# Combining Machine Learning and Genetic Algorithms to Solve the Independent Tasks Scheduling Problem

**Conference Paper** · June 2017

DOI: 10.1109/CYBConf.2017.7985766

Bernabe Dorronsoro
Universidad de Cádiz
**179** PUBLICATIONS **3,358** CITATIONS

Frédéric Pinel
University of Luxembourg
**29** PUBLICATIONS **512** CITATIONS

# Combining Machine Learning and Genetic Algorithms to Solve the Independent Tasks Scheduling Problem

Bernabé Dorronsoro
*School of Engineering*
*University of Cádiz, Spain*
*Email: bernabe.dorronsoro@uca.es*

Frédéric Pinel
*University of Luxembourg, Luxembourg*
*Email: frederic.pinel@uni.lu*

*Abstract*—**We propose a new accurate and fast hybrid parallel optimization algorithm for the independent tasks scheduling problem. The new technique combines the Virtual Savant (VS) with a parallel genetic algorithm (called PA-CGA). VS is an optimization framework based on machine learning that learns from a reference set of (pseudo-)optimal solutions how to solve the problem, providing accurate results in extremely low run times. We propose in this work the use of VS to generate a highly accurate initial population for the PA-CGA. Results show how initializing the population with VS (we test two versions of VS, differing on its training process) significantly increases the accuracy of the PA-CGA, compared to two other population initialization techniques: random and using a state-of-the-art heuristic.**

## 1. Introduction

Virtual Savant (VS) is a new optimization method [1] based on Machine Learning (ML) that allows computers autonomously learn how to solve a problem. It is inspired by people displaying the savant syndrome, who present outstanding abilities, as rapid calculation, despite their low Intelligence Quotient (IQ). For instance, some savants can enumerate and/or identify huge prime numbers without even knowing how to multiply. It is believed that they use pattern recognition rules to solve such problems [2], [3]. Therefore, VS makes use of ML techniques to discover the patterns that generate the desired solutions for a set of problem instances. These solutions could be either optimal solutions, generated by some reference algorithm or, simply, any observations. Once the system learns to solve the problem, it can efficiently and accurately solve new unseen problem instances in a massively parallel way, just by applying the discovered patterns.

VS has been only applied so far to solve the independent task scheduling problem [1]. In that paper, VS was successfully used to parallelize the well-known MinMin heuristic [4]. It could reproduce the solutions of MinMin with high accuracy, with over 80% correct tasks assignments on unseen problem instances, for some of the tested problem classes. Moreover, VS was highly competitive with the MinMin heuristic it learnt from in terms of the quality of solutions found, clearly outperforming it in some cases.

The independent task scheduling problem is an NP-complete problem that requires significant computation to find good solutions [5]. However, schedulers are expected to provide accurate solutions in short computation times. Therefore, heuristic and metaheuristic approaches arise as practical candidate solver algorithms to this problem. However, the best existing solutions are not able to promptly provide accurate solutions yet, especially in the case of large problem instances. To overcome this problem, some parallel algorithms have been developed. Some outstanding parallel algorithms for this problem are surveyed later in this paper. In [6], we proposed the Parallel Asynchronous Cellular Genetic Algorithm (PA-CGA), a novel parallel genetic algorithm that incorporates problem knowledge at the levels of population initialization and local search. The algorithm is described in Section 4.2, and it provides accurate solutions in a fixed 90 seconds wall time.

The main contribution of this work is the proposal of a novel hybrid algorithm that combines VS and PA-CGA. Specifically, we propose the use of VS to generate a diverse and highly accurate initial population of solutions, based on the recommendations of the Support Vector Machine (SVM) implemented in it. This initial population will be later evolved by the PA-CGA parallel genetic algorithm, especially designed for the considered problem [6]. The proposed hybrid algorithm, called VSGA, is able to outperform the original PA-CGA algorithm in most cases.

The structure of this paper is as follows. Next, we define the problem considered in this work. After that, some of the main existing techniques proposed in the literature for this problem are surveyed in Section 3. Section 4 describes the proposed VSGA hybrid algorithm, as well as VS and PA-CGA methods. Our main results are summarized in Section 5, and we finish with our main conclusions and lines for future work in Section 6.

## 2. The Independent Tasks Scheduling Problem

Task scheduling is a family of problems that capture the most important needs of datacenters and computational grids for efficiently allocating tasks to resources in a dynamic environment. Therefore, several versions of the problem can be formulated according to the needs of such applications.

In this work, we consider a version of the problem that arises quite frequently in parameter sweep applications, such as Monte-Carlo simulations [7]. In these applications, many tasks with no interdependencies are generated and submitted to be executed in the datacenter. In fact, more generally, the scenario in which independent tasks are submitted to a datacenter or a grid system is quite natural given that users independently submit their tasks and expect an efficient allocation of them. We notice that efficiency means to allocate tasks as fast as possible and to optimize some criterion, such as makespan or flowtime. Makespan is among the most important optimization criteria of a grid system. Indeed, it is a measure of its productivity (throughput).

The independent tasks scheduling problem considers a set of independent non-preemptive computing tasks to be processed on a number of heterogeneous machines. Tasks cannot be split across machines, and machines can only execute one task at a time. Assuming that the computing time needed to perform a task is known (assumption that is usually made in the literature [8], [9], [10]), we use the Expected Time to Compute (ETC) model by Braun et al. [8] to formalize the instance definition of the problem as follows:

- A *number* of independent (user/application) *tasks* to be scheduled.
- A *number* of heterogeneous *machine* candidates to participate in the planning.
- The *workload* of each task (in millions of instructions).
- The *computing capacity* of each machine (in *MIPS*).
- Ready time $ready_m$ indicates when machine $m$ will have finished the previously assigned tasks.
- The ETC matrix ($nb\_tasks \times nb\_machines$) in which $ETC[t][m]$ is the expected execution time of task $t$ on machine $m$.

