# 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 takes advantage of 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 efficient scheduling policies for VC projects involves subjective and time-demanding tuning that is driven by 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. In contrast with 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.

## Keywords

Distributed systems, Genetic algorithms, Global computing, Volatile systems

## 1. INTRODUCTION

Volunteer Computing (VC) is a paradigm that allows the use of 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 climatepredicton.net[1], 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. Finally, 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 work-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 often 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* for 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 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 differ-

[1] http://climateprediction.net

ent VC projects, we considered 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 compared and contrasted 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 used SimBA, a simulator of BOINC Applications [23]. SimBA allowed us 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. To prove that the quality of our results was produced by our evolutionary search method we compared the evolutionary based scheduling policies with a set of randomly-generated scheduling policies. Our results show that the proposed evolutionary method produces more general and effective scheduling policies than random-based scheduling methods.

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[2] (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 (Gflop).

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

| Project | Size of traces (days) | Number workers | Average size of WUI (Gflop) | Year |
|---|---|---|---|---|
| **P@H CHARMM** | 8 | 5093 | 8000 | 2004 |
| **P@H MFold** | 15 | 7810 | 10000 | 2004 |
| **FightAIDS@Home** | 25 | 35583 | 35000 | 2007 |
| **Proteome Folding** | 24 | 36590 | 47000 | 2007 |
| **Genome Comp.** | 50 | 30540 | 10000 | 2007 |

## 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 is 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. Examples of naive scheduling policies are: (1) First-Come-First-Served ($FCFS$): WUIs are sent to any host that applies for computation [1]; (2) Locality scheduling policy: WUIs are preferentially sent to hosts that already have the necessary data to accomplish the work [2]; and (3) Random assignment: WUIs are selected randomly. Knowledge-based scheduling policies look at the history of the worker applying for computation and the whole community. Examples of knowledge-based scheduling policies are as follows: (1) Fixed thresholds ($FIX_T$) checks the availability and reliability values of the requesting host; if they are above a certain predefined threshold, the scheduler assigns the requested work to that host [8]; (2) Variable thresholds ($VAR_T$) 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 number of requests generated by the hosts, then the thresholds decrease, otherwise they increase; and (3) World Community Grid scheduling policy ($WCG$) assigns computation based

[2]http://worldcommunitygrid.org

| | | |
|---|---|---|
| < **rule** > | := | $< logical\_expr > \mid < inequality >$ |
| < **logical_expr** > | := | $< rule > AND < rule > \mid NOT < rule >$ \| TRUE \| FALSE |
| < **inequality** > | := | $< arithmetic\_expr >< comp\_operator >< arithmetic\_expr >$ |
| < **arithmetic_expr** > | := | $< arithmetic\_expr >< arit\_operator >< arithmetic\_expr > \mid < operand >$ |
| < **operand** > | := | $< number > \mid < factor >$ |
| < **arit_operator** > | := | $+ \mid - \mid *$ |
| < **comp_operator** > | := | $> \mid < \mid >= \mid <=$ |
| < **number** > | := | $[0-1] \mid [0-100]$ uniform distributions |
| < **factor** > | := | $os \mid processor \mid availability\_L \mid reliability \mid availability\_G \mid wui\_valid \mid iops \mid flops \mid lifespan \mid claimed\_credit \mid granted\_credit \mid ravg\_credit \mid last\_rac\_update \mid avg\_turnaround \mid acquired\_wui \mid avg\_dedicated\_time \mid max\_tasks\_day \mid wui\_inprogress \mid app\_size\_wu \mid starving\_workers$ |

**Figure 1: Grammar used to build IF-THEN-ELSE rules**

on the average turnaround time of the worker, which is the average time a worker needs to return a result [3].

## 2.4 Genetic Algorithms

Genetic Algorithms (GA) [10] are based on the theory of evolution and natural selection of the fittest individuals. An individual is encoded as a sequence of symbols, usually organized in an array. Every symbol, or gene, represents a feature of the individual. Every individual is ranked by a fitness function. GA use genetic operations to evolve populations of individuals across generations. The evolution is carried out by selecting individuals based on their fitness and combining them to produce new individuals that will be evaluated in the next generation. Individuals may be modified using mutation operations. The evolutive process goes on for a fixed number of generations or until the fitness of at least one individual exceeds a given threshold.

# 3. METHODOLOGY

Our method is based on a distributed, evolutionary algorithm capable of searching a large space of scheduling policies looking for policies that outperform manually-designed scheduling policies. Our work includes a rigorous definition of scheduling policies (Section 3.1), a genetic algorithm used for the search, i.e., operations and fitness function (Section 3.2), and a computational environment in which we use our method (Section 3.3).

