC project</td>
</tr>
<tr>
<td>max_wui_day</td>
<td>worker/application</td>
<td>Maximum number of WUIs the worker can get in one day</td>
</tr>
<tr>
<td>wui_inprogress</td>
<td>worker</td>
<td>Number of WUIs still in progress</td>
</tr>
<tr>
<td>app_size_wui</td>
<td>application</td>
<td>Average size of WUs in flops</td>
</tr>
<tr>
<td>starving_workers</td>
<td>worker/application</td>
<td>Number of workers that did not receive work in the last unit of time</td>
</tr>
</tbody>
</table>

if ((host.availability>= 17.92)):
    n_jobs = host.req_comp
    self.cont1 = self.cont1 + 1
elif (((host.flops-(0.264*host.max_wui_day))<0.145) and (host.os<2)):
    n_jobs=int(host.req_comp*0.75)
    self.cont2 = self.cont2 + 1
elif not(67.340<=((host.wui_valid/(host.acquired_wui+1))):
    n_jobs=int(host.req_comp*0.5)
    self.cont3 = self.cont3 + 1
elif (((host.wui_valid/(host.acquired_wui+1))): >host.rac<0.111):
    n_jobs=int(host.req_comp*0.25)
    self.cont4 = self.cont4 + 1
else:
    n_jobs=1
    self.cont5 = self.cont5 + 1
assign_work(n_jobs)

Figure 3: Fragment of Python code generated by the individual in Figure 2

If a worker does not meet any of the m rules, it receives at least one WUI. This mechanism prevents starving workers, i.e., workers that do not receive any computation and therefore continue reapplying for WUI. Note that the scheduler does not necessarily distribute computation to a worker using the same rule during the entire worker’s lifespan.

3.2 Parallel Genetic Algorithm

The proposed method is built upon the concept of Genetic Algorithms (GA) in which we apply some variations to the traditional definition of GA. First of all, our method uses individuals with variable size and individuals are expressions rather than numerical arrays, i.e., floating point or integer. Moreover, the operations have been adapted to deal with expressions rather than numbers.

The \(i\)th individual \(C_{i,j}\) in generation \(j\) is represented as a concatenated sequence of rules, generated by the grammar as described in Section 3.1. A delimiter is used to separate two rules. The length of an individual is variable and ranges from 1 to 10 rules. If \(m \leq 10\) is the number of rules of individual \(C_{i,j}\) then its representation is as follows:

\[
C_{i,j} = rule_1, \ rule_2, \ rule_3, \ldots \ rule_m
\]

Each rule is composed of one or more conditions grouped by parenthesis, which may be nested. The number of conditions and the way they are grouped vary from rule to rule.

The method uses four GA operations, i.e., selection, mutation, crossover, and elitism, to evolve a population of individuals from one generation to the next. The selection is based on tournament selection of individuals where two individuals are randomly chosen to compete in the tournament; the individual with the best fitness wins and is chosen to pass its rules to the individuals in the next generation. Tournament selection, in contrast to roulette wheel selection, helps in keeping the diversity of the selected individuals [3]. In the elitism operation the best individual of generation \(j\) is passed to generation \(j + 1\) without any change. When passed by elitism, individuals do not have their fitness evaluated again. In the first generation the individual passed by elitism is a manually-designed scheduling policy that combines attributes used by the fixed-thresholds scheduling policy described in Section 2. In crossover, two individuals previously selected combine their rules to form two new individuals that will be evaluated in the next generation. The algorithm to combine the rules is presented in Figure 4. Note that for crossover, the order of the rules in new individuals is the same as in their parents and that the new individuals contain at least the same number of rules as the smallest parent and only one of them can be as long as the longest parent.

Three levels of mutation are used in our method: individual-level, rule-level, and condition-level. Individual-level mutation can be applied in three different ways: (1) if the individual has a medium size, the mutation randomly changes the position of one rule in the sequence; (2) if the individual is short, the mutation adds one
new rule; and (3) if the individual is long, the mutation deletes a randomly-chosen rule. An individual is considered short if it has less than four rules, long if it has more than seven rules, and medium otherwise. The rule-level mutation splits long rules. Since very long rules tend to be too restrictive, the method detects long rules with small usage frequency and splits them randomly the break point and keeping the order of nested parenthesis. The condition-level mutation replaces either operators or factors using knowledge-based mutation matrices that provide the probability that these changes may occur. Figures 5 and 6 show the mutation matrices used in the work presented in this paper. One or more levels of mutation can be applied to an individual, where the levels and their order are chosen randomly. A fixed fraction of individuals is mutated from one generation to the next.

