Accelerating the Computation of Multi-Objectives Scheduling Solutions for Cloud Computing

Christophe Cérin, Tarek Menouer, Mustapha Lebbah
University of Paris 13, Sorbonne Paris Cité, LIPN/CNRS UMR 7030,
99 avenue J.B. Clément – 93430 Villetaneuse, France
Email: {Christophe.Cerin,Mustapha.Lebbah,Tarek.Menouer}@lipn.univ-paris13.fr

***Abstract*—This paper presents two practical Large Scale Multi-Objectives Scheduling (LSMOS) strategies, proposed for Cloud Computing environments. The goal is to address the problems of companies that manage a large cloud infrastructure with thousands of nodes, and would like to optimize the scheduling of several requests submitted online by users. In our context, requests submitted by users are configured according to multi-objectives criteria, as the number of used CPUs and the used memory size, to take an example. The novelty of our strategies is to select effectively, from a large set of nodes forming the Cloud Computing platform, a node that execute the user request such that this node has a good compromise among a large set of multi-objectives criteria. In this paper, first we show the limit, in terms of performance, of exact solutions. Second, we introduce approximate algorithms in order to deal with high dimensional problems in terms of nodes number and criteria number. The proposed two scheduling strategies are based on exact Kung multi-objectives decision algorithm and k-means clustering algorithm or LSH hashing (random projection based) algorithm. The experiments of our new strategies demonstrate the potential of our approach under different scenarios.**

***Keywords*-Scheduling and resource management; Performance Optimization; Machine Learning; Cloud Computing.**

I. Introduction

The past few years have been devoted to the progress of the innovations in hardware architectures such as the multi-core parallel machines or MIC (Many Integrated Cores). This allows researchers to propose new parallel applications that fully exploit the potential of such hardware. Currently, to execute these applications, Cloud Computing has been gained a widely used environment which provides to users a lot of flexible execution models with large capacity of computing resources.

In this context, requests scheduling and resources allocation present big challenging issues. Various resources management methods with various policies were published in past studies for Grid and Cloud Computing[2, 4, 9, 12, 19–21, 23, 24]. Some of these studies focus on scheduling independent requests, while others deal with the scheduling of dependent or synchronous requests. In this paper, we propose two new practical Large Scale Multi-Objective Scheduling (LSMOS) strategies that aims to be part of a Cloud scheduler. In our context, the requests submitted by users are configured according to multi-objectives criteria, as the number of used CPUs and the used memory size, to take an example. The goal of the proposed LSMOS strategies is to select, from a large set of nodes, the node that can execute a user request and which has a good compromise between multi-objectives. One issue is in accelerating the computation of the decision because the set of nodes is supposed to be large. The two LSMOS strategies proposed in this paper are based on the exact Kung multi-objectives decision algorithm and one algorithm among the following two algorithms: (i) k-means clustering algorithm; and (ii) LSH (Locality-sensitive hashing) hashing algorithm, reduced to a random projection.

Indeed, the overall motivation for such LSMOS strategies comes from the industrial Fonds Unique Interministériel (FUI-22) Wolphin[1] project, a collaborative industrial project oriented towards the themes of orchestration and optimization of the execution of requests, encapsulated in containers. Ultimately, the project, supervised by the Alterway company [2], aims to provide an efficient solution for hypervision and invoicing of container-oriented infrastructures. In fact, the Wolphin project search to improve the scheduling of requests encapsulated in containers for a large scale nodes number by taking into account multi-objectives configuration of requests in terms of the number of used CPUs and the used memory size.

This concrete objective is the use case we consider and it is part of a more challenging issue which is to consider high dimensional problems in terms of nodes number and criteria number. In other words, we tackle the following problem: *given a set of user requests, find an allocation of requests on nodes such that the user satisfaction is maximized and such that the number of used resources is minimizing, given a large number of nodes and criteria for the decisions on resources*.

The organization of the paper is as follows. Section II presents some related works. Section III is divided into subsection III-A which describe the exact Kung multi-objectives decision algorithm, subsection III-B which describe the k-means clustering algorithm and subsection III-C which describe the LSH and random projections algorithm. All of them are used as key building blocks of our global strategies. Section IV describes our LSMOS strategies based on exact Kung algorithm and K-means or LSH like algorithms. Section V introduces exhaustive experiments that allow the validation of our LSMOS strategies. Finally, a conclusion and some future

[1] https://www.alterway.fr/wolphin-2-0-laureat-du-fui-22/
[2] https://www.alterway.fr

works are given in section VI.