The optimization problem consists in finding the tasks to machines assignments that minimize the *makespan*, defined as the finishing time of the last task:

$$\max\{completion[m] \mid m \in Machines\} \ , \qquad (1)$$

where *completion[m]* is the time when machine $m$ will finish processing the previously assigned tasks, as well as of those already planned. Formally, for a machine $m$ and a schedule $S$, the completion time of $m$ is defined as:

$$completion[m] = ready_m + \sum_{t \in S(m)} ETC[t][m] \ , \qquad (2)$$

where $ready_m$ represents the time when machine $m$ will finish all the previously assigned tasks (we consider $ready_m = 0$ for all machines in this work), and $S(m)$ is the set of tasks assigned to machine $m$.

Makespan is a well-known optimization criterion, among the most important ones, of a distributed system; it is a measure of its productivity (throughput). A solution to the problem is an assignment, and can be represented by a vector, where $solution[t] = m$ represents that the task of index $t$ is assigned to machine of index $m$.

The ETC is assumed given [8], [9], [10], and the instances we study are randomly generated according to the procedure in [8]. In the present study, all tasks are chosen as highly heterogeneous. The machines are consistent (a machine cannot be slower than another for one task, and faster for another task), and high and low heterogeneous machine sets are chosen for the evaluation (called $hihi$ and $hilo$ problems, respectively). These problem classes cover most scenarios we can find in real data-centers.

## 3. Related Work

The independent task scheduling problem has been addressed by a large number of heuristics and metaheuristics in the literature [1], [4], [6], [8], [11]. One of the most widely used batch mode dynamic heuristic for this problem is the well-known MinMin algorithm [4]. MinMin starts with a set of all unmapped tasks. Then, it establishes the minimum completion time for all unassigned tasks in any available machine. After that, the task with the overall minimum expected completion time is selected and assigned to the corresponding machine. The task is then removed from the set and the process is repeated until all tasks are mapped. The run time of MinMin is $O(m{\cdot}n^2)$, where n is the number of tasks and m the number of machines. Other simple heuristics for this problem with similar complexity exist, as MaxMin and Sufferage [8]. Tabak et al. [12] later improved the design of these heuristics to reduce their complexity to $O(m \cdot n \log n)$, but at the cost of some accuracy loss.

In addition to the mentioned heuristics, the problem has been addressed with metaheuristics, providing more accurate results at the cost of longer execution times [6], [11]. In order to reduce the computation time of heuristics and metaheuristics for this problem, some parallel approaches have been presented. Pinel et al. designed in [13] the first parallel implementation of MinMin for GPU architectures, together with a cellular genetic algorithm. Later, Iturriaga et al. proposed other simple heuristics for GPU to solve the same problem [14]. Our PA-CGA algorithm is a more accurate (but less scalable) parallel metaheuristic for multi-core processors that makes use of problem knowledge [6].

Finally, we presented in [1] the VS, a new method to automatically generate massively parallel algorithms from scratch, and its application to the considered scheduling problem. The method is based on ML to learn the behavior of some reference algorithm. It was applied to learn from MinMin algorithm, and it could accurately reproduce its solutions on unseen problem instances.

## 4. The method

We present in Section 4.3 our proposed accurate and efficient optimization solution for the considered problem. Before, sections 4.1 and 4.2 describe the VS and PA-CGA optimizers, which are the building blocks of our proposed hybrid approach.

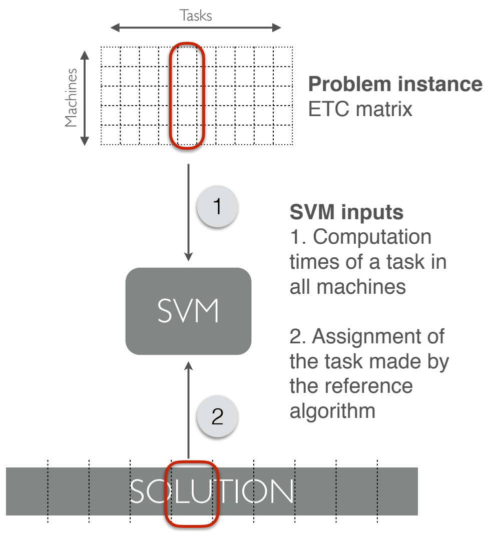

Tasks
Machines
Problem instance
ETC matrix
1
SVM inputs
1. Computation times of a task in all machines
SVM
2. Assignment of the task made by the reference algorithm
2
SOLUTION

Figure 1: Learning process of the Virtual Savant.

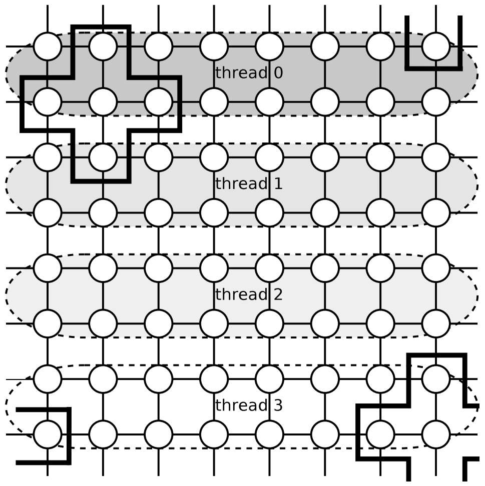


Figure 2: Partition of an 8x8 population over 4 threads and the neighborhood of two sample solutions (thick solid lines).

## 4.1. The Virtual Savant

The Virtual Savant is a novel technique to autonomously create parallel programs to solve optimization problems [1]. Based on ML, it is trained to automatically learn how to solve an optimization problem from a set of observations, normally generated by some optimization algorithm. After the training process, VS is able to accurately reproduce the solutions the reference algorithms produce on new, unseen, problem instances. Therefore, given an algorithm, VS is able to generate a new, completely different program, that reproduces its behavior, but in a highly efficient massively parallel way, thanks to its parallel pattern recognition engine.