## 3.1 Scheduling Policies

In our method, we consider a large set of possible scheduling policies. Each scheduling policy is composed of a variable number of IF-THEN-ELSE rules driving the work assignment. We use BNF (Backus-Naur Form) grammar to model the structure of the rules. Figure 1 shows our formalism. The components of the grammar are defined as follows: A rule is a logical expression or an inequality. A logical expression can be two rules joined by AND, a negated rule, the logical constant TRUE, or the logical constant FALSE. An inequality is two arithmetic expressions joined by a comparison operator. Note that logical expressions and inequalities can produce only Boolean values as results. Arithmetic expressions may be two arithmetic expressions joined by an arithmetic operator, or an operand. An operand is a number or a factor. An arithmetic operator can be an addition, subtraction, or multiplication. Note that the operator division is omitted because it can be replaced by a multiplication by the inverse. A comparison operator may be greater than, less than, greater than or equal, or less than or equal. A number is a random value given by a uniform distribution between 0 and 1, or a uniform distribution between 0 and 100. A factor is a feature characterizing the worker community or the application; the description of the factors is given in Table 2. We selected these factors because they provide us with a quantitative approach to measure VC project performance and some of them are also used in the other scheduling policies. The set of factors has been chosen to be large and the evolutionary component of our method is in charge of discarding those factors that ultimately do not play any relevant role in improving project performance.

A parser uses the grammar to generate a scheduling policy that acts as an individual for our genetic algorithm. A code generator uses the individual to generate the scheduling policies used in the computational environment. Figure 2 shows an example of one individual produced by the parser and Figure 3 shows part of the class containing the IF-THEN-ELSE rules for this individual.

```
1.- (85.141 <= claimed_credit) AND
    (availability >= 17.92),
2.- ((flops - (0.264 + max_wui_day)) < 0.145)
    AND (os < 2) ,
3.- NOT (67.340 <= wui_valid) ,
4.- ((wui_successful > ravg_credit) * 0.111)
```

**Figure 2: Example of one individual with four rules**

```
if ((85.141<=host.claimed_credit) and
   (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_success/(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**

The ordered IF-THEN-ELSE rules within an individual are used by the scheduler for assigning WUIs to workers. The scheduler goes through the rules starting from the first and proceeding until it finds a rule whose preconditions are true. The amount of work assigned to the worker is related to the position of this rule. If a

[3]http://worldcommunitygrid.org

**Table 2: Factors used to characterize worker community and applications**

| Factors | Dependency | Description |
|---|---|---|
| $os$ | worker | Operating system: (1) Windows, (2) Linux, (3) Darwin |
| $processor$ | worker | Vendor: (1) AMD, (2) Intel, (3) PowerPC Mac, (4) Intel Mac |
| $availability_L$ | worker/application | WUI completed without error or timeout over WUI distributed since last worker connection |
| $reliability_L$ | worker/application | WUI valid over WUI collected since the last worker connection |
| $availability_G$ | worker/application | WUI completed without error or timeout over WUI distributed since worker joined the project |
| $reliability_G$ | worker/application | WUI valid over WUI since worker joined the project |
| $iops$ | worker | Integer operations per second |
| $flops$ | worker | Floating point operations per second |
| $lifespan$ | worker | Number of hours since the worker joined the project |
| $claimed\_credit$ | worker | Amount of credits claimed in the last connection |
| $granted\_credit$ | worker | Total amount of credits granted for valid results |
| $ravg\_credit$ | worker | Recent average of granted credits for valid results |
| $last\_rac\_update$ | worker | Last time the worker received credits |
| $avg\_turnaround$ | worker | Average time to return a WUI for a given worker |
| $acquired\_wui$ | worker/application | Number of WUIs given to the worker in the last connection |
| $avg\_dedicated\_time$ | worker | Average time dedicated to VC project |
| $max\_wui\_day$ | worker/application | Maximum number of WUIs the worker can get in one day |
| $wui\_inprogress$ | worker | Number of WUIs still in progress |
| $app\_size\_wu$ | application | Average size of WUs in flops |
| $starving\_workers$ | worker/application | Number of workers that did not receive work in the last unit of time |

worker requests $k$ WUIs and meets rule one, then it gets all the work requested. If rule one is not met, the scheduler moves to rule two and decreases the amount of work assigned based on the formula in Equation 1, where, given an individual of $m$ rules and the worker meeting rule $s$, the number of WUI assigned to the worker is $\tilde{k}$:

$$\tilde{k} = k - \frac{k * (s-1)}{m} \quad (1)$$

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 $ith$ 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:

$$C_{i,j} = \quad rule_1\ , \quad rule_2\ , \quad rule_3\ , \quad ... \quad rule_m$$

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 selected randomly 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 selecting 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