II. Related work

In the literature, many problems of resources allocation, or placement of user's requests refer to the same class of scheduling problems. They consist generally in associating a user's request to one or several computing cores. Most of these problems are *NP*-difficult [25]. In this general context, several scheduling systems and computing frameworks have been proposed and they target either High Performance Computing or Cloud contexts.

A. Scheduling and High Performance Computing

As concrete examples, we may cite NetSolve [2], Globus [10], Condor [23], Tetris [12], LoadLeveler [21].

To comment a few, NetSolve [2] is a system which uses a very simple scheduling algorithm. The principle is that each *NetSolve Agent* maintains a list of ready users requests, then it starts by executing a request each time it detects the existing of some free cores. A *NetSolve Agent* is the key of scheduling in the NetSolve system. It is the responsible of the load balancing between all machines reserved by the system. It also decides about the placement of the requests. Condor [23] is designed for high computing applications. It involves counting all free cores in the network to optimize the scheduling of users' requests by using these free cores. Condor provides a resources' description language, called *classified ads*. It allows some controls about the resources allocated to each request.

Regarding the scheduling algorithms used by the different systems proposed in the literature, the majority are based on heuristics. As a consequence, they are not able to provide the optimal scheduling solution, this being for two reasons:

- The scheduler does not have a complete knowledge of all users' submitted requests. So it is impossible to generate an optimal schedule of data that the system does not have.
- Even if the scheduler has sufficient knowledge of the submitted requests, the optimal resolution is difficult to obtain in terms of time.

Feitelson et al. [9] propose a scheduling system with two scheduling phases. The first phase consists in checking if there exists any requests that have been waiting for a long time, and, if their deadline is exceeded. In this case, the algorithm executes these requests which have the highest priority. The second phase consists in executing other requests saved in a global queue starting from the highest request's priority.

B. Scheduling and Cloud Computing

For High Performance Computing applications, the purpose is to schedule, for instance, Bag-of-Tasks scientific workflows on clouds [1] and/or to consider deadlines on the execution of tasks. Our aim is not to build a HPC cloud system but rather to run, eventually, HPC applications in the Cloud.

In [26], authors presents an experimental system that can exploit a variety of online Quality-of-Service aware adaptive task allocation schemes, with a specific focus on the characteristics and workload of applications. In our work, the allocation system makes no assumption on the workload and characteristics of applications.

In [27], authors consider that user jobs demand different amounts for different resources. They also notice that current Infrastructure as a Service (IaaS) clouds provision resources in terms of virtual machines (VMs) with homogeneous resource configurations. They propose a resource allocation approach, called Skewness-Avoidance Multi-Resource allocation (SAMR), to allocate resource according to diversified requirements on different types of resources. In our case, to take into account heterogeneous architectures and resources we assume that the allocation is done dynamically, based on the availability of resources at the requested time, i.e. the amount of allocated resources (CPU, RAM, Disk space…) is determined at runtime according to an abstract user specification.

C. Short overview of multi-objectives related problems

Combinatorial and discrete optimization problems such as routing, task allocation, and scheduling are important optimization applications in the real world. For traditionally methods, the time required to solve a combinatorial problem may increase exponentially in the worst case, thereby making them computationally too costly. Moreover, if the optimization involves multiple objectives, the process becomes more complex and difficult to solve [29].

Xing et al. [28] present a simulation model to solve a multi-objectives Flexible Job-Shop Scheduling Problem (FJSSP). The FJSSP is very important in the fields of combinatorial optimization and production management. Throughout the experiments, authors showed that multi-objectives evolutionary algorithms are very effective for solving the FJSSP.

Knowles et al. [14] propose a Pareto archived evolution strategy to solve multi-objectives optimization problem. The algorithm introduces a Pareto ranking-based selection method and couples it with a partition scheme in objective space. It uses two different archives to save non-dominated solutions.

D. Multi-dimensional search

Machine learning algorithms for large-scale multi-objectives optimization may also be considered as techniques to accelerate the search of solutions in multi-dimensional space. We assume in this paper that solving large-scale multi-objectives scheduling problems on large-scale systems remains challenging and that efficient and practical solutions for real Cloud systems not widely deployed. Such general techniques from the field of machine learning are surrogate meta-models, multi-armed bandits [16], landscape analysis [6] and online/offline automatic algorithm selection and configuration [3].