The VS is inspired by people with the autistic savant syndrome, who show brilliant capabilities (as rapid calculation) despite their very low IQ. It is believed that they use pattern recognition rules to get outstanding solving capabilities to problems that, in many cases, they do not even understand [15], [16].

Our implementation of VS for the considered independent tasks scheduling problem makes use of one SVM to learn the machine every task must be assigned to. The training process is visualized in Fig. 1. For every task, the SVM receives as an input both the column of the ETC matrix corresponding to that task, containing the expected run time of the task on every machine, and the desired machine assignment to that task, obtained from the solution provided by the reference algorithm (i.e., the observation).

After the learning process is finished, VS runs multiple instances of the SVM in parallel (as many as the number of tasks in the problem instance) to generate the probabilities of assigning all tasks to every machine. These assignments are made independently of the assignments of the other tasks, so all processes can be independently run in parallel. It was reported in [1] a high accuracy of the SVM predictions, correctly assigning over 80% of the tasks (considering the maximum assignment probability for every task) in average, on 100 unseen problems, for some of the studied problem classes.

Once this probabilities vector is built, VS starts a refinement step, in which a solution is built from the generated probabilities vector. This is a process based on fitness that allows improving the quality of solutions, amending the inaccuracies that might happen in the SVM predictions. In our previous work [1], a random search was implemented in the refinement step. In this work, we make use of a highly specialized metaheuristic for the considered problem. The method is explained next.

## 4.2. The PA-CGA Genetic Algorithm

The PA-CGA, was presented as an accurate parallel hybrid genetic algorithm for the independent tasks scheduling problem in [6]. In PA-CGA, problem knowledge is used in the population initialization method (one solution of the initial population is generated with the accurate MinMin heuristic) as well as to apply the H2LL local search (explained later) to every offspring in order to find a local optimum.

The population of PA-CGA is structured in a toroidal grid, limiting the interaction of individuals to its local neighbors [17]. It is sketched in Fig. 2, where the allowed relationships between individuals (represented as circles) in the population are shown with lines connecting them. The effect of implementing such a population structure is a smooth diffusion of the information of good individuals through the population, allowing to preserve the population diversity for longer [18], [19], compared to other unstructured populations. The algorithm is designed for multi-core processors, and the population is logically partitioned into a number of contiguous blocks of the same size. Each block is evolved by one thread (see Fig. 2) by means of the application of the recombination and mutation genetic operators, as well as the H2LL local search. The neighborhood of an individual may include individuals from other population blocks, allowing the genetic information of individuals to cross block boundaries. The neighborhoods of two sample individuals are drawn in Fig. 2 with thick solid lines to show the interactions between individuals from different blocks, as well as the toroidal feature of the population (bottom-right individual's neighborhood).

The different threads evolve their population block independently, meaning that they do not wait on the other threads to complete their generation (the evolution of all the individuals in their block) before pursuing their evolution. Hence, if a breeding loop takes longer for an individual of a given thread, the individuals evolved by the other threads may go through more generations.

**Algorithm 1** Pseudo-code for PA-CGA.

```
1: $t_0 \leftarrow time()$; {record the start time}
2: $pop \leftarrow setup\_pop()$; {initialize population}
3: $par \leftarrow setup\_blocks(pop)$; {set parameters for all threads}
4: $do\_parallel(initial\_evaluation, par)$; {each thread evaluates its block}
5: $do\_parallel(evolve, par, t_0)$; {each thread evolves its block, see Algorithm 2}
```

Algorithms 1 and  2 provide a more detailed description of the algorithm. Function $do\_parallel(f, parm)$ means that $f(parm)$ is executed by all threads in parallel, but on different data items. All threads join before the next instruction.

**Algorithm 2** Pseudo-code for $evolve()$.

```
1: while $time() - t_0 \leq time$ do
2:     for all $ind$ in a thread's block do
3:         $neigh \leftarrow get\_neighborhood(ind)$;
4:         $parents \leftarrow select(neigh)$;
5:         $offspring \leftarrow recombine(p\_comb, parents)$;
6:         $mutate(p\_mut, offspring)$;
7:         $H2LL(p\_ser, iter, offspring)$;
8:         $evaluate(offspring)$;
9:         $replace(ind, offspring)$;
10:    end for
11: end while
```

Function $initial\_evaluation()$ computes the fitness of all individuals in the initial population. The stop condition for the considered scheduling problem is a wall clock time.

After performing the recombination and mutation operators, we also apply the H2LL local search operation. After that, $evaluate()$ computes the makespan of the schedule, and replaces the evolved solution in the population by the new one according to some criterion.

The local search operator moves a task, randomly chosen, from the most loaded machine (i.e., the one with the highest completion time) to a selected candidate machine, choosen among the $N$ least loaded ones (the most loaded machine's completion time defines makespan). Among these candidate machines, we select the one whose new completion time, with the addition of the task moved, is the smallest of all the candidates. This new completion time must also remain inferior to the makespan. Algorithm 3 describes this operator.

The completion times are often used in the local search, therefore we maintain up-to-date completion times for all machines to speed up computations. The $evaluate()$ function of Algorithm 2 only finds the maximum completion time. The completion times are kept up-to-date by each operator (recombination, mutation, local search). Such updates are efficiently performed by adding or removing the ETC of a task on a machine to the appropriate completion time.

**Algorithm 3** Pseudo-code for $H2LL$, our local search.

```
1: for all $iter$ iterations do
2:     sort $machines$ on ascending completion time
3:     $task \leftarrow$ random task from last $machines$;
4:     $best\_score \leftarrow CT$[last $machines$]; {makespan}
5:     for all $mac$ in $pop\_size/2$ first $machines$ do
6:         $new\_score \leftarrow CT[mac] + ETC[task][mac]$;
7:         if $new\_score < best\_score$ then
8:             $best\_mac \leftarrow mac$;
9:             $best\_score \leftarrow new\_score$;
10:        end if
11:    end for
12:    move $task$ to $best\_mac$ if any
13: end for
```

### 4.3. VSGA: The Hybridization of VS and PA-CGA

We present in this section the VSGA optimization algorithm for the independent tasks scheduling problem. This is a hybrid of VS and PA-CGA metaheuristic. As it is sketched in Fig. 3, it first executes the prediction step of VS to obtain the machines assignment probabilities vector, which is later used to generate an initial population of the PA-CGA algorithm. The population is created by randomly assigning values to all variables, according to the probabilities defined by VS. This way, we create a diverse population of highly accurate solutions, hopefully located in the most promising regions of the search space.