```
Select individuals C_{x,j} and C_{y,j}
with size_x and size_y number of
rules respectively.
Assuming that size_x ≥ size_y
Create empty individuals C_{x,j+1} and C_{y,j+1}
For i=1 to size_y
  Get rule i from C_{x,j}
  Get a random boolean variable. If 1 then:
     Append rule i from C_{x,j} to C_{x,j+1}
     Append rule i from C_{y,j} to C_{y,j+1}
  Otherwise:
     Append rule i from C_{x,j} to C_{y,j+1}
     Append rule i from C_{y,j} to C_{x,j+1}
If size_x > size_y
  For i=1 to size_x
     Get rule i from C_{x,j}
     Get a random boolean variable. If 1 then:
        Append rule i from C_{x,j} to C_{x,j+1}
     Otherwise:
        Append rule i from C_{x,j} to C_{y,j+1}
```

**Figure 4: Pseudocode of the crossover**

the levels and their order are chosen randomly. A fixed fraction of individuals is mutated from one generation to the next.

| Operators | $+$ | $-$ | $*$ | $>$ | $<$ | $\geq$ | $\leq$ |
|---|---|---|---|---|---|---|---|
| $+$ | 0 | 0.6 | 0.4 | 0 | 0 | 0 | 0 |
| $-$ | 0.7 | 0 | 0.3 | 0 | 0 | 0 | 0 |
| $*$ | 0.7 | 0.3 | 0 | 0 | 0 | 0 | 0 |
| $>$ | 0 | 0 | 0 | 0 | 0.1 | 0.7 | 0.2 |
| $<$ | 0 | 0 | 0 | 0.1 | 0 | 0.2 | 0.7 |
| $\geq$ | 0 | 0 | 0 | 0.5 | 0.1 | 0 | 0.4 |
| $\leq$ | 0 | 0 | 0 | 0.1 | 0.5 | 0.4 | 0 |

**Figure 5: Mutation matrix for operators**

In addition to the four GA operations, we use *pruning* to reduce the number of rules. The proposed method includes counters that monitor how many times a rule is applied; rules whose frequency is below a defined threshold are automatically removed from individuals. Long rules are possible but very unlikely to meet the threshold limits.

The fitness function is given by a multi-objective function that maximizes throughput and minimizes replication of computation due to (a) errors, i.e., network and computation failures; (b) timed out results, i.e., results too late for the simulation or never returned because of the worker volatility; and (c) invalid results, i.e., affected by malicious attacks, hardware malfunctions, or software modifications. A WU is successfully completed if it is not affected by errors or timeout. A WU is valid if at least $min\_valid$ WUIs have been completed successfully and their results agree, i.e., their results either are equal because the Homogeneous Redundancy (HR) policy is applied [22] or are within a certain range if no HR is used. The throughput, $Throughput_{WU}$, is the total number of successfully completed, valid WUs at the end of a simulation. When errors, timeout, or invalid results take place, the system reacts by generating new WUIs for the faulty WU. $min\_valid$ and the maximum number of WUIs per WU that a VC project can generate are both defined by the project designer: usually $min\_valid$ ranges from two to four and the maximum number of WUIs ranges from five to eight. The amount of replication is captured by the average WUIs distributed per WU, $AvgWUI_{WU}$, over the whole simulation; the fewer faulty WUs, the closer this value is to $min\_valid$.

The fitness function used in our method is the ratio between throughput, $Throughput_{WU}$, and average WUIs distributed per WU,$AvgWUI_{WU}$. The fitness function is given in Equation 2.

$$fitness = \frac{Throughput_{WU}}{AvgWUI_{WU}} \quad (2)$$

## 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 WUIs 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 done 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 and number of individuals per population were chosen to prevent premature convergence. The number of generations was set to 30 because of time constraints to perform our experiments; however we observed that 30 generations were sufficient for the convergence of the fitness function. 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 WUIs whose results must agree to consider valid a WU, and *Usage* states whether the project was used for training or testing.

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

| *Project* | *Simulated_ hrs* | *Min_valid* | *Usage* |
|---|---|---|---|
| P@H CHARMM | 170 | 3 | TRAINING |