E. Positioning

To the best of our knowledge, all of studies proposed previously in the context of scheduling use a mono-based objective strategy to select a node which executes a selected request. This mono-objective strategy are called after applying a filter concept. The filter concept consist to filter all available nodes

to keep only nodes that can execute a selected request, this concept is used for example by Google Kubernetes[3], Docker SwarmKit[4] and many others scheduling systems. However, the proposed mono-objective scheduling strategy are not adapted for a large scale number of nodes. The novelty of this paper is to propose a new practical Large Scale Multi-Objectives Scheduling (LSMOS) strategies adapted for Cloud Computing composed from an infrastructure with thousands nodes.

III. Key algorithms used by our global strategies

This section is divided in three subsections: (i) subsection III-A describe the exact Kung multi-objectives decision algorithm; (ii) subsection III-B describe the k-means clustering algorithm; and (iii) subsection III-C describe the LSH hashing algorithm. All of theses three algorithms (exact Kung, k-means and LSH) are used as key building blocks of our global LSMOS strategies describe in section IV.

A. Exact Kung multi-objectives decision algorithm

One key piece of our multi-objectives scheduling strategy is based on the exact Kung algorithm [15]. It is among the best algorithms used in the multi-objectives decision context [8]. As presented in [8], Kung algorithm firstly sorts the population (nodes that can execute a container) in descending order according to the first criterion (as example, number of free CPUs or percentage of CPUs load...). Thereafter, the set of nodes are recursively halved as Top half (T) and Bottom half (B) sub set of nodes. As T is better in objectives in comparison to B in the first objective, so we check the B for domination with T. The solutions of B which are not dominated by solutions of T are merged with members of T to form merged set of nodes M. This means, in the case of a minimization function, that a solution x_1 is better than other solution x_2, if the value of x_1 is smaller than the value of x_2. The algorithm, called Front(P), can be summarized in two steps:

- Sort the nodes according the descending order of importance in the number of waiting CPUs criterion and rename the population as P of size N.
- Front(P): if $|P|$=1, return P as the output of Front(P). Otherwise, T= Front($P^1 - P^{|P/2|}$) and B= Front($P^{|P/2|+1} - P^P$). IF i^{th} non-dominated solution B is not dominated by any non-dominated solution of T, create a merged set M={T U i}. Finally, return M as output of Front(P).

In this context, we say that a solution x_1 dominates an other solution x_2 if two conditions are satisfied:

1) Solution x_1 is no worse than x_2 in all multi-objectives criteria;
2) Solution x_1 is strictly better than x_2 in at least one objective criterion.

If a solution x_1 dominates an other solution x_2, it is equivalent to say that the solution x_2 is dominated by the solution x_1. In our context, the goal of Kung algorithm is to select a set of nodes with a "good" compromise, for instance between the availability of CPUs cores and the free memory size. Then, our strategy returns the first node that can execute a container from the set of nodes returned by the Kung strategy.

[3]https://kubernetes.io/

[4]https://github.com/docker/swarmkit/

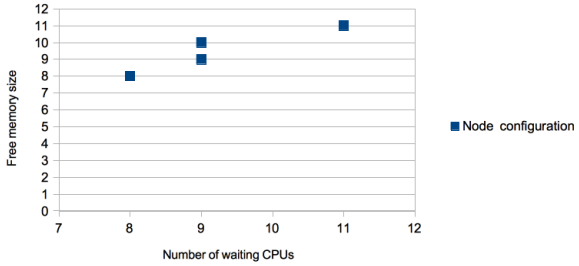

Fig. 1. Nodes configurations in terms of free CPUs and free memory

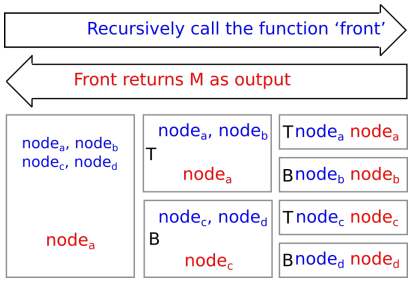

Fig. 2. Example of Kung strategy

1) Example of how the exact Kung algorithm works: Assume that at time t_0, we have a request R_x which need 6 CPUs and 6 GB of memory. We assume also that from all nodes of the infrastructure there are just four nodes ($node_a$, $node_b$, $node_c$ and $node_d$) which can execute R_x. The availability of each node in term of number of waiting CPUs and free memory size are presented in Figure1. As explained before, to select the first node that must execute the request R_x using Kung strategy, we start by ordering in descending order all nodes according to the value of the waiting CPUs criterion. Then, the set of nodes are recursively halved as Top (T) and Bottom (B) sub set of nodes as it is shown in the Figure 2 with red color. After applying the second step of the Kung algorithm as presented in the Figure 2 with blue color, the selected node is $node_a$.

