Parallel Virtual Savant for the Heterogeneous Computing Scheduling Problem

Juan Carlos de la Torre[a], Renzo Massobrio[a,b], Patricia Ruiz[a], Sergio Nesmachnow[b], Bernabé Dorronsoro[a,*]

[a] *Escuela Superior de Ingeniería, Universidad de Cádiz, Cádiz, Spain*
[b] *Universidad de la República, Uruguay*

Abstract

We present in this work the first parallel implementation of Virtual Savant (VS), a novel optimization method that is able to quickly generate pseudo-optimal solutions to a given combinatorial problem, thanks to its parallel pattern recognition engine. The proposed parallel implementation does not require any information exchange between the threads during the run, they just get/send the required information before/after the execution. This design allows for a flexible algorithm that can perform efficiently on both shared- and distributed-memory systems. Our implementation uses both OpenMP for parallel architectures and MPI for distributed environments, which can efficiently make use of both kind of systems. The performance of VS is extensively analyzed on four different computing infrastructures, varying the number of threads used on each considered architecture. In addition, we propose a simulator to accurately predict the performance of VS on any parallel system. Experimental results show that VS is able to make an efficient use of the available computing resources, showing good scalability properties on all studied architectures.

Keywords: Machine learning, computational intelligence, optimization, parallel processing, scheduling, Virtual Savant

1. Introduction

Virtual Savant (VS) is a novel computational intelligence method to automatically generate fast problem solvers, recently presented by Pinel et al. (2018a). The method relies on a massively parallel pattern matching engine to learn, from one or more reference algorithms, how to generate accurate solutions for a given combinatorial optimization problem. VS has demonstrated high accuracy when solving the Knapsack Problem (Massobrio et al., 2018b) and the scheduling of independent tasks (Pinel et al., 2018a), both of which are $\mathcal{NP}$-hard combinatorial optimization problems. Scheduling

*Corresponding author
Email addresses: juan.detorre@uca.es (Juan Carlos de la Torre), renzom@fing.edu.uy (Renzo Massobrio), patricia.ruiz@uca.es (Patricia Ruiz), sergion@fing.edu.uy (Sergio Nesmachnow), bernabe.dorronsoro@uca.es (Bernabé Dorronsoro)

is a relevant problem when working over parallel/distributed heterogeneous computing platforms (Nesmachnow et al., 2010). An efficient scheduler is crucial to achieve appropriate load-balancing levels, thus reducing the time required to execute a set of concurrent tasks. The Heterogeneous Computing Scheduling Problem (HCSP) proposes assigning computing resources to a given set of tasks in order to optimize a certain efficiency metric (most commonly the *makespan*, i.e., the time from the moment that the execution of the first task begins until the completion of the last task). Our research group has applied computational intelligence methods, including heuristics and metaheuristics, to efficiently solve different variants of the problem (Nesmachnow et al., 2012; Nesmachnow, 2013; Iturriaga et al., 2014). In Pinel et al. (2018b), we propose an innovative method to solve the problem with VS, a computational intelligence/machine learning approach that learns from a given reference method and scales-up to solve different problem instances by applying a problem (data) decomposition, taking advantage of multiple computing resources.

Despite the massively parallel design of VS, its parallel performance has not been formally evaluated to date. A first step in this direction was presented in our preliminary work (Massobrio et al., 2018a), where a parallel implementation of VS was proposed and implemented over a multicore platform. The parallel implementation showed excellent scalability properties, both when increasing the number of parallel resources and the problem size (since larger problems require more computations).

The research reported in this article extends the one in the previous work (Massobrio et al., 2018a) in different directions. Firstly, we propose and evaluate a novel parallel implementation of VS that works in both distributed- and shared-memory architectures, using Message Passing Interface (MPI) MPICH library and OpenMP, respectively. We compare it against the original sequential VS implementation in terms of their parallel performance and the quality of results they are able to achieve. Secondly, different parallel architectures are considered, including massively parallel CPUs (e.g., Intel®Xeon Phi™), last generation multi-core processors, and distributed HPC architectures. Thirdly, a specific simulator is proposed, which is able to accurately predict the execution time of VS in any parallel architecture, and is validated over the considered ones. Finally, larger problem instances than those considered in the previous work (Massobrio et al., 2018a) are studied. These instances are more suitable for realistic scenarios and have significantly larger computing demands. All these advances are specific contributions of the research reported in this article. Furthermore, numerical results from the experimental evaluation demonstrate that VS is able to take advantage of the availability of resources to compute accurate solutions to the problem, showing good scalability properties on all studied architectures.

The structure of the article is detailed next. Section 2 describes the main concepts about VS and Heterogeneous Computing Scheduling. After that, Section 3 details the parallel implementations we designed for VS. The proposed tool for predicting the execution times of the parallel VS implementation is presented in Section 4. The experimental analysis of the proposed VS implementation is reported in Section 5. Finally, Section 6 presents the conclusions and formulate the main lines of future work.

2. Virtual Savant and Heterogeneous Computing Scheduling

This section firstly introduces the Virtual Savant paradigm, then it reviews the main existing approaches to solve optimization problems with machine learning techniques, and finally, describes both the Heterogeneous Computing Scheduling Problem and the application of Virtual Savant to solve it.

2.1. Virtual Savant paradigm

As we already mentioned, Virtual Savant is a novel paradigm that aims to generate programs that solve a given optimization problem. The paradigm is inspired by the savant syndrome, a rare mental condition that manifests in people with severe mental disabilities which excel at certain abilities, far in excess of what would be considered ordinary (Treffert, 2009). People with this condition, *savants*, are prodigious at skills related to memory, including rapid calculation and artistic abilities. In most cases, savants have neurodevelopment disorders such as autism, which severely impact on their social skills. Due to this fact, in addition to the rareness of the condition, researchers still struggle to understand the thought processes of savants. The main hypothesis suggests that savants are able to learn from low-level information in their memory by applying pattern recognition techniques (Treffert, 2006). Consequently, savants may be able to solve a problem with superlative efficiency without fully understanding the underlying principles involved in solving that given problem.

In analogy to the savant syndrome, VS proposes using machine learning techniques to identify patterns that allow solving complex optimization problems. VS aims to learn how to solve a given problem using supervised machine learning on a training set of instances previously solved using a reference algorithm. After the training phase, VS is able to solve new, unknown, problem instances. Moreover, VS can solve instances that are larger than the ones seen during training, without any further re-training.

VS consists of two clearly separated phases, namely, *prediction phase* and *improvement phase*. In the prediction phase, a trained machine learning classifier is used to predict a solution to a previously unseen problem instance. During the improvement phase, the predicted solution is further refined using search procedures and heuristics.

2.2. Application of machine learning to optimization problems

There are already some works in the literature that apply machine learning techniques to solve combinatorial optimization problems. Next, we briefly review them.

Recurrent neural networks (RNN) were initially applied to discrete combinatorial problems in Vinyals et al. (2015). In this work, Vinyals et al. introduced the Pointer Networks model *ptr-nets* to solve the planar convex hulls, the computing of Delaunay triangulations, and the planar Travelling Salesman Problem. *Ptr-nets* are trained using solved instances of varying sizes of those problems. Results showed that *ptr-nets* are very competitive even in problem instances with larger sizes than those used during the training phase.