Once the initial population is created, PA-CGA is evolved for a fixed amount of time, and the best solution in the population is reported as the output of the algorithm. Therefore, we are replacing the generic heuristic methods originally designed for the refinement step of VS by a highly accurate parallel metaheuristic, specifically designed for the considered problem.

## 5. Experiments

We summarize in this section our results. In our experiments, we adopted the same configuration of the original PA-CGA, that was tuned for these same problem instances (see Table 1). We are using a population of 256 individuals, arranged into an square grid. The population is randomly initialized, except for one individual, which is the result of the *MinMin* heuristic. To evolve every individual, its parents are chosen from its linear 5 (L5) neighborhood (also called Von Neumann or NEWS neighborhood), composed of the

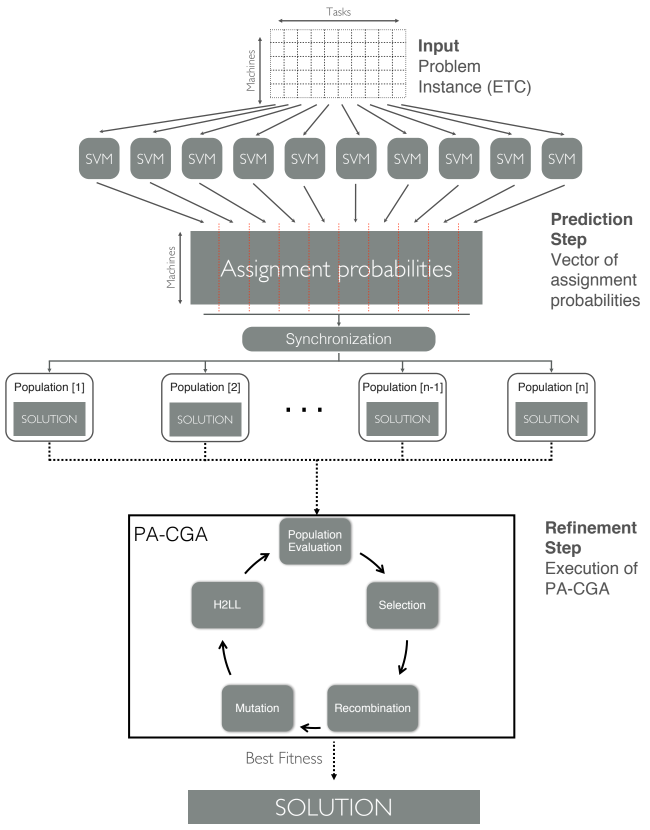


Figure 3: Global view of the proposed VSGA hybrid algorithm.

4 nearest individuals (at its north, east, west, and south positions in the population), plus the considered one. This neighborhood is chosen to smoothen the speed in which the information of good individuals spreads through the population, as well as to reduce concurrent memory access (at the borders of the different population blocks). The 2 best neighbors are selected as parents. The recombination operator used is the well-known two-point crossover ($tpx$). The mutation operator moves one randomly chosen task to a randomly chosen machine. The newly generated offspring replaces the current individual if it improves the fitness value. The number of threads used in all our experiments is 4, all threads running on one processor. Regarding VS, we used the SVM that was trained in our previous paper [1] for this same problem. The processor used for the experiments is an Intel Xeon E5 2670 clocked at 2.6 GHz. This processor has 8 cores. Regarding the number of iterations of the local search, we did different experiments with 0 (i.e., no local search is performed), 1, and 10 iterations. Finally, we run two series of experiments with 0.5 and 10 seconds run (the termination condition in the original paper where PA-CGA was presented was an execution time of 90 seconds).

Our results are summarized in tables 2 and 3 for the instances with high and low heterogeneity in the computational capacity of machines, respectively. They are average

TABLE 1: Parameterization of PA-CGA.

| | |
|---|---|
| *Population* | $16 \times 16$ |
| *Population initialization* | Min-min (1 ind) |
| | Initialization according to VS results |
| *Neighborhood* | linear 5 |
| *Selection* | best 2 |
| *Recombination* | two-point crossover, $p\_comb = 1.0$ |
| *Mutation* | random move, $p\_mut = 1.0$ |
| *Local search* | H2LL, $p\_ser = 1.0$, $iter = 0, 1, 10$ |
| *Replacement* | replace if better |
| *Stopping criterion* | 0.5 & 10 seconds, wall time |
| *Number of Threads* | 4 |

values on 30 different instances, and 100 independent runs of the algorithms on every instance (meaning that we performed 432,000 experiments in total). We present results for 0.5 and 10 seconds runs, performing 0, 1, and 10 iterations of the local search for 12×4, 128×4, and 512×16 problems. The compared algorithms are VSGA trained from PA-CGA (i.e., $\text{VSGA}_{\text{GA}}$) and MinMin ($\text{VSGA}_{\text{MM}}$), the original PA-CGA, and PA-CGAr, being its only difference with respect to PA-CGA that the initial population is completely randomly generated. The results of the best algorithm are emphasized in **bold font** for every combination of termination condition, iterations of the local search, and problem solved.