B. K-means clustering algorithm

K-means clustering algorithm is a type of unsupervised learning algorithm [18]. The goal of this algorithm is to build clusters of data, with the number of clusters is represented by the variable K [18]. The algorithm works iteratively to assign each data point to one of K clusters. Data points are clustered based on feature similarity. The principle of the K-means algorithm is described as follows:

1) Randomly choose k objects as center objects;
2) Calculate the distance between each object and each center object. Then, assign each object to the cluster which has the nearest center object;

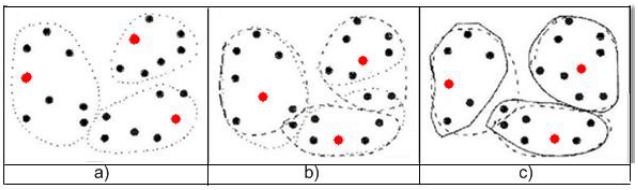

Fig. 3. Partitioning based on K-means [18]

3) Recalculate the new center object of each cluster;
4) Repeat step 2 and 3 until you achieve a cluster stability.

1) Example of how the K-means algorithm works: Let us consider a set of objects, as depicted in Figure 3 presented in [18]. In our context, each object represent a node with a configuration in term of CPUs and memory size. Let us also suppose that the number of clusters we want to have equal to 3 (k = 3).

We firstly arbitrarily choose three objects, each object represent an initial cluster center. In figure 3(a), each initial center is marked by a red color. Then, each object form the set of objects is assigned to a cluster based on the cluster center to which it is the nearest. Next, the cluster centers are updated. That means, each cluster recalculated its center based on the current objects in the cluster. Using the new cluster centers, the objects are redistributed to the clusters based on which cluster center is the nearest (Figure 3(b)). The process of iteratively reassigning objects to clusters to improve the partitioning is referred to as iterative relocation (Figure 3(c)). Eventually, no reassignment of the objects in any cluster occurs and so the process terminates. The resulting clusters are returned by the clustering process.

C. LSH (Locality Sensitive Hashing) algorithm

Locality-Sensitive Hashing (LSH) [7, 13] is one promising algorithmic complexity reduction approach relying on the idea that, if two points are close together in high-dimensional space, then they should remain close together after some projection to a lower-dimensional space (in our case to a straight line). LSH has many applications, among them near-duplicate detection, clustering, sketching, near-neighbor search ....

More formally, LSH is a probabilistic method based on a random scalar projection of multivariate data point $xL(x;w) = (Z^T x+U)/w$ where $Z \sim N(0,\mathbf{I}_d)$ is a standard d-variate normal random variable and $U \sim \text{Unif}(0,w)$ is a uniform random variable on $(0,w)$, $w > 0$. Due to the statistical properties of the normal distribution, close points in the original multivariate space tend to be projected in the same bucket and distant points tend to be projected into different buckets in the hash table, as verified by [22]. Larger values of w imply fewer buckets with more accurate preservation of characteristics of X_i, whereas smaller values of w imply more buckets with less accuracy.

Indeed, in our case, we can simplify the whole LSH process in computing a random projection matrix, then in multiplying it with the input, and, at last in using a modulo calculus to build the output matrix. Random projections have been introduced as a tool for dimensionality reduction and have been used to produce a number of results, both theoretical and applied. In this paper we reused the method of [11] because it deals with experiments and practical considerations. The full process, starting with input X, is thus:

1) Compute the random projection P as in [11];
2) Compute $E = X.P$ where X is the input set of n points in $\mathcal{R}^p$, in the form of a $n \times p$ matrix;
3) Sort the (small) set of data in E to form the output, to be passed to the front algorithm.

1) Example of how the LSH-like algorithm works: Assume that the input vector X is made of the following 4 points: $(1,5),(2,6),(3,7),(4,8)$. In our case each point represent a configuration of one node, for instance in terms of %CPUs and free memory size. To realize a projection of this 2-dimensional space in a 1-dimensional space, we build the projection matrix P according to the method in [11]. This gives matrix `[[ 1.939, 5.634 ]]`. The product $X.P$ is matrix: $[[30.109,37.681,45.254,52.827]]$ In this case, the elements in the matrix are sorted from left to right, and we have nothing more to do, meaning that the first point in X is the 'smallest' point,..., the last point in X is the greatest point for the front algorithm to come. Otherwise, we reorganize the input, thanks to the information that the index $\lceil X.P[i] \lceil$ may contain the rank of the i^{th} element in X in the auxiliary array T. For instance, $T[53] = 4, T[46] = 3, T[38] = 2, T[31] = 1$. Based on this information, it is trivial to produce the pseudo-output sorted sequence. Then, as the last step, we call any sorting algorithm such as Quicksort to produce the output required for the front algorithm. This is the tricky part of the work. It consists in reducing the number of candidates for the front through a projection of points in the same neighboring in the N-dimensional space, to the same location in the 1D space.