Reinforcement learning has also been applied in optimization. Firstly, Li & Malik (2016) proposed an iterative algorithm that automatically generates optimizers for continuous optimization problems with no constrains. This algorithm uses the gradient of the fitness function to guide the last iterations of the iterative process. More recently, Hu

et al. (2017) extended the work presented in Vinyals et al. (2015) by applying deep reinforcement learning to *Ptr-nets* to solve a combinatorial optimization problem. The proposal was evaluated for predicting the sequence of items to pack in a bin when solving the three-dimensional bin packing problem. Heuristics were used for computing the orientation and location of items. This approach outperformed previous works in the literature for the studied problem instances by a 5% on average.

Machine learning techniques have been applied to other optimization problems in several domains. For instance, software optimization during compilation searches for the most suitable code transformations which do not alter the semantic of the code while improving a given metric, usually related to efficiency. In Ashouri et al. (2017), a two-level approach is proposed which uses machine learning to find sub-sequences for accelerating the best optimization for LLVM.

The Virtual Savant was initially presented by Pinel & Dorronsoro (2014), where it was evaluated on the task scheduling problem. It was later extended incorporating a genetic algorithm instead of a local search during the improvement phase, and also applying different reference algorithms during the training phase (Dorronsoro & Pinel, 2017; Pinel et al., 2018b). The parallel capabilities of the VS paradigm were evaluated on the resolution of the 0/1 Knapsack problem using a many-core computing environment in Massobrio et al. (2018b).

Because VS predicts each variable in the solution vector independently, it can efficiently use multiple computing resources simultaneously, and it can therefore be deployed on massively-parallel computing architectures. This massive parallel capability differentiates VS from all the surveyed works.

2.3. Heterogeneous Computing Scheduling Problem

Parallel computing systems are comprised of a coordinated set of processing elements (namely resources, processors, or machines), interconnected by a network, which are able to work cooperatively to solve complex problems. Heterogeneous computing systems are those whose resources have variable computational capabilities, most commonly the CPU processing power, but often other features such as RAM memory or external storage (Khokhar et al., 1993). Taking into account the diverse computing capabilities of heterogeneous computing systems, finding suitable task-to-machine assignments is a key issue to achieve an appropriate load-balancing and efficiency. Generally, the goal of the scheduling problem is to find an assignment which optimizes a given metric related to efficiency, economic profit, or quality of service offered to the users of the system.

The HCSP considers an heterogeneous computing system comprised of several resources (*machines*) and a set of tasks with variable computational requirements to be executed in the system. A task is defined as an atomic workload unit, i.e., it must be executed without interruptions and cannot be split into smaller chunks (the scheduling problem follows a non-preemptive model). The execution time of any individual task varies from one machine to another and is assumed to be known beforehand, following a static scheduling approach.

Several variants of the HCSP have been proposed, by considering task-to-machine assignments that optimize different quality metrics (Nesmachnow et al., 2010; Nesmachnow, 2013; Iturriaga et al., 2014). In this article, the goal is to optimize the *makespan*, a well-known optimization criterion related to the productivity of a computing system.

Makespan is defined as the period of time between the start of the first task (in the set of tasks to be executed) and the completion of the last task. This problem model is suitable for applications following the bag-of-tasks approach for independent executions on nowadays cluster, grid, and cloud computing systems.

The mathematical formulation for the HCSP considers:

- A set of tasks $T = \{t_1, \ldots, t_n\}$ to be scheduled and executed on the system.
- A set of heterogeneous machines $M = \{m_1, \ldots, m_m\}$.
- A function $ET : T \times M \to \mathcal{R}^+$ where $ET(t_i, m_j)$ indicates the execution time of task t_i on machine m_j.

The HCSP proposes finding an assignment function $f : T \to M$ that minimizes the makespan, defined by Eq. 1.

$$\text{makespan} = \max_{m_j \in M} \sum_{t_i \in T, f(t_i) = m_j} ET(t_i, m_j) \tag{1}$$

The case studied in this article considers independent tasks, where the execution of any given task does not depend on the completion of any other. Even though more general formulations of the HCSP exist—e.g., accounting for task dependencies and other objectives (Dorronsoro et al., 2014)—the independent task model is highly relevant, specially in distributed computing environments. Independent-task applications are commonly used in scientific research and shared computing infrastructures, where different users submit tasks to be executed. Consequently, the relevance of the HCSP variant studied in this work is given by its direct application to realistic distributed computing environments.

2.4. Virtual Savant for the HCSP

The VS implementation for the HCSP uses MinMin as a reference algorithm. MinMin is one of the most popular methods for approximately solving the HCSP and consists of a two-phase heuristic that greedily picks the task that can be completed the soonest (Luo et al., 2007). The algorithm focuses on balancing the load among all machines, often leading to better makespan values than other heuristics (Braun et al., 2001).

Figure 1 outlines the training process used in VS for the HCSP. The VS implementation for the HCSP uses Support Vector Machines (SVMs) for the prediction phase, which are trained using MinMin as the reference algorithm. Each task in a HCSP instance is considered individually during the training phase of VS. Thus, training vectors hold the execution time of a given task on each machine. The classification label corresponds to the identifier of the machine assigned to that task by the MinMin heuristic. Consequently, a single MinMin solution to a given HCSP instance provides as many observations as the number of tasks in the problem instance. This fact allows drastically reducing the number of solved problem instances required in the training process. The LIBSVM implementation with a Radial Basis Function (RBF) kernel was used (Chang & Lin, 2011). In order to improve training times on manycore architectures, a specific LIBSVM fork was developed (Massobrio et al., 2017).

Once the training of the SVM is completed, VS is able to solve new, previously unknown, and even larger instances than those seen during training. The complete VS

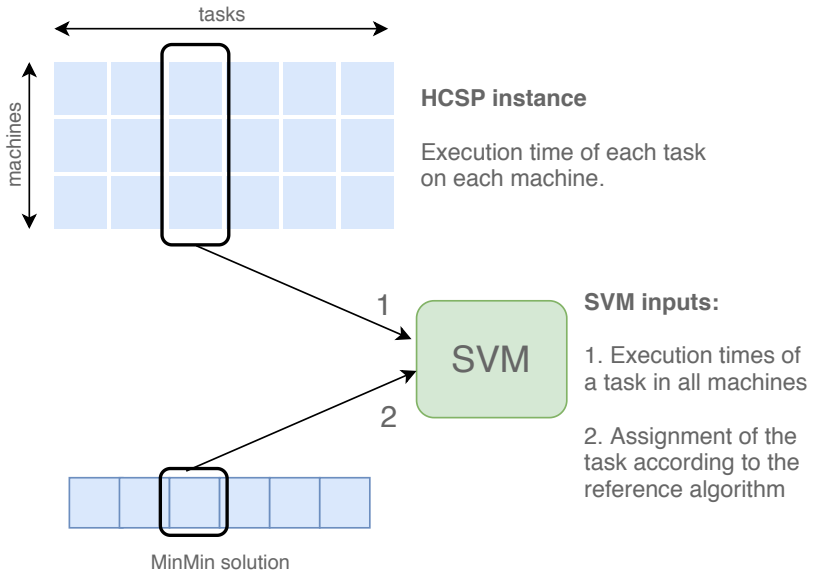


Figure 1: Training scheme of VS for the HCSP

execution workflow is outlined in Figure 2 and consists of two distinct phases: *prediction* and *improvement*.