We can see in Table 2 that the two VSGA versions clearly outperform PA-CGAr in all cases. With respect to PA-CGA, it stands out as the best option for the 512×16 *hihi* problems, with small average differences of 1.23% and 0.77% with respect to $\text{VSGA}_{\text{GA}}$ and $\text{VSGA}_{\text{MM}}$, respectively. The smallest differences were found in the case of 10 iterations of H2LL and 10 seconds run, which are reduced to 0.61% and 0.27% for $\text{VSGA}_{\text{GA}}$ and $\text{VSGA}_{\text{MM}}$, respectively. The worst performance of the VSGA algorithms for the biggest problems are in concordance with our results in [1], where VS obtained the worst accurate values for these problems. We can see how the accuracy of the algorithms improves when increasing the wall time from 0.5 to 10 seconds, as it could be expected. However, the maximum improvement achieved was around 6%, for PA-CGAr with 5 iterations of H2LL. Regarding the other algorithms including problem knowledge in the population initialization process, the difference is never higher than 0.7%.

The results for the *hilo* problems are even more favorable to the proposed VSGA versions, as it is shown in Table 3. In this case, $\text{VSGA}_{\text{GA}}$ performs the best for all combinations of H2LL iterations and execution time tested in 12×4 and 128×4 problems. In the case of the biggest problem, PA-CGA is the best algorithm for 0 and 1 iterations cases, but it is outperformed by $\text{VSGA}_{\text{MM}}$ when 10 iterations of H2LL are performed. Therefore, the two proposed VSGA algorithms report the best results in 14 cases out of 18.

We would like to emphasize how the two VSGA algorithms proposed obtain in 0.5 seconds better results than those that PA-CGA can achieve in 10 seconds for problems 12×4 and 128×4. There are only a few exceptions in the

TABLE 2: Computational results for hihi problems.

| Problem Size | Algorithm | 0.5 seconds run 0 iter | 1 iter | 10 iter | 10 seconds run 0 iter | 5 iter | 10 iter |
|---|---|---|---|---|---|---|---|
| $12 \times 4$ | $\text{VSGA}_{\text{GA}}$ | **2,0874E2** | 2,0693E2 | 2,0664E2 | **2,0870E2** | 2,0692E2 | 2,0663E2 |
| | | **5,0622E1** | 4,8402E1 | 4,8135E1 | **5,0619E1** | 4,8374E1 | 4,8132E1 |
| | $\text{VSGA}_{\text{MM}}$ | 2,0897E2 | **2,0682E2** | 2,0663E2 | 2,0887E2 | **2,0680E2** | **2,0663E2** |
| | | 5,1118E1 | **4,8102E1** | 4,8130E1 | 5,0734E1 | **4,8113E1** | **4,8128E1** |
| | PA-CGAr | 2,0945E2 | 2,0707E2 | 2,0663E2 | 2,0946E2 | 2,0706E2 | 2,0663E2 |
| | | 4,9276E1 | 4,8350E1 | 4,8132E1 | 4,9103E1 | 4,8399E1 | 4,8129E1 |
| | PA-CGA | 2,0933E2 | 2,0708E2 | **2,0663E2** | 2,0928E2 | 2,0708E2 | 2,0663E2 |
| | | 4,9467E1 | 4,8354E1 | **4,8129E1** | 4,9503E1 | 4,8406E1 | 4,8130E1 |
| $128 \times 4$ | $\text{VSGA}_{\text{GA}}$ | **1,9450E3** | **1,9405E3** | **1,9350E3** | **1,9446E3** | **1,9398E3** | **1,9337E3** |
| | | **1,3129E2** | **1,3075E2** | **1,3006E2** | **1,3095E2** | **1,3047E2** | **1,2993E2** |
| | $\text{VSGA}_{\text{MM}}$ | 1,9595E3 | 1,9521E3 | 1,9385E3 | 1,9592E3 | 1,9514E3 | 1,9361E3 |
| | | 1,3413E2 | 1,3338E2 | 1,3078E2 | 1,3421E2 | 1,3319E2 | 1,3036E2 |
| | PA-CGAr | 2,2202E3 | 1,9815E3 | 1,9401E3 | 2,2211E3 | 1,9793E3 | 1,9370E3 |
| | | 1,5100E2 | 1,3592E2 | 1,3051E2 | 1,4966E2 | 1,3590E2 | 1,3045E2 |
| | PA-CGA | 1,9826E3 | 1,9723E3 | 1,9400E3 | 1,9826E3 | 1,9714E3 | 1,9368E3 |
| | | 1,3737E2 | 1,3548E2 | 1,3120E2 | 1,3742E2 | 1,3530E2 | 1,3067E2 |
| $512 \times 16$ | $\text{VSGA}_{\text{GA}}$ | 1,7095E3 | 1,7002E3 | 1,6933E3 | 1,7091E3 | 1,6982E3 | 1,6821E3 |
| | | 3,0497E1 | 3,0429E1 | 3,0375E1 | 3,0311E1 | 3,0408E1 | 3,0227E1 |
| | $\text{VSGA}_{\text{MM}}$ | 1,7010E3 | 1,6917E3 | 1,6856E3 | 1,7005E3 | 1,6902E3 | 1,6764E3 |
| | | 3,0897E1 | 3,0791E1 | 3,0089E1 | 3,1299E1 | 3,0911E1 | 2,9891E1 |
| | PA-CGAr | 2,2133E3 | 1,9644E3 | 1,8096E3 | 2,2047E3 | 1,8291E3 | 1,7115E3 |
| | | 3,2523E1 | 3,4099E1 | 3,1327E1 | 3,2326E1 | 3,3304E1 | 3,1281E1 |
| | PA-CGA | **1,6812E3** | **1,6773E3** | **1,6803E3** | **1,6805E3** | **1,6763E3** | **1,6719E3** |
| | | **3,0521E1** | **3,0147E1** | **3,0551E1** | **3,0558E1** | **3,0213E1** | **3,0024E1** |

TABLE 3: Computational results for hilo problems.