IV. Large Scale Multi-Objectives Scheduling Strategy

The novelty of this paper is in the introduction of two new LSMOS strategies. In following, we start by presenting in subsection IV-A the goal of the LSMOS strategies, then we present in subsection IV-B our first proposed algorithm based on a exact Kung algorithm (see subsection III-A) and K-means algorithm (see subsection III-B). Subsection IV-C present our second proposed algorithm based on a exact Kung algorithm (see subsection III-A) and LSH algorithm (see subsection III-C).

A. The goal of the LSMOS strategies

The goal of the LSMOS strategies is to found, from a large set of nodes forming a Cloud Computing platform, a node with a good compromise between multiple criteria to execute a request submitted by a user. In our context, and for the sake of simplicity of both the presentation and the experiments, we consider two criteria: (i) number of CPUs; and (ii) free memory size.

The LSMOS strategies are proposed in the context of cloud platform with thousands nodes. They are based on an approximations of the exact solutions for the problem of determining

Algorithm 1 LSMOS algorithm based on exact Kung and K-means algorithms

Input: n, number of nodes.
Input: D, set of all nodes.
Input: K, empiric value used by the K-means algorithm (for instance, choose $K = \log(n)$).
Output: x, elected node.

V = centers objects of all clusters returned by K-mean algorithm apply on input D.
for i $\Leftarrow$ 1 to K **do**
  E[i] = Add N objects around V[i].
end for
F = front given by exact Kung alg. on objects saved in E.
x = first node in F.

the best compromise i.e. the Kung algorithm (Pareto front). Indeed, we are guessing that approximate solutions are good enough for front-based algorithms because after the building of the front it remains to choose a point on the front. It is always questionable to select such point and to measure the impact of this choice on the 'optimal' solution. Optimality often depends on a user point of view: CPU criteria is better than memory criteria, for instance.

B. LSMOS strategy based on exact Kung and K-means algorithms

Algorithm 1 shows the principle of the proposed LSMOS strategy based on exact Kung and K-means algorithms. The principle consist to start by applying the K-means algorithm on all set of nodes to form K clusters with the same characteristics. K can be chosen to be equal to $\log(n)$, with n being the size of the problem i.e. the number of nodes in the infrastructure. This step reduces the state space of the problem. Then we choose a set of nodes in each cluster, for instance randomly or at a certain distance of the centroid or by applying the best known k-nearest neighbors classification algorithm [5]. The set of resulting nodes is called E. The objective is to have more information to improve the accuracy of the result. Finally apply exact Kung algorithm on the E set in order to obtained a front F. The node selected by this LSMOS strategy is the first node saved in F.

C. LSMOS strategy based on exact Kung and LSH algorithms

Algorithm 2 shows the overall steps of the proposed LSMOS strategy based on exact Kung and LSH-like algorithms. The principle consists to start by applying the LSH-like hashing algorithm to have a hashing vision of all nodes of the infrastructure . The goal is to have quickly a sort nodes. Then, apply exact Kung algorithm on the hashing set of nodes in order to obtained a front F. The node selected by this LSMOS strategy is the first node saved in F.

V. Experimental evaluation

In this section, we present some experimental evaluation to motivate our LSMOS strategies. The following experimentation are done on a MacBook laptop (1,1 GHz Intel Core m3 and 8GB 1867 MHz LPDDR3).

Algorithm 2 LSMOS algorithm based on exact Kung and LSH algorithms

Input: D, set of all nodes.
Output: x, elected node.

E = set of nodes returned by LSH like hashing algorithm on input D.
F = front returned by exact Kung algorithm on objects saved in E.
x = first node in F.