In the prediction phase, VS receives as input an HCSP problem instance to solve, which consists of a matrix with the execution time of each task on each machine. Given that the training phase is done considering each task separately, the prediction (i.e., the task-to-machine assignment) can be done independently for each task. Thus, multiple copies of the same SVM can be spawned, forming a pool. The problem instance is split among this pool and predictions are made for each variable in a parallel fashion. This design, where predictions are made independently for each task, provides VS with a high degree of parallelism. At the finest grain, the VS design allows predicting the machine assigned to each task in the problem instance using a different copy of the same SVM classifier. The output of each SVM corresponds to the probability of assigning the task given as input to each of the available machines. The predictions of all SVMs in the pool are gathered to form a matrix that holds, for each task, its assignment probability on each machine.

During the improvement phase, the matrix with assignment probabilities computed in the prediction phase is used to generate a set of candidate solutions to the problem instance. The set of candidate solutions is randomly generated considering the probability of assigning each task to each machine. Then, a simple local search heuristic is applied over each generated solution. The local search operator used in this work consists in iteratively moving a randomly-chosen task from the machine with the highest completion time (i.e., the most loaded one) to a machine among the N least loaded ones, such that the completion time after the task is moved is minimized. The improvement phase is also inherently parallel, since the improvement of a given generated solution is independent from the others. Thus, one candidate solution can be generated and improved

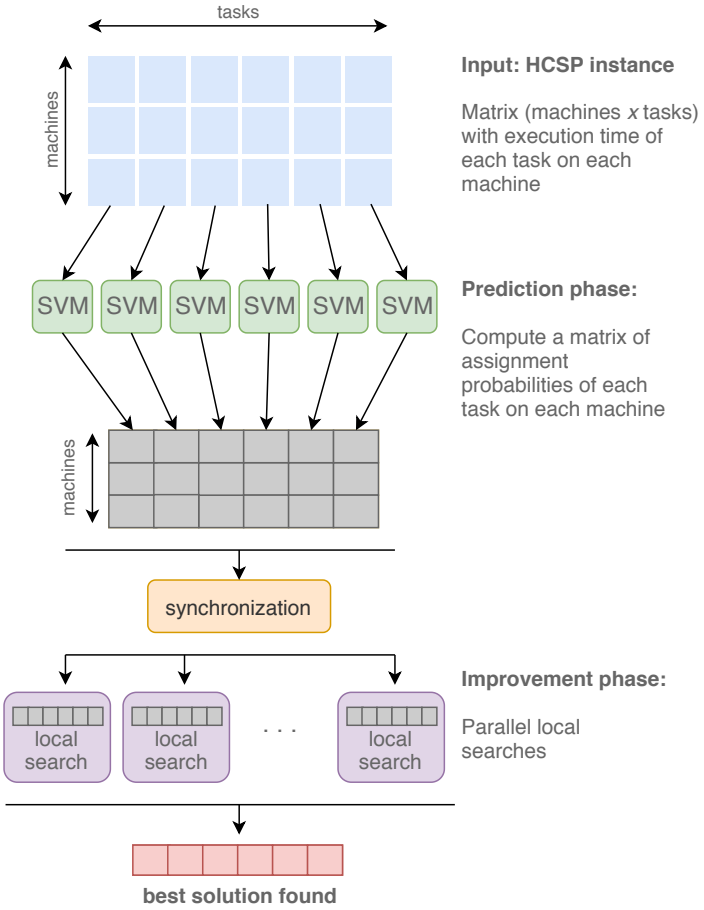


Figure 2: Virtual Savant for the HCSP

per computing resource available. After all local searches are completed, the overall best solution found is returned.

3. Parallel implementation of Virtual Savant

Despite its massively parallel design, parallel implementations of VS have not been fully studied to date. In this section, the first parallel implementation of VS is presented, which works for both distributed- and shared-memory architectures. It is based on the Message Passing Interface (MPI) MPICH library and OpenMP. The fact that the parallel components in VS (i.e., the prediction and the improvement phases) have no data dependencies among their processes allows for efficient implementations of the model in both distributed- and shared-memory platforms.

Our proposed VS implementation follows the same structure as the design presented in Fig. 2. In it, a master node launches the prediction phase in parallel, using as many

processes as the number of available computing nodes. Assuming a homogeneous architecture, the number of problem variables associated to every process (i.e., the number of predictions every process should compute) in order to balance the load of all computing resources is:

$$\left\lceil \frac{\text{number of variables}}{\text{number of cores}} \right\rceil .$$

The master node sends to every process the information required by the SVM, for every assigned variable. This information is taken from the problem instance, and in the particular case of the HCSP, corresponds to the execution time on all machines for every assigned variable. Therefore, the amount of data transmitted to every computing node is: assigned_variables $\times$ number_machines $\times$ 16 bits; in other words, the number of variables to be predicted using the SVM times the number of machines in the problem instance times the space needed to store an integer number: 16 bits. Additionally, the SVM implementation is parallel itself and makes use of all the available threads in each core of the computing node. In our implementation, if several variables are to be predicted in a given computing node, it is done in parallel, using as many SVM replicates as the number of threads the core can execute in parallel.

The master node waits until all processes are finished to build the probabilities matrix. Every process sends, for each assigned variable, the probability of assigning the corresponding task to each machine, so the amount of information is exactly the same as that received from the master at the beginning of the execution. After this matrix of probabilities is built (by just merging all collected results into one single matrix), the master node runs as many local searches as threads the system can execute in parallel. Each thread receives the matrix of probabilities from the master, builds a randomly-generated solution according to these probabilities, and iteratively applies the local search operator to this solution. After the threads finish executing the local search algorithm, they return the best solution found. Among all received solutions, the master node reports the best one as the result of VS. We would like to remark at this point that, when increasing the number of computing resources, VS increases the number of local searches executed, therefore the number of computations performed is multiplied.

4. Parallel Virtual Savant Performance Predictor

We present in this section our VS performance predictor. We call it the Parallel Savant Simulator, or PASSIM for short. It allows predicting the execution time of our parallel VS design in different parallel architectures with any number of resources. The simulator is evaluated by predicting execution times of VS when solving the HCSP.

The execution time of the parallel VS depends, of course, on the underlying computing infrastructure. This time is directly related to many different features, including: i) the specific hardware used to solve the problem, ii) the underlying communication network, and iii) the problem instance being solved. PASSIM uses as input information corresponding to all these aspects, and outputs the set of best computing solutions according to the specific hardware configurations available.

The simplest and most widely used statistical technique for predictive modeling is Linear Regression (LR), which provides an equation that models the relation between

a set of independent variables (explanatory variables or regressors) and the response (dependent variable). Equations 2 and 3 show the typical linear regression equations.

$$f(k) = \beta_1 \cdot \theta_1(k) + \beta_2 \cdot \theta_2(k) + ... + \beta_p \cdot \theta_p(k) + \epsilon(k), \quad (2)$$

$$f(k) = \sum_{i=1}^{p} \beta_i \cdot \theta_i(k) + \epsilon(k). \quad (3)$$

In our problem, the set of independent variables, $\theta_i(k)$, is represented by the different features considered. The response obtained, $f(k)$, is the expected performance. The different β_i are the coefficients and the ϵ models the error term, i.e., the part of $f(k)$ the regression model is unable to explain.

At this point, not only choosing the most influential variables, but also the most appropriate number of them that shall be included in the model is a complex task. The reason is the direct impact this choice has on the results obtained. The *least absolute shrinkage and selection operator*, also known as *Lasso*, is a regression analysis that selects the most appropriate variables and regularization with the goal of maximizing prediction accuracy. It was firstly introduced by Tibshirani (1996). Lasso performs the standard least-squares minimisation technique with the addition of a penalty on the parameter vector as shown in Equation 4.

$$\min_{\theta} \frac{1}{2} \|(f(k) - \beta \cdot \theta)\|_2^2 + \lambda \|\theta\|_1 \quad , \quad (4)$$

where $\lambda \in [\lambda_{min}, \lambda_{max}]$ behaves as a regularisation parameter. λ controls the trade-off between approximation error and sparseness. Lasso potentially sets $\beta_i = 0$ for some i. Consequently, Lasso behaves as a structure selection instrument.