| Problem Size | Population Initialization | 0.5 seconds run 0 iter | 1 iter | 10 iter | 10 seconds run 0 iter | 5 iter | 10 iter |
|---|---|---|---|---|---|---|---|
| $12 \times 4$ | $\text{VSGA}_{\text{GA}}$ | **2,9465E2** | **2,9295E2** | **2,9224E2** | **2,9466E2** | **2,9291E2** | **2,9221E2** |
| | | **5,2640E1** | **5,2335E1** | **5,2000E1** | **5,2651E1** | **5,2314E1** | **5,2007E1** |
| | $\text{VSGA}_{\text{MM}}$ | 2,9575E2 | 2,9344E2 | 2,9227E2 | 2,9569E2 | 2,9343E2 | 2,9223E2 |
| | | 5,3082E1 | 5,2545E1 | 5,2056E1 | 5,3092E1 | 5,2538E1 | 5,2045E1 |
| | PA-CGAr | 2,9674E2 | 2,9357E2 | 2,9230E2 | 2,9658E2 | 2,9360E2 | 2,9223E2 |
| | | 5,3091E1 | 5,2454E1 | 5,2033E1 | 5,3069E1 | 5,2488E1 | 5,2005E1 |
| | PA-CGA | 2,9663E2 | 2,9365E2 | 2,9229E2 | 2,9666E2 | 2,9354E2 | 2,9222E2 |
| | | 5,3034E1 | 5,2473E1 | 5,2011E1 | 5,3060E1 | 5,2448E1 | 5,1998E1 |
| $128 \times 4$ | $\text{VSGA}_{\text{GA}}$ | **3,0341E3** | **3,0322E3** | **3,0292E3** | **3,0339E3** | **3,0317E3** | **3,0278E3** |
| | | **2,0257E2** | **2,0234E2** | **2,0223E2** | **2,0253E2** | **2,0233E2** | **2,0217E2** |
| | $\text{VSGA}_{\text{MM}}$ | 3,0487E3 | 3,0435E3 | 3,0328E3 | 3,0484E3 | 3,0424E3 | 3,0305E3 |
| | | 2,0419E2 | 2,0375E2 | 2,0272E2 | 2,0422E2 | 2,0371E2 | 2,0249E2 |
| | PA-CGAr | 3,1349E3 | 3,0679E3 | 3,0361E3 | 3,1348E3 | 3,0643E3 | 3,0322E3 |
| | | 2,0714E2 | 2,0374E2 | 2,0288E2 | 2,0695E2 | 2,0319E2 | 2,0266E2 |
| | PA-CGA | 3,0645E3 | 3,0574E3 | 3,0354E3 | 3,0643E3 | 3,0560E3 | 3,0316E3 |
| | | 2,0519E2 | 2,0432E2 | 2,0271E2 | 2,0533E2 | 2,0414E2 | 2,0244E2 |
| $512 \times 16$ | $\text{VSGA}_{\text{GA}}$ | 3,1024E3 | 3,0969E3 | 3,0960E3 | 3,1017E3 | 3,0958E3 | 3,0946E3 |
| | | 7,5394E1 | 7,5030E1 | 7,4962E1 | 7,5295E1 | 7,5094E1 | 7,4827E1 |
| | $\text{VSGA}_{\text{MM}}$ | 3,0986E3 | 3,0932E3 | **3,0923E3** | 3,0979E3 | 3,0923E3 | **3,0912E3** |
| | | 7,5334E1 | 7,5222E1 | **7,5101E1** | 7,5303E1 | 7,5132E1 | **7,4996E1** |
| | PA-CGAr | 3,1887E3 | 3,1770E3 | 3,1598E3 | 3,1879E3 | 3,1724E3 | 3,1355E3 |
| | | 7,6849E1 | 7,6390E1 | 7,5537E1 | 7,6668E1 | 7,6449E1 | 7,4519E1 |
| | PA-CGA | **3,0921E3** | **3,0896E3** | 3,1049E3 | **3,0915E3** | **3,0890E3** | 3,1012E3 |
| | | **7,4633E1** | **7,4512E1** | 7,6580E1 | **7,4597E1** | **7,4489E1** | 7,5161E1 |

10 iterations case. Therefore, the population initialization mechanism based on VS really helps the algorithm to start from highly promising regions of the search space and quickly find accurate solutions. This is a very interesting result, especially for the considered problem, for which good quality solutions are required in really low times. In addition, the generated population provides enough diversity of solutions to allow the algorithm to continue progressing during the evolution and avoid local optima. The comparison between the results after 0.5 and 10 seconds run with no local search confirm that.

If we compare the performance of the algorithms for *hihi* and *hilo* problems, we can see that it is more favorable for the two VSGA algorithms in the latter case. This agrees with the fact that VS was found to be more accurate for *hilo* problems in [1].

We also compared the overall performance of every algorithm, aggregating all results for all problems. The Friedman rank of the algorithms is presented in Table 4. As it can be seen, the two overall best algorithms are $\text{VSGA}_{\text{MM}}$ and $\text{VSGA}_{\text{GA}}$ with 10 iterations of H2LL, in that order, followed by PA-CGA with 10 iterations of H2LL too. Indeed, the two VSGA algorithms outperform PA-CGA and PA-CGAr with the same number of iterations of the local search, both for 0.5 and 10 seconds runs. The only exception is the case of PA-CGA with no local search, outperforming $\text{VSGA}_{\text{MM}}$ when the wall time is set to 0.5 seconds (but this happens with low difference in rank value). The worst performing algorithm in all cases is PA-CGAr, the only one that is not using problem knowledge in the population initialization process.