A. Limits of exact methods

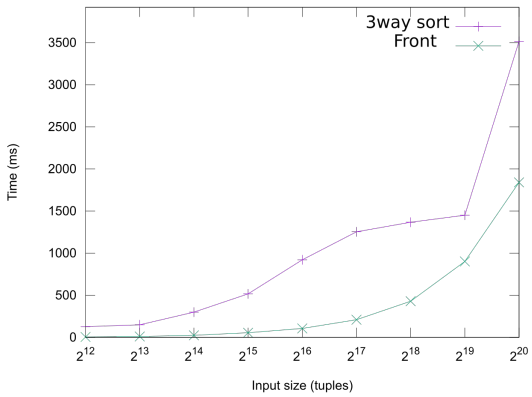

Fig. 4. Exact Kung algorithm (random values in $0 \cdots 100$)

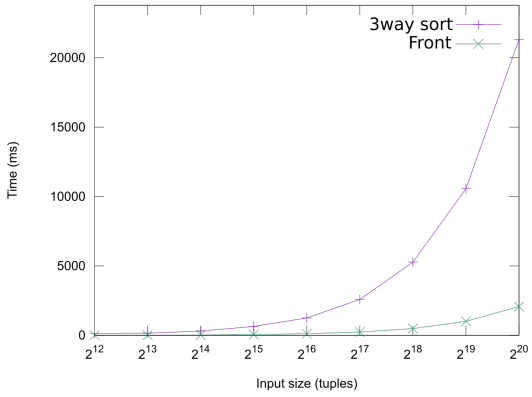

Fig. 5. Exact Kung algorithm (random values in $0 \cdots 10M$)

Figures 4 and 5 show the computing time obtained with the 3way sorting method and the front method for a set of experiences with different input sizes. Note that the combination between the 3way and front execution times forms the total execution time. From Figures 4 and 5, we notice that the exact Kung algorithm spends more time in the 3way sorting method than in the front computation (more information about

the front method is given in subsection III-A). On Figure 4, the configuration we have is such that the values (simulation of the load of nodes) are initialized randomly between 0 and 100%. For example, 2^{12} tuples (input size) means that we use an infrastructure with 4096 nodes. Each node, has a random value for its CPUs load set between 0 and 100%. On Figure 5 the configuration is different. The values that the algorithm handle are set randomly between 0 and 10M, making more variety in the input set comparing to the first experiment and stressing the sorting algorithm.

Results show in Figures 4 and 5 are obtained by using a parallel multi-threaded implementation with the Go language for both the 3way sorting and the front computation. They raise the objective of building a more efficient version for the computation of the node that will execute the submitted request, by acting on the sorting and/or the front computation. Our assumption is also in using an approximated approach rather than in computing an exact value for computing the elected node.

B. Performance with LSMOS strategies

1) Metrics of performance: For comparing the exact value of the front and the approximated value returned by our framework, we use the following metrics. The Jaccard index is a statistic used for comparing the similarity of sample sets. It is defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A|+|B|-|A \cap B|}.$$

If $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ are two vectors with all real $x_i, y_i \geq 0$ then their Jaccard similarity coefficient (also known then as Ruzicka similarity) is defined as

$$J(\mathbf{x},\mathbf{y}) = \frac{\sum_i \min(x_i, y_i)}{\sum_i \max(x_i, y_i)},$$

In our case, since x_i and y_j are 2D points, we compute a similarity coefficient on Euclidean distances to the origin.

2) Experimental settings: The experiments are conducted with MacOS High Sierra (version 10.13.6) laptops equipped with 8 and 16GB of memory, and 1.4GHz Intel Core i7 and 2.9GHz Intel Core i5. On both systems, we use Go version 1.10.3 for Darwin/AMD64. We now introduce some representative results with our Go codes.

3) Experiments: Table I corresponds to an experiment with Algorithm 1 (k-means + front computation) where the input data are taken randomly from 0 to 10M, hence a big variety for the values since the problem size is at most 1M. A single data is a dictionary with 5 keys and 5 values. Its size, which is about 10 times the size of a 32 bits integer, represents the size of a single data that we have to swap during the different steps of the algorithms. The settings are $k = 5$ and each cluster contains $50 * \log_2(n)$ points, n being the input size. As expected, the computational time is spent isn the K-means-like algorithm and not in the front computation since the input size of the front computation grows as the logarithm of the input size.

Table II is related to the Jaccard indexes. The experimental conditions are the same as for Table I. We notice high values for the indexes, here between 0.6925 and 0.9986 which is satisfactory for sparse data. These results confirm the potential of our method to predict the front.

TABLE I
K-MEANS-LIKE ALGORITHM IN ACTION ON 1.4GHZ INTEL I7.