| FightAIDS | 550 | 3 | TRAINING |
| P@H MFold | 340 | 3 | TESTING |
| Folding | 600 | 11 | TESTING |
| Genome | 1200 | 3 | TESTING |

Our experiments included two comparisons. First, we compared the scheduling policies generated by the genetic algorithm (GA-

| Factors | A to F | G to S |
|---|---|---|
| A to F | T1 | 0 |
| G to S | 0 | T2 |

Organization of the matrix

| T1 | | | | | | | |
|---|---|---|---|---|---|---|---|
| | Factors | A | B | C | D | E | F |
| A | $os$ | 0 | 0.4 | 0.15 | 0.15 | 0.15 | 0.15 |
| B | $processor$ | 0.4 | 0 | 0.15 | 0.15 | 0.15 | 0.15 |
| C | $availability_L$ | 0.1 | 0.1 | 0.1 | 0.3 | 0.3 | 0.1 |
| D | $reliability_L$ | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.3 |
| E | $availability_G$ | 0.1 | 0.1 | 0.3 | 0.1 | 0.1 | 0.3 |
| F | $reliability_G$ | 0.1 | 0.1 | 0.1 | 0.3 | 0.3 | 0.1 |

| T2 | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Factors | G | H | I | J | K | L | M | N | O | P | Q | R | S |
| G | $iops$ | 0 | 0.2 | 0 | 0 | 0.2 | 0 | 0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| H | $flops$ | 0.2 | 0 | 0 | 0 | 0.2 | 0 | 0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| I | $lifespan$ | 0 | 0 | 0 | 0.2 | 0.5 | 0 | 0 | 0.2 | 0.1 | 0 | 0 | 0 | 0 |
| J | $claimed\_credit$ | 0.05 | 0.05 | 0 | 0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0 |
| K | $granted\_credit$ | 0.05 | 0.05 | 0.1 | 0.1 | 0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0 |
| L | $ravg\_credit$ | 0 | 0 | 0 | 0.1 | 0.1 | 0 | 0.1 | 0.3 | 0.1 | 0.1 | 0.1 | 0.1 | 0 |
| M | $last\_rac\_update$ | 0 | 0 | 0 | 0.1 | 0 | 0.1 | 0 | 0.4 | 0 | 0.3 | 0 | 0.1 | 0 |
| N | $avg\_turnaround$ | 0 | 0 | 0 | 0.1 | 0.1 | 0.1 | 0.2 | 0 | 0.1 | 0.1 | 0.2 | 0.1 | 0 |
| O | $acquired\_wui$ | 0 | 0 | 0 | 0 | 0 | 0.3 | 0.1 | 0.2 | 0 | 0 | 0.3 | 0.1 | 0 |
| P | $avg\_dedicated\_time$ | 0.1 | 0.1 | 0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0 | 0.1 | 0.1 | 0 |
| Q | $max\_wui\_day$ | 0.05 | 0.05 | 0 | 0.2 | 0.1 | 0.1 | 0 | 0 | 0.3 | 0 | 0 | 0.2 | 0 |
| R | $wui\_inprogress$ | 0 | 0 | 0 | 0 | 0 | 0.1 | 0.1 | 0.1 | 0.2 | 0.3 | 0.2 | 0 | 0 |
| S | $app\_size\_wu$ | 0 | 0.2 | 0.2 | 0 | 0.1 | 0 | 0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0 |

**Figure 6: Mutation matrix for factors**

designed) against four manually-designed scheduling policies: First-Come-First-Served ($FCFS$), fixed availability and reliability thresholds ($FIX_T$), variable availability and reliability thresholds ($VAR_T$), and the World Community Grid scheduling policy ($WCG$). All these policies are described in Section 2. Most BOINC projects use the FCFS policy. The threshold-based policies have been applied only in simulated environments [8]. Note that the World Community Grid policy is the result of several person-years of design effort and includes sophisticated correlated rules. Second, we performed a non-triviality test in which we compared our GA-designed scheduling policies against 2100 scheduling policies randomly generated. The purpose of this comparison is to estimate whether the quality of our results is accidental or due to the proposed evolutionary approach. To create the random policies we used the same set of rules used by our method (Figure 1), but we did not apply the evolutionary operators to evolve those policies. The number of randomly generated policies, i.e., 2100, was selected to be equal to the total number of policies generated during the evolutionary process (i.e., 30 generations $\times$ 70 individuals per generation = 2100).

## 4.2 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.

Figures 7 and 8 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.

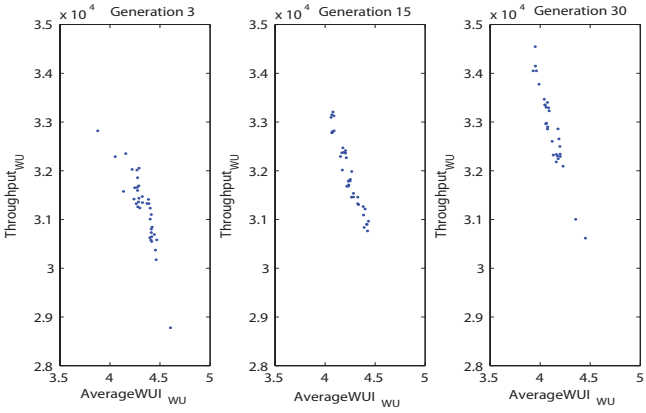


**Figure 7: Evolution across generations 3, 15, and 30 of GA-designed scheduling policies for P@H with CHARMM training dataset**

At the end of the training process, we selected the 40 GA-designed scheduling policies that reached high overall fitness values. The number of GA-designed policies was arbitrarily chosen (about 2% of the total population). A larger number of policies might ultimately produce better results at the cost of decreasing the merit of the training process. By contrast, the latter should narrow the search space of scheduling policies. 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