![Mutation matrix](image1)

### 3.3 Computational Environment

The scheduling policies produced by the grammar are automatically encoded as Python subclasses by the code generator and integrated as a new scheduling policy in SimBA. SimBA uses the policies with the same parameters and flags used to run the manually-designed scheduling policies for a given VC project trace. When the simulation of the VC project ends, the parameters reported by SimBA include the total number of valid WUs (Throughput$_{WU}$) and average WUs per WU (AvgWUI$_{WU}$). These values are used by our method to evaluate the fitness of every single scheduling policy.

The evaluation of scheduling policies is designed as a master-slave system taking advantage of its inherent task parallelism, where the evaluation of every individual per generation is performed by one slave and the evolution of the complete population as well as the gathering of statistics is made by the master. We use the message passing interface (MPI) to communicate between master and slave processes and to synchronize the evolution of generations.

## 4. EXPERIMENTS AND RESULTS

### 4.1 Set-up of Experiments

We ran our evolutionary algorithm for 30 generations with a population size of 70 individuals (scheduling policies) and a mutation rate of 60%. Such a mutation rate has been chosen to prevent too rapid convergence. The computation was distributed across the nodes of a Beowulf cluster with 64 dual-core nodes each with a 2.0 GHz AMD Opteron processor, 256 GB RAM, and 10TB disk space. We trained our method with SimBA using performance traces from two BOINC projects, Predictor@Home with CHARMM and FightAIDS@Home. We tested the forty best scheduling policies with SimBA using performance traces belonging to other three BOINC projects, Predictor@Home with MFold, Human Proteome Folding and Genome Comparison. Note that SimBA uses traces to emulate the worker community, their error rates, timeout, etc. The features of the training and testing projects are shown in Table 3. In this table, Project indicates the name of the BOINC project, Simulated$_{hrs}$ is the length of the BOINC project for which the traces have been collected, Min$_{valid}$ is the minimum number of WUs whose results must agree to consider valid a WU, and Usage states whether the project was used for training or testing.

We compared the scheduling policies generated by the genetic algorithm (GA-designed) against four manually-designed scheduling policies: First-Come-First-Served (FCFS), fixed availability and reliability thresholds (F1Xr), variable availability and relia-

![Mutation matrix for operators](image2)
### Results

The overall fitness function used for our method is the combination of the two fitness functions of the two projects used for training, P@H with CHARMM and FightAIDS@Home. Since the values returned by the two fitness functions may differ by up to an order of magnitude, each of the two fitness values is normalized with respect to the best manually-designed scheduling policy for the specific training project that the value is associated with. Then the normalized values are averaged to compute the overall fitness. Figure 7 shows the overall fitness of the best individual per generation. The figure shows how the overall value of the fitness function is above one and constantly increasing as the population evolves. The fact that the fitness function is above one means that at least one of the two training fitness functions outperforms the best associated manually-designed scheduling policy.

Figures 8 and 9 show three stages of the evolution of the whole population of scheduling policies for P@H with CHARMM and FightAIDS@Home respectively (our training projects); the dots are associated to GA-designed scheduling policies. The goal here is to minimize the average WUIs per WU (x-axis) while maximizing the throughput (y-axis). The figures show that the populations move in a diagonal, from the bottom-right corner to the top-left corner, meeting the above described goal.