The output of PASSIM is not a single solution but a set of them. The main reason is that, in order to give a single solution, PASSIM should also do scheduling in case the number of resources differ from the number of tasks. However, we consider this should be decided post-hoc. Therefore, PASSIM gives as output a table with a set of optimal solutions for the available resources in terms of the task scheduling. An overview of the operation mode of PASSIM is shown in Figure 3.

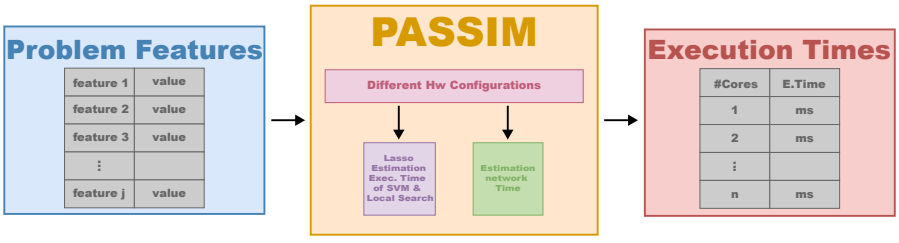


Figure 3: Operation mode of the Parallel Savant Simulator, PASSIM

5. Experimentation

In this section, we detail the experiments performed in this work and we also present the main results obtained.

5.1. Methodology

In the experimental evaluation, two different problem instances of the HCSP, introduced in Section 2.3, are considered: i) instance *512×16*, consisting of 512 tasks and 16 machines, ii) instance *4096×16*, consisting of 4096 tasks and 16 machines. Both problem instances were generated according to the methodology described by Braun et al. (2001). Tasks and machines were considered as highly heterogeneous and machines were considered consistent (i.e., a machine cannot be faster than another for a given task and slower for another task). The largest instances used during the training phase of VS were of size 512×16. Thus, instance *4096×16* allows studying the scalability of VS in the problem dimension, when solving a larger instance than those used for the training phase. The effectiveness of VS was studied over a larger benchmark of 180 HCSP instances with varying size and machine heterogeneity in our previous work (Massobrio et al., 2018b).

As explained in Section 2.4, the local search of VS for the HCSP, randomly gets a task from the machine with the highest completion time and includes it in a machine among the N least loaded ones. This value N was set to 50% of the total number of machines.

Four different hardware architectures were considered during the experimental evaluation:

- *Desktop*, which corresponds to a regular desktop PC with an Intel i7 4792MQ quad-core processor with 3.20GHz of peak frequency and 8 GB of RAM.
- *Xeon*, a standalone server comprised of an Intel Xeon E5-2620 CPU with 6 cores and a maximum peak frequency of 2.6 GHz.
- *Cluster*, which consists of a supercomputing sever composed of 3 rack-server type-C (c7000) from HP. Each of them contains 16 nodes bl460c, with 2 Intel Xeon E5 2670 with 2.60 GHz processors per node. Each of these processors have 8 cores, thus it has 48 x 2 x 8 = 768 cores. Nodes are equipped with 128 GB of RAM, and they are connected through a 10 Gigabit Ethernet.
- *Phi*, comprised of an Intel®Xeon Phi™ 7250 processor, with 1.60 GHz maximum frequency speed and 68 cores, and 48 GB of RAM.

Regarding PASSIM, the data set for creating the regression model was obtained by executing 100 times both problem instances with all the available resources. Then, for the validation of the proposed approach, we use the well-known k-fold cross-validation statistical method. In machine learning, cross-validation is used to estimate the accuracy of a model on unseen data, i.e., it estimates how accurately the predictive model is considering a limited data sample (Geisser, 1993). In predictive modeling, there are two different data sets: the training and the validation sets. The k-fold cross-validation technique randomly shuffles the samples and divides them into k sets. From those, it uses k-1 sets for training and the remaining one for validation. This procedure is repeated k times until each subset is used as the validation set once. The value of k highly influences the resulting accuracy of the model. Literature suggests that a commonly used value for k, with low bias and modest variance, is 10 (McLachlan et al., 2004).

As mentioned in Section 4, PASSIM needs as input the set of variables that explains the behavior of the model for predicting the execution times of the parallel VS. In our

work, these variables are represented by features of i) the hardware architecture, ii) the communication network and iii) the problem instance. Table 1 shows and briefly describes all the features we have identified. Out of these features, hardware architecture and problem instance are the inputs given to Lasso.

Table 1: Set of features of the problem given to PASSIM as entry for performance prediction.

info	variable	type	description
Hardware	CPU_Mode	Int	Bits
	N_Sockets	Int	# of sockets in motherboard
	N_Cores	Int	# of cores per socket
	N_Threads	Int	# of threads per core
	N_Numa	Int	# of NUMA nodes
	MHz_Cpu_Max	Int	Max. Frequency of processor (MHz)
	MHz_Cpu_MIN	Int	Min. Frequency of processor (MHz)
	N_BogoMIPS	Int	# of BogoMIPS
	KB_Cache_L1d	Int	L1 cache capacity (KB)
	KB_Cache_L2d	Int	L2 cache capacity (KB)
	KB_Cache_L3d	Int	L3 cache capacity (KB)
	KB_Ram	Int	RAM memory capacity (KB)
Network	Mb_Base_Tx	Int	Network speed (Mbps)
	N_MTU	Int	MTU size (Bytes)
	N_Steps	Int	# of network information exchanges
Problem	N_Instances	Int	# of Instances
	N_Threads_used/SVM	Int	# of Threads used per SVM

5.2. Numerical results

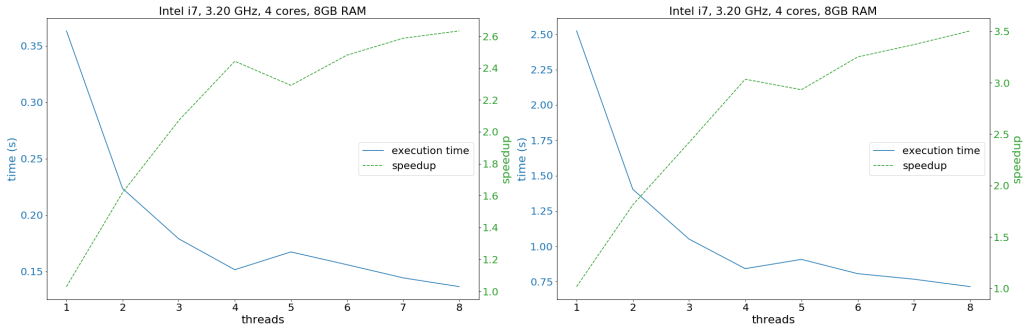
The performance of VS was studied on each of the architectures considered for the two studied HCSP instances.

Figure 4 shows the execution times of VS on the Desktop architecture when varying the number of threads. For each of the studied instances, the average execution time of 10 independent runs is reported. Additionally, the obtained speedup when using each number of threads is shown. Speedup is defined as the ratio between the execution time of the sequential implementation and the execution time of its equivalent parallel implementation. It can be seen how execution time is reduced when increasing the number of threads, due to the parallel capabilities of the VS model. For a correct interpretation of the provided results, it is worth noting that the computational load of the improvement phase does not remain constant, since the number of local searches increases with the number of threads used, as mentioned in Section 3. The ideal case for the improvement phase is that the run time remains almost constant when increasing the number of threads. This would mean that there is no overhead or bottlenecks when increasing the number of threads, a desirable condition for parallel algorithms. Consequently, the only improvement in execution times due to the use of multiple computing resources can be achieved during the prediction phase of VS.