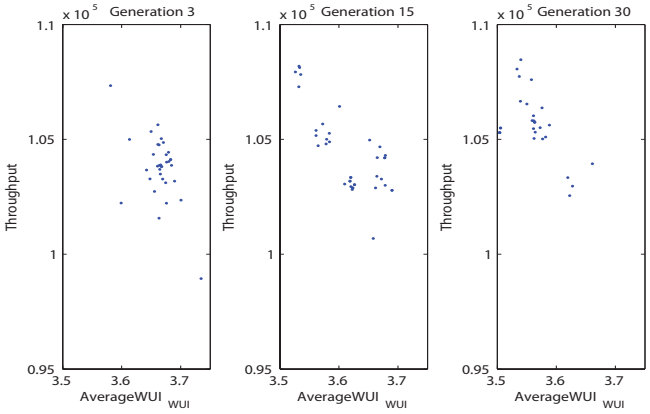


**Figure 8: Evolution across generations 3, 15, and 30 of GA-designed scheduling policies for FightAIDS@Home training dataset**

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 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, Figures 13.a, .b, and .c compare and contrast the performance of the manually-designed scheduling policies ($FCFS$, $FIX_T$, $VAR_T$ , and $WCG$), the four GA-designed scheduling policies ($GA1$, $GA2$, $GA3$, and $GA4$), and the randomly generated scheduling policies ($BestRand$ is the best random policy per project, and $MeanRand$ is the average over the 2100 samples). Note that the best random policy, $BestRand$, refers to a different policy in each of the three projects. In contrast with our method, which was able to find four common scheduling policies in the first 10-best policies, none of the first 50-best randomly generated scheduling policies was common across projects, evidencing their lack in generalization capability.

Figure 13.a presents throughput and average WUIs for P@H MFold, Figure 13.b presents throughput and average WUIs for Human Proteome Folding, and Figure 13.c presents throughput and average WUIs for Genome Comparison (note that the last two projects are part of World Community Grid). The dashed lines in each figure serve as reference to identify areas of the search space in which either GA-designed scheduling policies or randomly generated policies, if falling, perform better than the best manually-designed scheduling policies for that project. Policies that fall above the horizontal dashed line outperform the best manually-designed scheduling policy in terms of throughput. Policies that fall to the left of the vertical dashed line outperform the best manually-designed scheduling policy in terms of error reduction (average WUIs)

Table 4 quantitatively reports the same results as Figure 13. In the rows belonging to the manually-designed scheduling policies ($FCFS$, $FIX_T$, $VAR_T$, and $WCG$), the underlined values represent the best achieved value for the project in the associated column. Neither $BEST_{Rand}$, the best randomly-generated scheduling policy, nor the average of the set of 2100 random policies, $MEAN_{Rand}$, offers the best result in any of the three testing projects according to both of the two metrics. In the rows belonging to the GA-designed scheduling policies ($GA1$, $GA2$, $GA3$, and $GA4$) 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 performance. The last column of the table ($Average$) shows the average improvement in percentage of the scheduling policy in the associated row across projects. Because each project has a different best random scheduling policy, the $Average$ column of $BEST_{Rand}$ is left empty. 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. Note that the goal of our work is to find scheduling policies that are general enough to improve performance across different projects or within the same project at different times of the project lifespan. The dynamic behavior of VC projects might indeed result in scenarios in which the same project has different resource requirements at different times.

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.

Finally, Figure 13 shows that none of the randomly generated scheduling policies outperformed both the GA-designed and manually designed scheduling policies. The results in this paper show that the simple random generation of scheduling policies is not effective and that the success of our GA-policies is due to the evolutionary process. By using an *'intelligent'* combination of factors, such a process leads to general, effective scheduling rules for VC projects.

# 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 classifica-