At the end of the training process, we selected the 40 GA-designed scheduling policies that reached the higher overall fitness value. From those 40 policies we automatically selected 10 policies for every testing project through a cross-selection process. The selection follows this procedure: Given a testing project (e.g., P@H with MFold), 10 GA-designed policies are selected by using the 40 best policies identified by the other two testing projects (in this case Human Proteome Folding and Genome Comparison) and choosing the 10 policies with the highest overall fitness computed for the latter two projects. We observed that only four policies were present in all the three set of 10-best policies of each testing project; in the rest of this paper we focus only on these four policies that we call GA1, GA2, GA3, and GA4. Figures 10, 11, 12, and 13 summarize the four policies and the associated frequency representing the

---

**Table 3: Training and testing VC projects**

<table>
<thead>
<tr>
<th>Project</th>
<th>Simulated hrs</th>
<th>Min_valid</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>P@H CHARMM</td>
<td>170</td>
<td>3</td>
<td>TRAINING</td>
</tr>
<tr>
<td>FightAIDS</td>
<td>550</td>
<td>3</td>
<td>TRAINING</td>
</tr>
<tr>
<td>P@H MFold</td>
<td>340</td>
<td>3</td>
<td>TESTING</td>
</tr>
<tr>
<td>Folding</td>
<td>600</td>
<td>11</td>
<td>TESTING</td>
</tr>
<tr>
<td>Genome</td>
<td>1200</td>
<td>3</td>
<td>TESTING</td>
</tr>
</tbody>
</table>

---

![Figure 6: Mutation matrix for factors](image)

![Figure 7: Evolution of the overall fitness achieved by the best individual](image)
number of times workers applying for computation are served by that rule. Note that the number of rules per policy ranges from two to four; for some policies the complexity of the rules is much higher than for the human-designed policies; and each rule is used by each project but with different frequencies. Also note that if none of the rules are true, one single WUI is assigned to the worker (in the figures this is represented by else).

For each testing project, Table 4 compares and contrasts the performance of the four GA-designed scheduling policies (GA1, GA2, GA3, and GA4) with the manually-designed scheduling policies. In the rows belonging to the manually-designed scheduling policies (FCFS, FIXT, VART, and WCG) the underlined values represent the best achieved value for the project in the associated column. In the rows belonging to the GA-designed scheduling policies the bold values are associated with the GA-designed policy that outperforms the best manually-designed scheduling policy for the project in the associated column. The percentage of improvement of the GA-designed scheduling policies with respect to the best manually-designed scheduling policy (marked as ref in Table 4) is reported in the Improve column. Positive values represent real improvements and negative values represent deterioration of the performance. The last column of the table (Average) shows the average improvement in percentage of the scheduling policy in the associated row across projects. The results presented in this section are discussed in Section 4.3.

4.3 Discussion

In general, the results presented in this paper show that none of the manually-designed scheduling policies works best for all projects. More specifically, if we consider the throughput, the policy based on variable thresholds is better for P@H with CHARMM and Human Proteome Folding; the FCFS policy is better for P@H with MFold and FightAIDS@Home; and the Fixed thresholds policy is better for Genome Comparison. In terms of average WUI per WU, the World-Community-Grid policy is more effective in reducing the number of errors and timeout for FightAIDS@Home and Human Proteome Folding.

On the other hand the GA-designed policies improve throughput across projects, in particular GA1 increases throughput across all the three testing projects (+9.1% for P@H with MFold, +12.2% for Human Proteome Folding, and +0.3% for Genome Comparison) while the other three GA-designed scheduling policies increase throughput for at least two of them. In terms of average WUI per WU, GA2 performs similarly to the best manually-designed scheduling policy across the three projects with an average of +0.3%. GA1, GA3 and GA4 have a similar performance to the best manually-designed scheduling policies for P@H with MFold and Genome Comparison but perform poorly for Human Proteome Folding. This loss in performance can be associated to a possible imbalance of the fitness function. Work in progress is addressing this issue.

5. RELATED WORK

Evolving sequences of rules for classification using genetic algorithms (GA) or genetic programming (GP) have been extensively studied in the past. Relevant work includes [5, 13, 24]. Only recently the scheduling problem has been addressed as a classification problem. Work in this field includes scheduling for open shop problems [9], scheduling in multiprocessors [15, 18], as well as single-machine and parallel systems [6, 7, 11, 12]. To our knowledge, no work has been done in the past for the design of scheduling policies in VC using GA or GP.