The maximum speedup achieved in Desktop architecture is 2.63 for instance 512×16 and 3.50 for instance 4096×16. These are highly successful results if we take into account that the architecture implements 4 physical cores. In both cases, we can appreciate a desirable scalability of the algorithm when using up to 4 threads, given that we roughly get a 2.5x and 3x speedup when using 4 threads for the small and large instances, respectively. Then, this upward trend is mitigated when more than 4 threads are used. The explanation is that in these cases, the processor is using hyperthreading, because

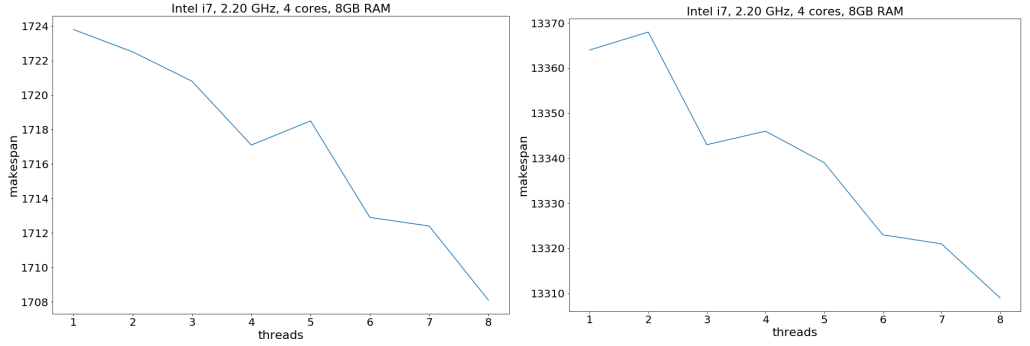
the number of physical cores is 4. In any case, it can be seen how hyperthreading can help in further reducing the execution times.

We present in Figure 5 the average makespan values achieved in the experiment. We can appreciate a trend where the average makespan values obtained decrease when increasing the number of threads used. Again, the reason is that the higher the number of threads used, the more local searches are executed and, therefore, the higher the chance of computing better results. Therefore, the increase in the quality of the solutions found comes at no cost in the execution time, thanks to the parallel capabilities of the VS paradigm.



(a) 512×16 (b) 4096×16

Figure 4: Average execution times with varying number of threads on Desktop architecture

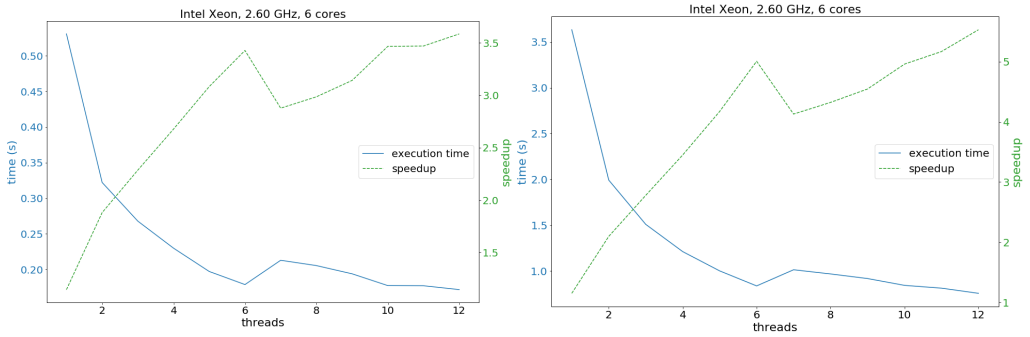


(a) 512×16 (b) 4096×16

Figure 5: Makespan with varying number of threads on Desktop architecture

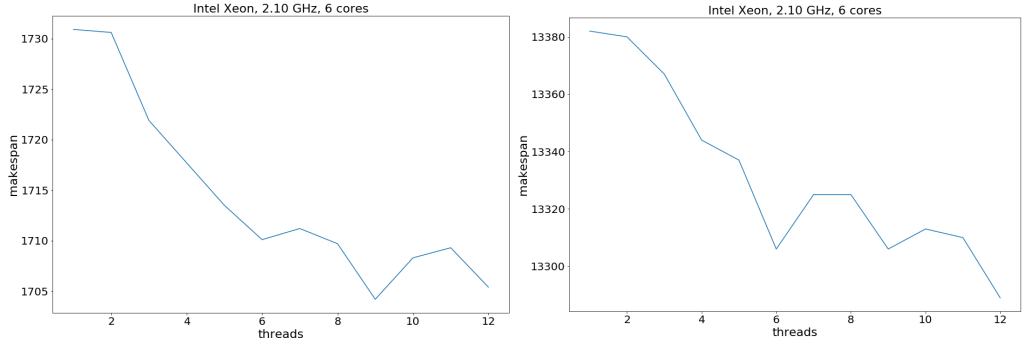
Similarly, Figures 6 and 7 show the average execution times and speedup values, and the makespan values, respectively, of VS when executing on the Xeon architecture using different number of threads. The studies are reported for the two problem instances considered.

The experiments performed in the Xeon architecture allowed studying the parallel capabilities of VS in a standalone workstation in addition to a regular desktop PC. Speedup values of up to 3.59 and 5.53 were achieved for the 512×16 and 4096×16 instances, re-



(a) 512×16 (b) 4096×16

Figure 6: Average execution times with varying number of threads on Xeon architecture



(a) 512×16 (b) 4096×16

Figure 7: Makespan with varying number of threads on Xeon architecture

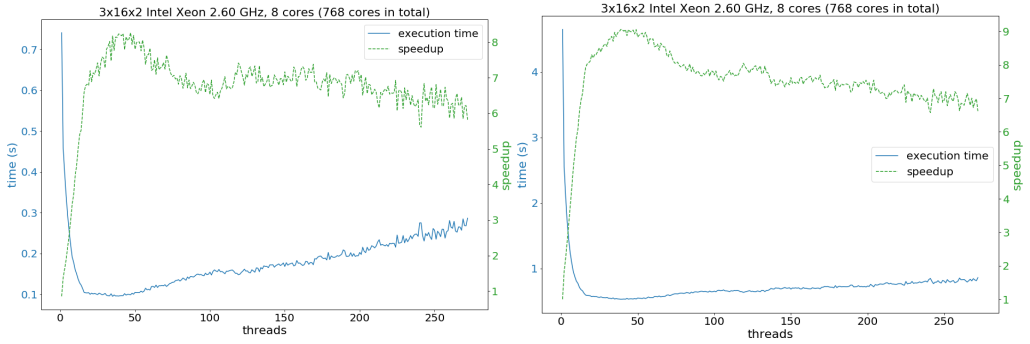
spectively. As it happened for the Desktop architecture, results show the good parallel capabilities of the VS paradigm, given that the processor has 6 physical cores. As in the Desktop case, hyperthreading can help in some cases to reduce computation times.. Significant improvements in the quality of the solutions found (i.e., better makespan values) can be computed when increasing the number of threads, without negatively impacting on the overall execution time.

Then, the performance of VS was evaluated on a distributed computing infrastructure. Figure 8 shows the execution times and speedup values of VS executing on the Cluster architecture when varying the number of threads used. Similarly, Figure 9 reports the makespan achieved in this experiment. As in the previous cases, the makespan value decreases (i.e., solutions are better) when increasing the number of threads used.