```
R1:   not(((last_rac_update+avg_turnaround)+
     (iops*availability_G))<=granted_credit)
R2:  (granted_credit>avg_turnaround)
R3:   else
```

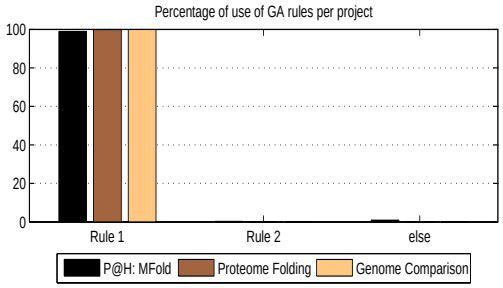


**Figure 9: GA1 rules (left) and their frequency usage (right) for the three testing VC projects**

```
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
```

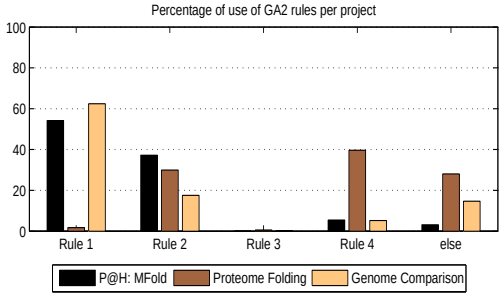


**Figure 10: GA2 rules (left) and their frequency usage (right) for the three testing VC projects**