TABLE 4: Friedman rankings of the algorithms

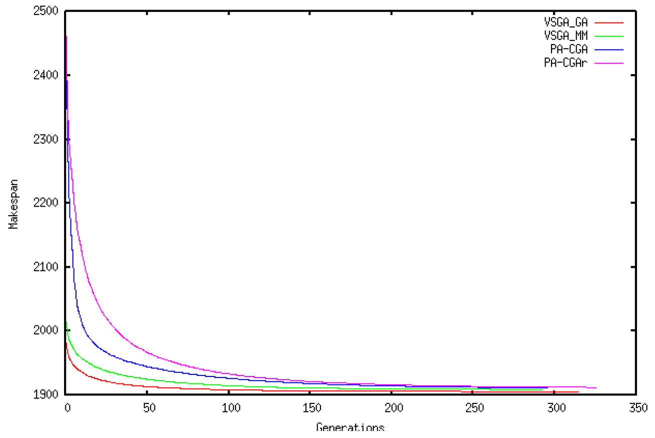
| 0.5 seconds run | | 10 seconds run | |
|---|---|---|---|
| Algorithm | Ranking | Algorithm | Ranking |
| $\text{VSGA}_{\text{MM}}$ – 10 iter | 2.94 | $\text{VSGA}_{\text{MM}}$ – 10 iter | 2.74 |
| $\text{VSGA}_{\text{GA}}$ – 10 iter | 3.01 | $\text{VSGA}_{\text{GA}}$ – 10 iter | 2.90 |
| PA-CGA – 10 iter | 4.31 | PA-CGA – 10 iter | 3.78 |
| $\text{VSGA}_{\text{GA}}$ – 1 iter | 5.08 | $\text{VSGA}_{\text{GA}}$ – 1 iter | 5.30 |
| $\text{VSGA}_{\text{MM}}$ – 1 iter | 5.71 | $\text{VSGA}_{\text{MM}}$ – 1 iter | 5.82 |
| PA-CGA – 1 iter | 5.79 | PA-CGAr – 10 iter | 5.68 |
| PA-CGAr – 10 iter | 6.07 | PA-CGA – 1 iter | 5.92 |
| $\text{VSGA}_{\text{GA}}$ – 0 iter | 7.37 | $\text{VSGA}_{\text{GA}}$ – 0 iter | 7.73 |
| PA-CGA – 0 iter | 8.20 | $\text{VSGA}_{\text{MM}}$ – 0 iter | 8.37 |
| $\text{VSGA}_{\text{MM}}$ – 0 iter | 8.24 | PA-CGA – 0 iter | 8.42 |
| PA-CGAr – 1 iter | 9.49 | PA-CGAr – 1 iter | 9.53 |
| PA-CGAr – 0 iter | 11.77 | PA-CGAr – 0 iter | 11.80 |

We also compared $\text{VSGA}_{\text{MM}}$ with 10 iterations of H2LL, the best overall performing algorithm, against the other ones according to Holm statistical test [20]. The obtained adjusted $p-$values are presented in Table 5. Those values $\leq$ 0.05 stand for statistical significant differences between the algorithms. We can see that the algorithm statistically outperforms all other studied algorithms but $\text{VSGA}_{\text{GA}}$ with 10 iterations of H2LL.

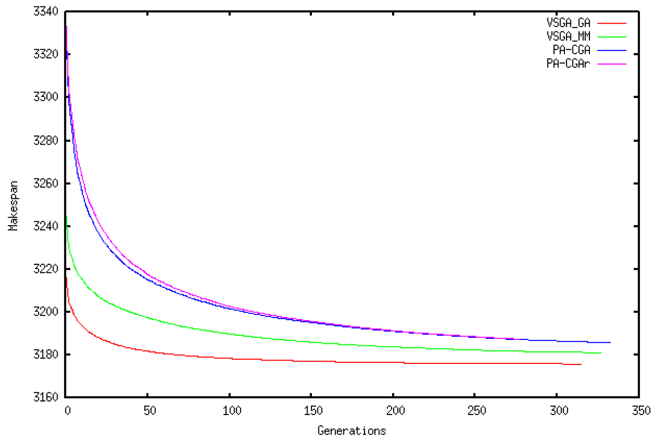
TABLE 5: Adjusted $p-$values for the Holm statistical test with 95% confidence

| 0.5 seconds run | | 10 seconds run | |
|---|---|---|---|
| algorithm | $p-$value | algorithm | $p-$value |
| $\text{VSGA}_{\text{GA}}$ – 10 iter | 0.8665 | $\text{VSGA}_{\text{GA}}$ – 10 iter | 0.6663 |
| PA-CGA – 10 iter | 6.6484E-4 | PA-CGA – 10 iter | 0.0123 |
| PA-CGAr – 10 iter | 1.4364E-15 | PA-CGAr – 10 iter | 3.7533E-14 |
| $\text{VSGA}_{\text{GA}}$ – 1 iter | 6.6484E-8 | $\text{VSGA}_{\text{GA}}$ – 1 iter | 5.0405E-11 |
| $\text{VSGA}_{\text{MM}}$ – 1 iter | 1.5754E-12 | $\text{VSGA}_{\text{MM}}$ – 1 iter | 2.6269E-15 |
| PA-CGA – 1 iter | 3.5984E-13 | PA-CGA – 1 iter | 2.9039E-16 |
| PA-CGAr – 1 iter | 1.8938E-65 | PA-CGAr – 1 iter | 2.0179E-70 |
| $\text{VSGA}_{\text{GA}}$ – 0 iter | 2.0709E-30 | $\text{VSGA}_{\text{GA}}$ – 0 iter | 1.6221E-38 |
| $\text{VSGA}_{\text{MM}}$ – 0 iter | 4.5590E-43 | $\text{VSGA}_{\text{MM}}$ – 0 iter | 9.3791E-49 |
| PA-CGA – 0 iter | 1.3767E-42 | PA-CGA – 0 iter | 1.3232E-49 |
| PA-CGAr – 0 iter | 5.2244E-118 | PA-CGAr – 0 iter | 1.3692E-124 |

To finish this section, we show in Fig. 4 the evolution of the average fitness of the population during the run, in every generation. Results are averaged after 100 runs for one single instance of size 128×4 for each considered problem class: *hihi* and *hilo*. It can be seen in the *hihi* case how the average fitness of the initial population of the two VSGA versions is over 15% better, compared to the PA-CGA and PA-CGAr algorithms. This fact, together with the diversity of solutions in it, allows the VSGA algorithms performing a faster convergence, therefore achieving higher quality solutions, outperforming both PA-CGA and PA-CGAr algorithms. However, in the *hilo* problems, the average fitness of the initial population is less than 5% better for VSGA algorithms, but the difference with PA-CGA and PA-CGAr at the end of the evolution is more important than in the *hihi* case. This is because the solutions generated by VS are highly close to those of the reference algorithm, but the small number of inaccuracies have a strong impact on the fitness value. However, we can see how the algorithm can overcome such inaccuracies, finding highly fitted solutions at the end of the run. Finally, we can see that PA-CGA and PA-CGAr perform similarly in this latter case, so the algorithm could not benefit much from adding the MinMin solution to the initial population in this case.