Problem size	**K-means** (ms)	**Front** (ms)
4096	29.06	3.86
8192	52	3.93
16384	107.6	4.4
32768	280.6	4.33
65536	507.8	4.2
131072	1213.53	4.93
262144	2426.8	4.86
524288	5950.13	6.53
1048576	11207.4	5.13

TABLE II
SIMILARITY ANALYSIS (MEAN VALUES OVER 45 ITERATIONS)

Problem size	**Jaccard Index**
4096	0.9986
8192	0.8283
16384	0.8321
32768	0.8873
65536	0.6925
131072	0.96153
262144	0.8082
524288	0.7416
1048576	0.8665

Table III corresponds to an experiment with Algorithm 2 (random projection + front computation) where the input data are taken randomly from 0 to 10M, hence a big variety for the values since the problem size is at most 1M. Since there are few duplicates in the input, the results in Table III confirm our expectation about the computation time of the Random projection which is similar to the computation of the front, through the Kung algorithm. This example illustrates the fact that the random projection has only eliminated a few candidates i.e. the input problem size has not been reduced because of the variety in the input data. Table IV lists the Jacquard index under the previous experimental conditions. As expected, we notice a quasi-perfect detection of the front. Table V is related to the Jaccard Similarity Coefficients. As expected, under the experimental conditions, the measures are closed to 1.

TABLE III
LSH-LIKE ALGORITHM IN ACTION ON 1.4GHZ INTEL I7.

Problem size	**Projection** (ms)	**Front** (ms)
4096	296	9
8192	261	12.4
16384	341.8	26.3
32768	339.8	59
65536	531.8	115.6
131072	736	244.6
262144	1198.2	700.1
524288	2327.5	1680
1048576	4780.5	3514

Table VI corresponds to an experiment with Algorithm 2 (random projection + front computation) where the input data

TABLE IV
SIMILARITY ANALYSIS (MEAN VALUES OVER 45 ITERATIONS)

Problem size	Jaccard Index
4096	0.999
8192	0.999
16384	0.999
32768	0.999
65536	0.999
131072	0.999
262144	0.999
524288	0.999
1048576	0.999

TABLE V
SIMILARITY ANALYSIS (MEAN VALUES OVER 45 ITERATIONS)

Problem size	Jaccard Similarity Coeffs
4096	1
8192	1
16384	1
32768	1
65536	1
131072	1
262144	1
524288	1
1048576	0.9569

are taken randomly from 0 to 100, hence a configuration with many equal values since the problem size is at most 1M. The random projection will generate identical points in the 1D plan, reducing drastically the input size of the front problem. As expected, the acceleration for the computational problem is huge comparing to Table III, and the experiment illustrates the sensibility of the LSH-like algorithm regarding the shape of the input data.

Table VII lists the Jacquard index under the previous experimental conditions. We notice some differences that we can explain with the different Gaussian laws we use, since a Gaussian law is generated randomly at each iteration of our algorithm. Clearly, a study of the data should make it possible to better choose the Gaussian law in order to smooth the results. Learning the Gaussian law to use remains an open question. However, under these experimental conditions, since we have many identical points, we may assume that the solution reflects a good choice for the whole scheduling algorithm motivating our paper. Table VIII is related to the Jaccard Similarity Coefficients under the previous experimental conditions. As expected, the measures indicate that the approximate values are far to be equal to the exact values.

TABLE VI
LSH-LIKE ALGORITHM IN ACTION ON 2.9GHZ INTEL I5.

Problem size	Projection (ms)	Front (ms)
4096	0.066	0.866
8192	0.133	1.133
16384	0.5	0.266
32768	1	0.2
65536	2.2	0.4
131072	5.4	0.8
262144	10.8	1.066
524288	25	1.8
1048576	45.26	1.06

TABLE VII
SIMILARITY ANALYSIS (MEAN VALUES OVER 45 ITERATIONS)

Problem size	Jaccard Index
4096	0.881
8192	0.851
16384	0.414
32768	0.630
65536	0.989
131072	0.573
262144	0.740
524288	0.772
1048576	0.378

TABLE VIII
SIMILARITY ANALYSIS (MEAN VALUES OVER 45 ITERATIONS)

Problem size	Jaccard Similarity Coeffs
4096	0.0713
8192	0.0505
16384	0.0629
32768	0.1240
65536	0.0274
131072	0.0645
262144	0.0340
524288	0.0438
1048576	0.0384

VI. CONCLUSION

We presented, in this paper, the LSMOS strategies adapted for Cloud computing environment in order to compute an approximate value for the (Pareto) front. Our LSMOS strategies are based on exact Kung multi-objectives algorithm combined with K-means clustering algorithm or LSH-like hashing algorithm. The objective of this study is to execute a user request on the node which has a good compromise between multi-objectives criteria. We discussed the conditions to get a quick answer versus a precise answer. A balance between the Normal law to use (i.e. the shape of the data) and the expected precision should be identified, in practical cases, in order to satisfy the final user. We have shown the impact of such considerations on the obtained results. We have introduced previously in [17], the key ideas of a new scheduling and resources management system based on an economic model. To choose the node that must execute a request submitted online by a user, the system presented in [17], applies the Bin Packing strategy. As a perspective, we propose to add to the system proposed in [17] our LSMOS strategies and do experimental evaluation inside an infrastructure with large set of nodes. As a second perspective, we propose also to realize an implementation inside the Docker SwarmKit toolkit.