```
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
```

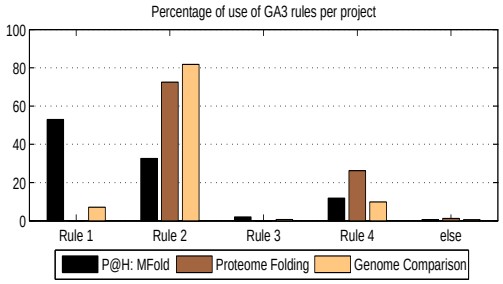


**Figure 11: GA3 rules (left) and their frequency usage (right) for the three testing VC projects**

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

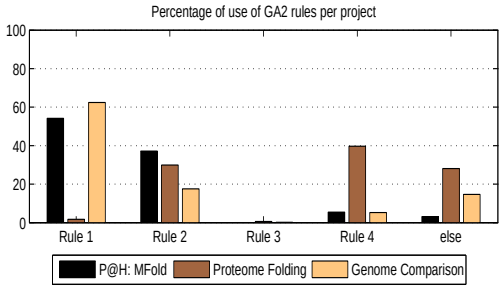


**Figure 12: GA4 rules (left) and their frequency usage (right) for the three testing VC projects**

tion 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

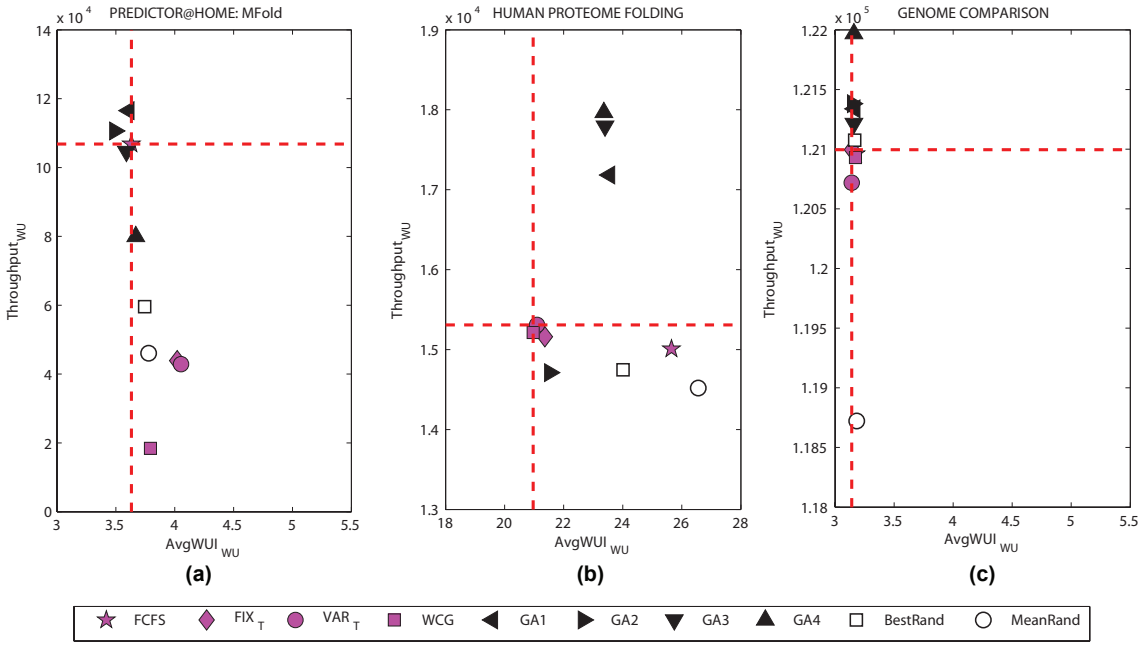


**Figure 13: Throughput and average WUIs per WU for the three testing projects**

**Table 4: Comparison of throughput and average WUI per WU**

| Throughput | | | | | | | |
|---|---|---|---|---|---|---|---|
| Trace | P@H MFold | *Improve* | Human Prot. Fold. | *Improve* | Genome Comp. | *Improve* | *Average* |
| $FCFS$ | 106833 | $ref.$ | 15008 | $-2.0\%$ | 120960 | $-0.0\%$ | $-0.6\%$ |
| $FIX_T$ | 43888 | $-58.9\%$ | 15160 | $-1.0\%$ | 120995 | $ref.$ | $-19.9\%$ |
| $VAR_T$ | 42898 | $-59.8\%$ | 15309 | $ref.$ | 120719 | $-0.2\%$ | $-20.0\%$ |
| $WCG$ | 18446 | $-82.7\%$ | 15213 | $-0.6\%$ | 120930 | $-0.1\%$ | $-27.8\%$ |
| $BEST_{Rand}$ | 59589 | $-44.2\%$ | 14747 | $-3.7\%$ | 121075 | $0.1\%$ | $-$ |
| $MEAN_{Rand}$ | 46056 | $-56.9\%$ | 14521 | $-5.1\%$ | 118721 | $-1.9\%$ | $-21.3\%$ |
| $GA1$ | **116533** | $9.1\%$ | **17183** | $12.2\%$ | **121338** | $0.3\%$ | $+7.2\%$ |
| $GA2$ | **110612** | $3.5\%$ | 14712 | $-3.9\%$ | **121381** | $0.3\%$ | $-0.03\%$ |
| $GA3$ | 104657 | $-2.0\%$ | **17794** | $16.2\%$ | **121215** | $0.2\%$ | $+4.8\%$ |
| $GA4$ | 79960 | $-25.2\%$ | **17967** | $17.4\%$ | **121968** | $0.8\%$ | $-2.3\%$ |
| Average WUIs per WU | | | | | | | |
| Trace | P@H MFold | *Improve* | Human Prot. Fold. | *Improve* | Genome Comp. | *Improve* | *Average* |
| $FCFS$ | 3.633 | $ref.$ | 25.650 | $-22.3\%$ | 3.176 | $-1.1\%$ | $-7.8\%$ |
| $FIX_T$ | 4.021 | $-10.7\%$ | 21.362 | $-1.9\%$ | 3.143 | $-0.0\%$ | $-4.2\%$ |
| $VAR_T$ | 4.054 | $-11.6\%$ | 21.096 | $-0.6\%$ | 3.142 | $ref.$ | $-4.0\%$ |
| $WCG$ | 3.794 | $-4.4\%$ | 20.968 | $ref.$ | 3.173 | $-1.0\%$ | $-1.8\%$ |
| $BEST_{Rand}$ | 3.745 | $-3.1\%$ | 24.010 | $-14.5\%$ | 3.165 | $-0.8\%$ | $-$ |
| $MEAN_{Rand}$ | 3.779 | $-4.0\%$ | 26.560 | $-26.7\%$ | 3.183 | $-1.3\%$ | $-10.6\%$ |
| $GA1$ | **3.611** | $0.6\%$ | 23.553 | $-12.3\%$ | 3.167 | $-0.8\%$ | $-4.1\%$ |
| $GA2$ | **3.492** | $3.9\%$ | 21.527 | $-2.7\%$ | 3.149 | $-0.2\%$ | $+0.3\%$ |
| $GA3$ | **3.590** | $1.2\%$ | 23.400 | $-11.6\%$ | 3.160 | $-0.6\%$ | $-3.6\%$ |
| $GA4$ | 3.670 | $-1.0\%$ | 23.360 | $-11.4\%$ | 3.160 | $-0.6\%$ | $-4.3\%$ |

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 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 the manually-designed policies that are limited to increasing throughput for single projects, and to randomly generated policies that consistently perform poorly and do not generalize across projects. Our 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, a systematic selection of parameters, 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 run-time GA-based scheduling policies for other projects. 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, and Kevin Reed and the WCG community for their help with the collection of the performance traces.

## 7. REFERENCES

[1] D. Anderson, E. Corpela, and R. Walton. High-performance task distribution for volunteer computing. In *Proc. of the 1st IEEE Int. Conference on e-Science and Grid Technologies*, 2005.

[2] F. Bellosa. Locality-information-based scheduling in shared-memory multiprocessors. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, 1996.

[3] T. Blickle and L. Thiele. A mathematical analysis of tournament selection. In *Proc. of the 6th Int. Conference on Genetic Algorithms*, 1995.