(a) 128×4 *hihi* problem



(b) 128×4 *hilo* problem

Figure 4: Convergence plots after 0.5 seconds of the considered algorithms for 128×4 instances

## 6. Conclusions and Future Work

We present in this paper a novel hybrid algorithm that combines an accurate parallel genetic algorithm, PA-CGA, for the independent task scheduling problem with a pioneer optimization technique, called the Virtual Savant (VS) and based on ML, that is used here to generate its initial population. The VS method makes use of parallel pattern recognition to learn from a reference algorithm how to build solutions to the considered problem. Once it is trained, it can quickly generate, in a massively parallel way, similar solutions to those of the reference algorithm on unseen problem instances.

We evaluate here two versions of our proposed hybrid algorithm, called VSGA, differing only on the algorithm used in the VS learning process. The two versions of VSGA were compared against two versions of PA-CGA: the original one (in which the initial population is randomly created but one solution that is built using MinMin heuristic), plus another version that starts from a completely random initial population. Six parameterizations were studied for every algorithm, representing different intensities of exploration of local neighborhoods and execution run lengths. We observed an overall better performance of the VSGA algorithms, supported by our statistical study at 95% confidence level.

As future work, we are focussing on enhancing the VS method to improve its accuracy on the biggest studied problem, for which the performance of VSGA method is comparable to PA-CGA. We will also work on hybridizing VS with other methods that would allow exploiting its massive parallelism.

## Acknowledgments

Bernabé Dorronsoro would like to acknowledge the Spanish MINECO and European FEDER for the support provided under contracts TIN2014-60844-R (the SAVANT project) and RYC-2013-13355.

## Acronyms

| | | |
|---|---|---|
| **ETC** | Expected Time to Compute | 2 |
| **IQ** | Intelligence Quotient | 1 |
| **ML** | Machine Learning | 1 |
| **PA-CGA** | Parallel Asynchronous Cellular Genetic Algorithm | 1 |
| **SVM** | Support Vector Machine | 1 |
| **VS** | Virtual Savant | 1 |

## References

[1] F. Pinel and B. Dorronsoro, "Savant: Automatic generation of a parallel scheduling heuristic for map-reduce," *The International Journal of Hybrid Intelligent Systems*, vol. 11, no. 4, pp. 287–302, 2014.

[2] L. Bouvet, S. Donnadieu, S. Valdois, C. Caron, M. Dawson, and L. Mottron, "Veridical mapping in savant abilities, absolute pitch, and synesthesia: an autism case study," *Frontiers in psycology*, vol. 5, p. 106, 2014.

[3] N. Charness, E. M. Reingold, M. Pomplun, and D. M. Stampe, "The perceptual aspect of skilled performance in chess: Evidence from eye movements," *Memory & Cognition*, vol. 29, no. 8, pp. 1146–1152, 2001.

[4] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[5] E. Horowitz and S. Sahni, "Exact and approximate algorithms for scheduling nonidentical processors," *J. ACM*, vol. 23, pp. 317–327, 1976.

[6] F. Pinel, B. Dorronsoro, and P. Bouvry, "A new parallel asynchronous cellular genetic algorithm for scheduling in grids," in *Nature Inspired Distributed Computing (NIDISC) workshop of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2010, p. 206b.

[7] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments," in *Heterogeneous Computing Workshop*, 2000, pp. 349–363.

[8] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810–837, 2001.

[9] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *ECE Technical Reports*, p. 270, 1992.

[10] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *Concurrency, IEEE*, vol. 6, no. 3, pp. 42–50, 1998.

[11] F. Xhafa, J. Carretero, B. Dorronsoro, and E. Alba, "A tabu search algorithm for scheduling independent jobs in computational grids," *Computing and Informatics Journal*, vol. 28, pp. 1001–1014, 2009.

[12] E. K. Tabak, B. B. Cambazoglu, and C. Aykanat, "Improving the performance of independenttask assignment heuristics MinMin, MaxMin and Sufferage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1244–1256, 2014.

[13] F. Pinel, B. Dorronsoro, and P. Bouvry, "Solving very large instances of the scheduling of independent tasks problem on the GPU," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 101–110, 2013.

[14] S. Iturriaga, S. Nesmachnow, F. Luna, and E. Alba, "A parallel local search in CPU/GPU for scheduling independent tasks on large heterogeneous computing systems," *The Journal of Supercomputing*, vol. 71, no. 2, pp. 648–672, 2015.

[15] L. Mottron, L. Bouvet, A. Bonnel, F. Samson, J. A. Burack, M. Dawson, and P. Heaton, "Veridical mapping in the development of exceptional autistic abilities," *Neurosci. Biobehav. Rev.*, vol. 37, no. 3, pp. 209–228, 2012.

[16] H. Welling, "Prime number identification in idiots savants: Can they calculate them?" *J. Autism Dev. Disord.*, vol. 24, no. 2, pp. 199–207, 1994.

[17] E. Alba and B. Dorronsoro, *Cellular Genetic Algorithms*. Springer, 2008.

[18] B. Dorronsoro and P. Bouvry, "Improving classical and decentralized differential evolution with new mutation operator and population topologies," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 1, pp. 67–98, 2011.

[19] E. Alba and B. Dorronsoro, "The exploration/exploitation tradeoff in dynamic cellular genetic algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 2, pp. 126–142, 2005.

[20] S. García, D. Molina, M. Lozano, and F. Herrera, "A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: A case study on the CEC'2005 special session on real parameter optimization," *Journal of Heuristics*, vol. 15, pp. 617–644, 2009.