Results show that VS is also able to make an efficient use of a distributed computing infrastructure. When comparing the execution times of VS in the Xeon and the Cluster infrastructure, it can be seen that no significant penalties in execution times exist when moving from a shared-memory computing infrastructure to a distributed environment. A slight time execution penalty can be observed when comparing the case when only

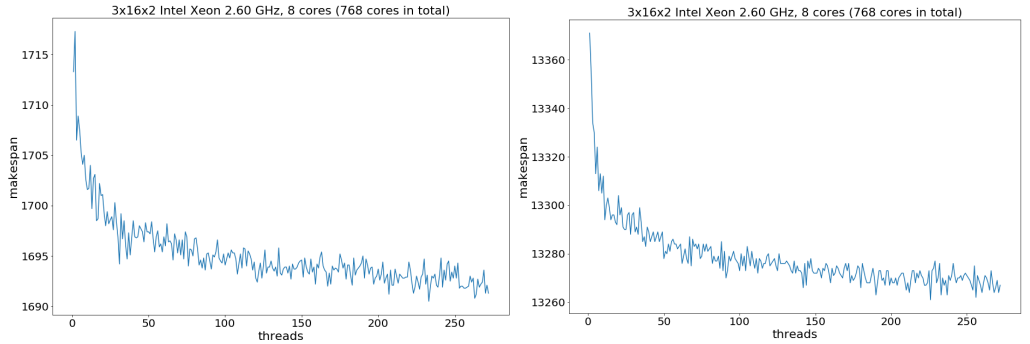
one server is used (threads values 1 to 8 in the plot) to that of using 23 servers (the rightmost results in the plots). At this point, we would like to note that the method finds the solution in less than 0.4 seconds using 23 servers, in the case of the small problem. We foresee that a higher computational load in every node is required to achieve better speedup values, motivated by the slight performance improvement of VS that is appreciated when increasing the number of threads for the large instance. In this sense, we present some preliminary results for very large problem instances later in Section 5.4, and we plan a deeper study on the efficiency of VS in this architecture as future work.

Regarding the quality of solutions achieved, a similar trend to the other studied architectures can be observed: better makespan values are achieved when increasing the number of computing resources. The presence of spikes in the plots are due to the stochastic nature of the local search. However, the desired downward trend appears in all studied architectures.



(a) 512×16 (b) 4096×16

Figure 8: Average execution times with varying number of threads on Cluster architecture

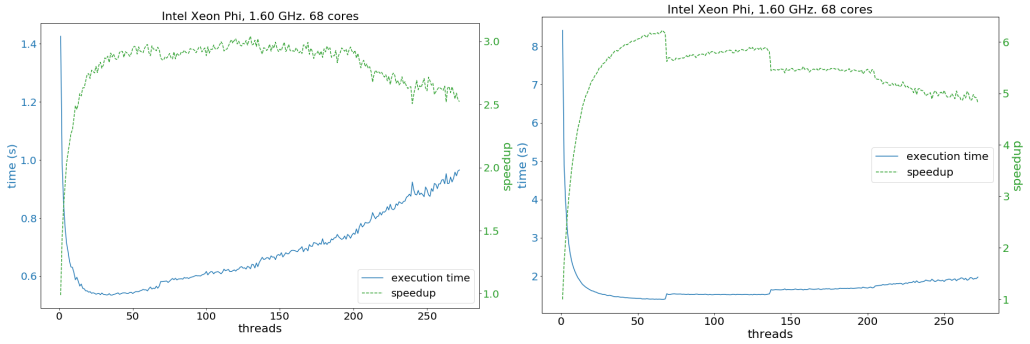


(a) 512×16 (b) 4096×16

Figure 9: Makespan with varying number of threads on Cluster architecture

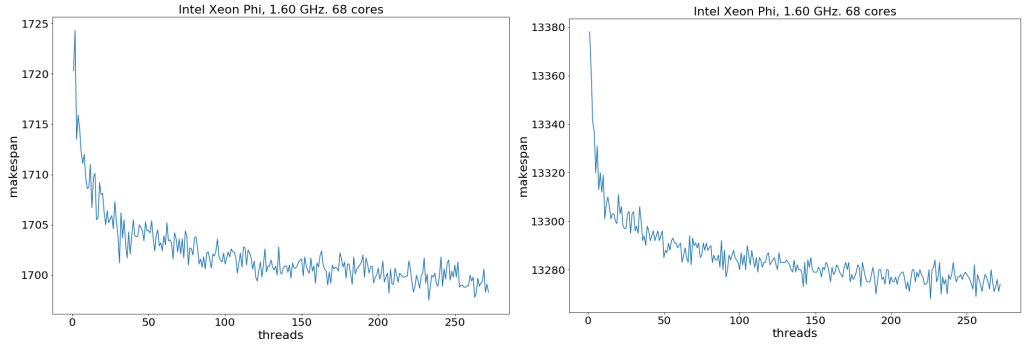
Lastly, the performance of VS was evaluated on the massively parallel Phi architecture. Mean execution times with varying number of threads are presented in Figure 10,

alongside speedup values, while average makespan values are reported in Figure 11. It can be noticed that execution times in Xphi architecture are the longest ones in our study, the reason is that the computation units in Xphi system are highly limited in resources and speed compared to the other ones. However, results show that VS is able to take advantage of the massive parallelism capabilities Xphi offers. Execution times are reduced when increasing the number of threads up to the number of physical cores available and then sligthly increase when more threads than the number of available physical cores are spawned.



(a) 512×16 (b) 4096×16

Figure 10: Average execution times with varying number of threads on Phi architecture



(a) 512×16 (b) 4096×16

Figure 11: Makespan with varying number of threads on Phi architecture

In terms of the quality of the computed solutions, results show that, once again, the average makespan is reduced when increasing the number of threads spawn. Thus, VS is able to take advantage of the availability of more computing resources to further refine the predicted solutions during the improvement phase.

5.3. VS performance prediction

In order to accurately predict the performance of VS on any architecture, we have proposed PASSIM (described in Section 4). It requires as an input a set of problem

features, as well as some HW and network characteristics, for accurately predicting its performance. Most of these variables (all but the network related ones) are given to Lasso for modeling the problem. The obtained output is shown in Figure 12. It presents the Minimum Squared Error (MSE) obtained after using the 10-fold cross-validation approach in terms of the values of λ. We can see that for values lower than $\lambda \simeq 70$, the MSE remains constant and, from that point on, it increases. The explanation variables identified by Lasso for best modelling the problem are four: N_Threads, KB_Cache_L3d, N_Instances and N_Threads_used/SVM, the remaining ones are set to zero or close to zero values. These results correspond to PASSIM trained on the four hardware architectures considered. In order to predict execution times on new architectures, PASSIM should be re-trained with the specifics characteristics of the new architecture.

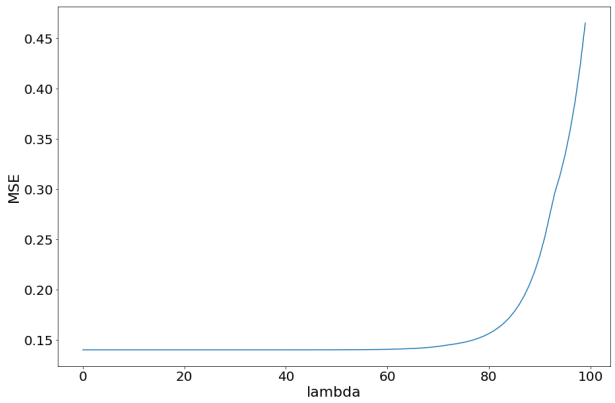


Figure 12: Evolution of the MSE in terms of the λ values of Lasso when using the 10-fold cross-validation approach.