Choosing a Gaussian law, based on inferred knowledge on the data is also an issue that we plan to investigate. The method to privileged will be to learn on the data, in using machine learning algorithms. We should also investigate the problem of normalizing the data.

Acknowledgments. This work is funded by the French *Fonds Unique Ministériel (FUI)* Wolphin Project. The Go source files are available at https://github.com/lebbah/Go.

References

[1] J. Agarkhed and R. Ashalatha. Optimal workflow scheduling for scientific workflows in cloud computing. In *2016 International Conference on Emerging Technological Trends (ICETT)*, pages 1–6, Oct 2016.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Technical report, University of Tennessee, USA, 2002.

[3] L. P. Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle. Automatic configuration of GCC using irace. In *Artificial Evolution - 13th International Conference, Évolution Artificielle, EA 2017, Paris, France, October 25-27, 2017, Revised Selected Papers*, pages 202–216, 2017.

[4] S. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource management in legion. *Journal of Future Generation Computing Systems*, 1997.

[5] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967.

[6] F. Daolio, A. Liefooghe, S. Vérel, H. E. Aguirre, and K. Tanaka. Problem features versus algorithm performance on rugged multiobjective combinatorial fitness landscapes. *Evolutionary Computation*, 25(4), 2017.

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In J. Snoeyink and J. Boissonnat, editors, *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pages 253–262. ACM, 2004.

[8] L. Ding, S. Zeng, and L. Kang. A fast algorithm on finding the non-dominated set in multi-objective optimization. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 4, pages 2565–2571 Vol.4, Dec 2003.

[9] D. Feitelson and M. Jettee. Improved utilization and responsiveness with gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 238–261, 1997.

[10] I. Foster and C. Kesselman. The globus project: a status report. In *Heterogeneous Computing Workshop. (HCW 98) Proceedings. 1998 Seventh*, pages 4–18, Mar 1998.

[11] D. Fradkin and D. Madigan. Experiments with random projections for machine learning. In *In KDD 03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 517–522. ACM Press, 2003.

[12] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, Aug. 2014.

[13] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of Computing*, 8(1):321–350, 2012.

[14] J. D. Knowles and D. W. Corne. M-paes: a memetic algorithm for multiobjective optimization. In *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, volume 1, pages 325–332 vol.1, 2000.

[15] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, Oct. 1975.

[16] K. Li, Á. Fialho, S. Kwong, and Q. Zhang. Adaptive operator selection with bandits for a multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 18(1):114–130, 2014.

[17] T. Menouer and C. Cerin. Scheduling and resource management allocation system combined with an economic model. In *the 15th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA 2017)*, 2017.

[18] J. H. M. K. J. Pei. *Data Mining: Concepts and Techniques*. 3rd Edition, elsevier edition, 2011.

[19] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *7th IEEE International Symposium on High Performance Distributed Computing*, 1998.

[20] R. Raman, M. Livny, and M. Solomon. Resource management through multilateral matchmaking. In *the Ninth International Symposium on High-Performance Distributed Computing*, 2000.

[21] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The easy - loadleveler api project. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, 1996.

[22] M. Slaney, Y. Lifshits, and J. He. Optimal parameters for locality-sensitive hashing. *Proceedings of the IEEE*, 100(9):2604–2623, 2012.

[23] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Beowulf cluster computing with linux. *Beowulf Cluster Computing with Linux*, 2002.

[24] F. Teng. *Management des données et ordonnancement des tâches sur architectures distribuées*. PhD thesis, Ecole Centrale Paris, 2011.

[25] J. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[26] L. Wang and E. Gelenbe. Adaptive dispatching of tasks in the cloud. *IEEE Transactions on Cloud Computing*, 6(1):33–45, Jan 2018.

[27] L. Wei, C. H. Foh, B. He, and J. Cai. Towards efficient resource allocation for heterogeneous workloads in iaas clouds. *IEEE Transactions on Cloud Computing*, 6(1):264–275, Jan 2018.

[28] L.-N. Xing, Y.-W. Chen, and K.-W. Yang. Multi-objective flexible job shop schedule: Design and evaluation by simulation modeling. *Applied Soft Computing*, 9(1):362 – 376, 2009.

[29] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32 – 49, 2011.