[4] J. Cavazos and E. B. Moss. Inducing heuristics to decide whether to schedule. In *Proc. of the ACM SIGPLAN 04 Conference on Programming Language Design and Implementation*, 2004.

[5] K. A. De-Jong and W. M. Spears. Learning Concept Classification Rules using Genetic Algorithms. In *Proc. of the 12th Int. Conference on Artificial Intelligence IJCAI-91*, 1991.

[6] C. Dimopoulos and A. M. S. Zalzala. A genetic programming heuristic for the one-machine total tardiness problem. In *Proc. of the Congress on Evolutionary Computation*, 1999.

[7] C. Dimopoulos and A. M. S. Zalzala. Investigating the use of genetic programming for a classic one-machine scheduling problem. *Advances in Engineering Software*, 32:489–498, 6 2001.

[8] T. Estrada, D. Flores, M. Taufer, P. J. Teller, A. Kerstens, and D. Anderson. The effectiveness of threshold-based scheduling policies in BOINC projects. In *Proc. of the 2nd IEEE Int. Conference on e-Science and Grid Technologies (eScience)*, 2006.

[9] E. Hart, P. Ross, and D. Corne. Evolutionary scheduling: A review. *Genetic Programming and Evolvable Machines*, 6(2):191–220, 2005.

[10] J. H. Holland. Genetic algorithms and classifier systems: foundations and future directions. In *Proc. of the 2nd Int. Conference on Genetic Algorithms and their Application*, 1987.

[11] D. Jakobović and L. Budin. Dynamic scheduling with genetic programming. In *Proc. of the 9th European Conference on Genetic Programming*, 2006.

[12] D. Jakobović, L. Jelenković, and L. Budin. Genetic programming heuristics for multiple machine scheduling. In *Proc. of the 10th European Conference on Genetic Programming*, 2007.

[13] J. R. Koza. Concept formation and decision tree induction using the genetic programming paradigm. In *Proc. of 1st Workshop on Self Adaptivity in Grid Computing*, 1991.

[14] A. D. MacKerell Jr., B. Brooks, C. L. Brooks III, L. N. B. Roux, Y. Won, and M. Karplus. *CHARMM: The Energy Function and Its Parametrization with an Overview of the Program*, volume 1. John Wiley & Sons, 1998.

[15] M. J. Oudshoorn and L. Huang. Evolving toward an optimal scheduling solution through adaptivity. *J. Parallel Distrib. Comput.*, 62(7):1203–1222, 2002.

[16] V. S. Pande et al. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68(91), 2003.

[17] D. Puppin. Adapting convergent scheduling using machine learning. In *Proc. of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[18] F. Seredynski and A. Y. Zomaya. Coevolution and evolving parallel cellular automata - based scheduling algorithms. In *Selected Papers from the 5th European Conference on Artificial Evolution*, Springer-Verlag, 2002.

[19] J. Skolnick, A. Kolinski, and A. R. Ortiz. Monsster: a method for folding globular proteins with a small number of distance restraints. *J. of Molecular Biology*, 265(2):217–241, 1997.

[20] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, 2003.

[21] M. Taufer, C. An, A. Kerstens, and C. L. Brooks III. Predictor@home: A protein structure prediction supercomputer based on global computing. *IEEE Transactions on Parallel and Distributed Systems*, 17(8):786–796, 2006.

[22] M. Taufer, D. P. Anderson, P. Cicotti, and C. L. Brooks. Homogeneous redundancy: a technique to ensure integrity of molecularsimulation results using public computing. In *Proc. of the IEEE Int. Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[23] M. Taufer, A. Kerstens, T. Estrada, D. A. Flores, and P. J. Teller. SimBA: a discrete event simulator for performance prediction of volunteer computing projects. In *Proc. of the Int. Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*, 2007.

[24] C. Zhou, W. Xiao, T. M. Tirpak, and P. C. Nelson. Evolving accurate and compact classification rules with gene expression programming. *IEEE Transactions on Evolutionary Computation*, 7(6):519–531, 2003.