Moreover, we have also tested the accuracy of PASSIM when predicting the performance of VS on the cluster architecture, as it is the only one making use of network communications. We have tested the accuracy of the prediction with the small HCSP instance, and using the lowest value of λ. The results are shown in Figure 13. As we can see, the prediction of the performance of the parallel VS is highly accurate even when predicting the behaviour of up to 23 nodes, the obtained value of the MSE is as low as 0.04. With the same instance and for the smaller lambda, the MSE for Xphi, Desktop and Xeon is 0.08, 0.17, and 0.16, respectively. In Figure 14, we show the mean square error of the predicted time over the measured execution time for all possible values of λ. As expected, Figures 14 and 12 have a similar shape.

5.4. Experimentation on very large scale instances

To finish our experiments, we also evaluated the performance of parallel VS on very large instances, composed of 32768 and 65536 tasks. The obtained results were similar as those reported in previous sections for Desktop, Xeon and Phi architectures. As an example, we show in Figure 15 the results we obtained for the Desktop architecture. In this case, we have a considerably higher number of tasks to predict, and we also increased the number of iterations of the local search to 1 million, given that the complexity of the problem importantly raises with the instance size.

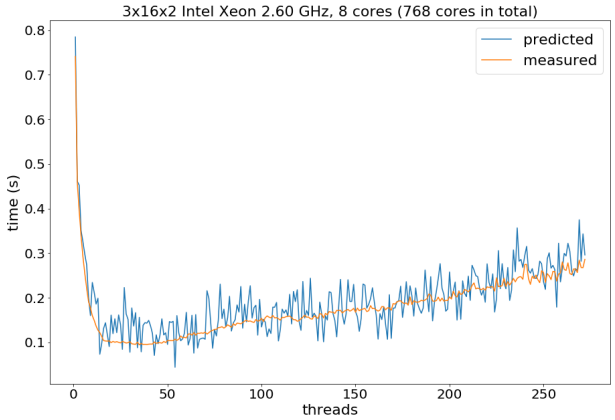


Figure 13: Measured execution time of the HCSP 512×16 instance versus the estimated execution time considering λ has the lowest value.

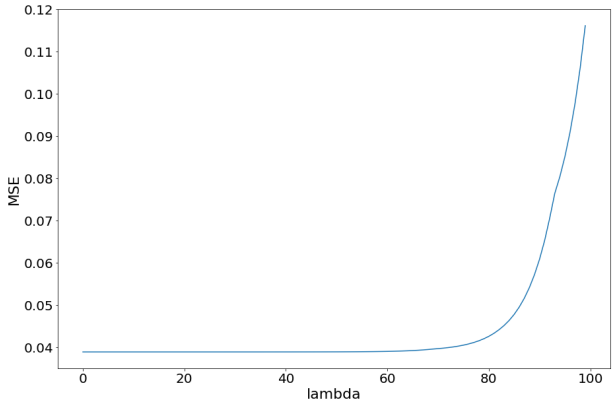
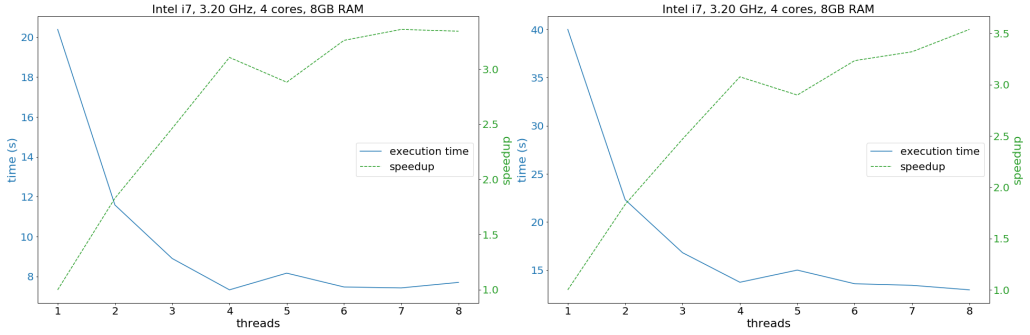


Figure 14: MSE of the predicted execution time for the HCSP 512×16 instance in terms of all possible values of λ.

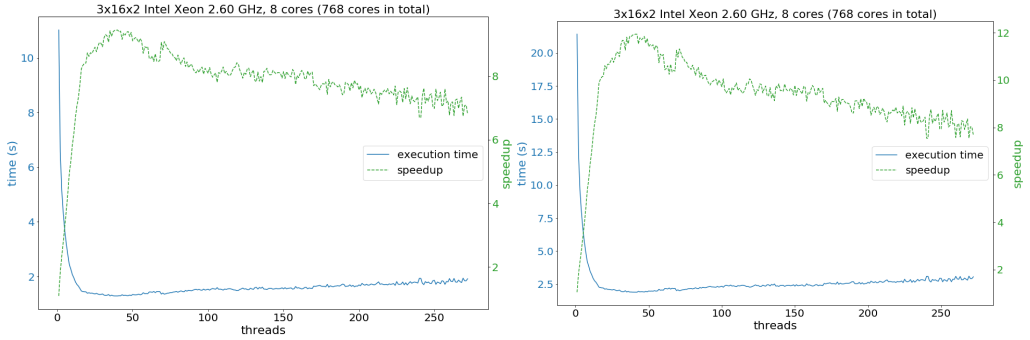
It can be seen in the figure how for the two very large instances VS exhibits a similar behavior as in the case of the smaller instances studied in Section 5.2. Speedup raises almost linearly when increasing the number of cores used, reaching up to 3.4x for 4 cores. Then, there is a slight performance loss when hyperthreading comes into use, but it increases again until the whole capacity of the processor is in use, when the best performance is obtained.

We repeated the same experiments in the Cluster architecture. The results are presented in Figure 16. Also in this platform, an improvement in the speedup of VS is appreciated when increasing the problem size. We can see how it can reach up to 12x speedup for the largest instance. The execution time is around 2.5 seconds, approximately seven times faster than the Desktop architecture. However, we think that, thanks to the efficient parallel design of VS, it would be possible to further improve speedup values considerably in the Cluster architecture by assigning a higher load of work to the nodes.



(a) 32768×16 (b) 65536×16

Figure 15: Average execution times with varying number of threads on Desktop architecture



(a) 32768×16 (b) 65536×16

Figure 16: Average execution times with varying number of threads on Cluster architecture

This is an interesting line for future work, and it can be done by implementing a more demanding optimization algorithm for the improvement phase, which might also lead to better results.

Finally, we would like to mention that the MinMin algorithm, which is a well-known and fast constructive heuristic from the state of the art, takes more than 2 minutes to find a solution to the largest studied instance (65536×16) in the Desktop architecture. It means that VS, which is trained from MinMin results, is roughly 11 times faster than MinMin on this architecture, and the solution it finds is only 3.85% worse than the one reported by MinMin.

6. Conclusions and Future Work

We present in this work the first parallel implementation and evaluation of Virtual Savant, a novel paradigm that uses machine learning to solve complex optimization problems. The parallel implementation relies only on two synchronization points and demands no information exchange among the parallel processes. We used MPICH Message Passing Interface and OpenMP libraries for the implementation, which allows efficiently dealing

with both shared- and distributed-memory architectures in a transparent way. The VS algorithm is suitable for both architectures, since the parallel processes are independent of each other.