Goals and methodologies in related work are different from the work presented in this paper. Past work focuses on the minimization of the completion time for (1) the whole application, or (2) single tasks and their tardiness, i.e., the difference between when a task returns and its expected completion time. Dimopoulos and Zalzala in [6, 7] as well as Jakobovic and Budin in [11] target the minimization of tardiness by using genetic programming to evolve scheduling policies in the form of dispatching rules for a single-machine problem. Jakobovic et al. apply their method to parallel systems in [12]. In [18] and [15], the goal is to minimize the total execution time of the parallel application by using GA or GP. In [18], Seredybski and Zomaya base their method on convolution of cellular automata rather than rules and the combination of all the automata results is gathered in a global scheduling policy. In [15], Oudshoorn and Huang, outline communication and synchronization among task as part of their adapting scheduling process for multiprocessors. Because of the fact that VC systems are non-dedicated and volatile, task execution time is not as relevant as the overall throughput; therefore in this paper we target throughput.

The search for compiler heuristics based on different machine learning techniques that optimize the final execution time of a program is presented in [4, 17, 20]. The approach in [17] is similar to
R1: not(((last_rac_update+avg_turnaround)+
   (iops*availability_G))<=granted_credit)
R2: (granted_credit>avg_turnaround)
R3: else

Figure 10: GA1 rules (left) and their frequency usage (right) for the three testing VC projects

R1: ((availability_G)>=0.729)
R2: (82.084>(last_rac_update))
R3: ((granted_credit<91.57) and
   (availability_L<=0.85))
R4: (iops>=(6.1192*mean_exec_time))
R5: else

Figure 11: GA2 rules (left) and their frequency usage (right) for the three testing VC projects

R1: (acquired_wui>25.329)
R2: (not(((max_wui\_day<=claimed_credit) and
   (not(((ravg_credit<=0.53)and
   not((0.146<availability_L)))) and
   (83.592<(lifespan)))))
R3: (granted_credit<claimed_credit)
R4: (0.145>granted_credit)
R5: else

Figure 12: GA3 rules (left) and their frequency usage (right) for the three testing VC projects

R1: ((availability_G>(reliability_L*availability_
   and (availability_L<0.195))
R2: ((0.533*(availability_G))>=0.1026)
R3: not((iops>=ravg_credit))
R4: (reliability_L>=availability_G)
R5: else

Figure 13: GA4 rules (left) and their frequency usage (right) for the three testing VC projects
our method, but it targets the scheduling of instructions rather than tasks.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a distributed evolutionary algorithm for the design of scheduling policies in volunteer computing (VC), i.e., systems consisting of PCs connected to the Internet and belonging to the public. The proposed method produces scheduling policies that increase throughput across a variety of different VC projects, in contrast to manually-designed policies that are limited to increasing throughput for single projects. The method is also time-efficient, since it allows us to design effective policies in a time window of a week, while policies used in VC projects normally take several person-years for monitoring, tuning, and validating the scheduling rules.

Work in progress includes several extensions to the basic algorithm, including automatic update of the mutation matrices to avoid any manually-introduced biases, tuning the fitness function to better capture the cost related to replication of computation, extension to the rule-pruning scheme to allow for shorter and thus more human-understandable policies, and combination of the evolutionary search with a hill-climbing component. We will also work on extending the experimental evaluation of our system, developing scheduling policies for other projects and performing non-triviality tests to quantify the efficiency gains of the proposed methods over purely random search. Finally, we will perform extensive analysis of the policies developed by our system, trying to gain insights about the behavior of VC environments from them.

Acknowledgment

This work was supported by the National Science Foundation, grant #SCI-0506429, DAPLDS - a Dynamically Adaptive Protein-Ligand Docking System based on multi-scale modeling, and by the CONACyT fellowship #171595. The authors thank John Cavazos for his feedback, Kevin Reed and the WCG community for their help with the collection of the performance traces.

7. REFERENCES


<table>
<thead>
<tr>
<th>Trace</th>
<th>P@H</th>
<th>Improve</th>
<th>Human</th>
<th>Improve</th>
<th>Genome</th>
<th>Improve</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFold</td>
<td></td>
<td></td>
<td>Prot. Fold.</td>
<td></td>
<td>Comp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace</td>
<td>P@H</td>
<td>Improve</td>
<td>Human</td>
<td>Improve</td>
<td>Genome</td>
<td>Improve</td>
<td>Average</td>
</tr>
<tr>
<td>MFold</td>
<td></td>
<td></td>
<td>Prot. Fold.</td>
<td></td>
<td>Comp.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Comparison of throughput and average WUI per WU