Moreover, we present PASSIM, the Parallel Savant Simulator that predicts the execution time of VS in different parallel architectures. PASSIM uses Lasso to estimate the execution times of VS, taking into account problem instance characteristics, the underlying hardware available and the time needed for data exchange. Experimental results show that Lasso limits the number of variables needed for an accurate prediction to only four: the number of available threads, the Cache L3d in KB, the number of problem instances, and the number of threads assigned to each SVM. We have validated PASSIM in the cluster architecture as it makes use of the network communication with the smallest HCSP instance considered in this work. Experimental results show that PASSIM is highly accurate in its prediction with a MSE as low as 0.04 for not only the lowest value of λ but up to 0.7.

Experimental evaluation over four different computing infrastructures showed that the massively parallel design of VS allows taking advantage of available computing resources in order to compute accurate solutions for the HCSP problem. Increasing the number of parallel resources used allows reducing the execution time of the prediction phase and does not increase the overall computation time, despite the fact that the computational demand of VS increases with the number of threads. In addition, another interesting result is that the makespan value (measuring the quality of the obtained results) generally decreases (i.e., improves) when increasing the number of threads.

We complemented our experiments with an evaluation of the performance of VS on the different architectures for very large problem instances. We obtained similar curves of results as those obtained for the previosuly studied instances, where the speedup increases with the number of cores, with some efficiency loss when hyperthreading technology is used. However, speedup values were better for the large instances studied.

As future work, we plan to improve our VS implementation with a scheduler to perform an adequate load balance in heterogeneous parallel systems, and evaluating the accuracy of PASSIM in such cases. We also plan to perform a deep analysis on the scalability and performance of VS in the cluster architecture using a larger benchmark of bigger problem instances. Additionally, it would be interesting to explore the behaviour of VS and the prediction accuracy of PASSIM on other computing infrastructures such as GPU and MPP. Finally, we think it will be very interesting to enhance Lasso in order to make it more generic to be able to make different predictions in terms of the problem being solved by VS.

Acknowledgements

This work was partially funded by the University of Cadiz (contracts PR2018-056 and PR2018-062). B. Dorronsoro would like to acknowledge the Spanish Ministerio de Economía, Industria y Competitividad and ERDF for the support provided under contracts TIN2014-60844-R (the SAVANT project) and RTI2018-100754-B-I00 (iSUN project). The work of R. Massobrio and S. Nesmachnow is partly funded by PEDECIBA and ANII, Uruguay. R. Massobrio would like to thank Fundación Carolina, Spain. J.C. de la Torre would like to acknowledge the Spanish Ministerio de Ciencia, Innovación y Universidades for the support through FPU grant (FPU17/00563).

References

Ashouri, A., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J., Bignoli, A., Palermo, G., & Silvano, C. (2017). Micomp: Mitigating the compiler phase-ordering problem using optimization sub- sequences and machine learning. *ACM Trans. on Architecture and Code Optimization*, *vol. 14*.

Braun, T., Siegel, H., Beck, N., Blni, L., Maheswaran, M., Reuther, A., Robertson, J., Theys, M., Yao, B., Hensgen, D., & Freund, R. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, *61*, 810 – 837.

Chang, C.-C., & Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, *2*, 27:1–27.

Dorronsoro, B., Nesmachnow, S., Taheri, J., Zomaya, A., Talbi, E.-G., & Bouvry, P. (2014). A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. *Sustainable Computing: Informatics and Systems*, *4*, 252–261.

Dorronsoro, B., & Pinel, F. (2017). Combining machine learning and genetic algorithms to solve the independent tasks scheduling problem. In *The 3rd IEEE Int. Conf. on Cybernetics* (pp. 1–8).

Geisser, S. (1993). *Predictive Inference*. New York, NY: Chapman and Hall.

Hu, H., Zhang, X., Yan, X., Wang, L., & Xu, Y. (2017). Solving a new 3d bin packing problem with deep reinforcement learning method. *CoRR*, *abs/1708.05930*. `arXiv:1708.05930`.

Iturriaga, S., Nesmachnow, S., Luna, F., & Alba, E. (2014). A parallel local search in CPU/GPU for scheduling independent tasks on large heterogeneous computing systems. *The Journal of Supercomputing*, *71*, 648–672.

Khokhar, A., Prasanna, V., Shaaban, M., & Wang, C. (1993). Heterogeneous computing: Challenges and opportunities. *Computer*, *26*.

Li, K., & Malik, J. (2016). *Learning to optimize*. Technical Report CoRR, abs/1606.01885.

Luo, P., Lü, K., & Shi, Z. (2007). A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, *67*, 695–714.

Massobrio, R., Dorronsoro, B., & Nesmachnow, S. (2018a). Virtual savant for the heterogeneous computing scheduling problem. In *2018 International Conference on High Performance Computing Simulation (HPCS)* (pp. 821–827). doi:`10.1109/HPCS.2018.00133`.

Massobrio, R., Dorronsoro, B., & Nesmachnow, S. (2018b). Virtual savant for the heterogeneous computing scheduling problem. In *Int. Conf. on High Performance Computing & Simulation* (pp. 1–7).

Massobrio, R., Dorronsoro, B., Nesmachnow, S., & Palomo-Lozano, F. (2018b). Automatic program generation: Virtual savant for the knapsack problem. In *International Workshop on Optimization and Learning: Challenges and Applications* (pp. 1–2).

Massobrio, R., Nesmachnow, S., & Dorronsoro, B. (2017). Support Vector Machine Acceleration for Intel Xeon Phi Manycore Processors. In *Latin America High Performance Computing Conference.* (pp. 1–14).

McLachlan, G. J., Do, K.-A., & Ambroise, C. (2004). *Analyzing microarray gene expression data*. Wiley.

Nesmachnow, S. (2013). Parallel multiobjective evolutionary algorithms for batch scheduling in heterogeneous computing and grid systems. *Computational Optimization and Applications*, *55*, 515–544.

Nesmachnow, S., Cancela, H., & Alba, E. (2010). Heterogeneous computing scheduling with evolutionary algorithms. *Soft Computing*, *15*, 685–701.

Nesmachnow, S., Cancela, H., & Alba, E. (2012). A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. *Applied Soft Computing*, *12*, 626–639.

Pinel, F., & Dorronsoro, B. (2014). Savant: Automatic Generation of a Parallel Scheduling Heuristic for Map-Reduce. *International Journal of Hybrid Intelligent Systems*, *11*, 287–302.

Pinel, F., Dorronsoro, B., & Bouvry, P. (2018a). The virtual savant: Automatic generation of parallel solvers. *Information Sciences*, *432*, 411–430. doi:`https://doi.org/10.1016/j.ins.2017.12.021`.

Pinel, F., Dorronsoro, B., & Bouvry, P. (2018b). The virtual savant: Automatic generation of parallel solvers. *Information Sciences*, *432*, 411–430.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, *58*, 267–288.

Treffert, D. (2006). *Extraordinary People: Understanding Savant Syndrome*. iUniverse.

Treffert, D. (2009). The savant syndrome: an extraordinary condition. a synopsis: past, present, future. *Philosophical Transactions of the Royal Society B: Biological Sciences*, *364*, 1351–1357.

Vinyals, O., Fortunato, M., & Jaitly, N. (2015). Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 28* (pp. 2692–2700). Curran Associates, Inc.