Parallel Preprocessing for the Optimal Camera Placement Problem

Mathieu Brévilliers, Julien Lepagnot, Julien Kritter, and Lhassane Idoumghar

Abstract—**This paper deals with the preprocessing needed for the optimal camera placement problem, which is stated as a unicost set covering problem (USCP). Distributed and massively parallel computations with graphics processing unit (GPU) are proposed in order to perform the reduction and visibility preprocessing respectively. An experimental study reports that a significant speedup can be achieved, and we give a general heterogeneous parallel approach that brings together these parallel computations. In addition to that, a set-based differential evolution (DE) method is applied to solve 10 instances of the considered problem, and promising results are reported.**

Index Terms—**Distributed computing, graphics processing unit (GPU), optimal camera placement problem, preprocessing, set-based differential evolution (DE) algorithm, unicost set covering problem (USCP).**

I. Introduction

Intelligent video surveillance systems aim at monitoring areas of interest by using appropriate networks of cameras: the placement of these cameras is thus of great importance because of the requested quality of service and the deployment costs. Such an optimal camera placement problem can be stated as a decision problem in a discrete search space [1]: given the set of possible camera locations, find the optimal subset that can meet the operational requirements. In this work, the problem is modelled as a unicost set covering problem (USCP) together with a three-dimensional discretization of the monitored area.

Firstly, the optimization process needs the following input data: the list of points to be covered, the list of possible camera locations, and the lists of points covered by each possible camera location. This so-called visibility preprocessing is performed according to the practical context, where the 3D setting avoids blind spot (with regard to a 2D model), but at the cost of a much larger computational effort [2]. For this reason, it is of high interest to design parallel approaches that can compute these input data within a reasonable time, so that larger problems can be optimized.

Secondly, these input data can be reduced in order to speed up the optimizing process. The main reduction strategy is a variant of the so-called column domination test for non-unicost set covering problems [3] (sets that cover fewer points at the same cost can be removed, instead of those that cover the same points at a higher cost). It consists in discarding the so-called dominated camera locations: for each possible camera location c_1, if another one c_2 can cover the same points as c_1 and some additional points, then c_1 is said to be dominated by c_2 and it can be removed from the set of available camera locations [4]. When the size of the input problem is increasing, this reduction preprocessing becomes quickly time-consuming and it would be interesting to perform it in a parallel way.

Thirdly, the result of the reduction preprocessing is given as input data for the optimization process. Optimal camera placement is a topic of active research, and various approaches have been proposed to solve this problem [2], such as binary integer programming (BIP), particle swarm optimization algorithms (PSO), genetic algorithms (GA), and simulated annealing algorithms (SA).

The main contribution of this paper is the design of a general heterogeneous parallel approach combining distributed and massively parallel computations on graphics processing units (GPU) in order to perform all the preprocessings needed to solve the optimal camera placement problem stated as a USCP. An experimental study shows that significant speedup can be achieved for the two afore mentioned preprocessings. The second contribution is the use of the set-based differential evolution (DE) approach defined in [5] for the optimization of the optimal camera placement problem. Two hybrid set-based DE algorithms are proposed and the relevance of this approach is experimentally demonstrated with promising results on 10 instances.

The remainder of this article is organized as follows. Section II introduces the problem modelling and 10 problem instances. Section III presents in detail the visibility preprocessing and investigates to what extent it can be accelerated by a GPU implementation. Section IV explains how the reduction preprocessing can be performed in a distributed way and reports the resulting speedup according to the size of the cluster. Section V gives the general heterogeneous parallel approach for preprocessing an instance of the optimal camera placement problem. Section VI presents the set-based DE approach together with an experimental study for solving the 10 instances previously defined. Finally, concluding remarks and perspectives are given in Section VII.

II. Problem Description

A. Problem Modelling

In this paper, the optimal camera placement problem is stated as follows: given the camera specifications and a three-dimensional area to monitor, find a minimum set of camera locations (i.e. position and orientation angles) that ensures a full coverage of this area.

Manuscript received October 27, 2017; revised December 12, 2017. This work was supported by the French Agence Nationale de la Recherche (ANR) as part of the OPMoPS project (ANR-16-SEBM-0004).

The authors are with the LMIA Research Laboratory, Université de Haute-Alsace, Mulhouse, France (e-mail: mathieu.brevilliers@uha.fr, julien.lepagnot@uha.fr, julien.kritter@uha.fr, lhassane.idoumghar@uha.fr).

DOI: 10.7763/IJMO.2018.V8.621

The monitored area is discretized by using a regular grid of points: the distance between two consecutive points in the grid is thus equal to 1 unit of length u. The technical specifications of the camera are its horizontal field of view H (in degrees) and its maximal depth of view D (in units of length). The camera has a pyramid of vision (see Fig. 1): the height from the base to the apex is D, and the rectangular base has width $w = 2D \times \tan\left(\frac{H}{2} \times \frac{\pi}{180}\right)$ and height $h = \frac{w}{\mu}$, where $\mu = \frac{1920}{1080}$ is the aspect ratio of a standard full HD camera.

A point of the grid is said to be covered by a given camera, if it is inside the pyramid of vision of this camera. Then, a full coverage of this area means that each point of the grid is covered by at least one camera.

A camera location is defined by a point in the grid that is above or at least on the top of the monitored area, and by two discrete angles (pan and tilt angles) that characterize the orientation of the camera. The pan angle α is the rotation angle along the Z axis, and the tilt angle β is the rotation angle along the Y axis (see Fig. 2).

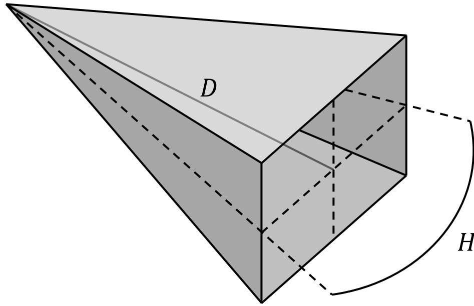

Fig. 1. Pyramid of vision with horizontal field of view H and height D.

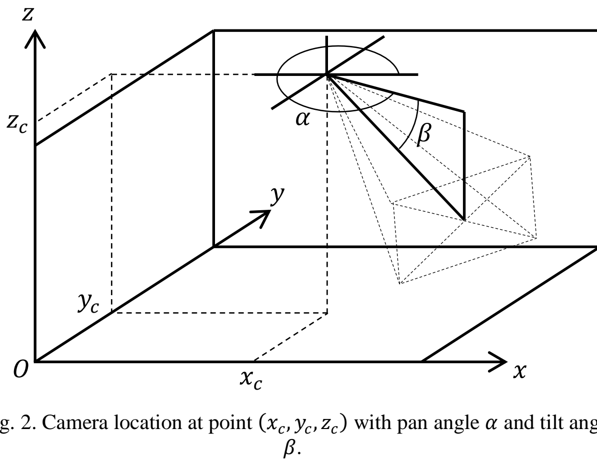

Fig. 2. Camera location at point (x_c, y_c, z_c) with pan angle α and tilt angle β.

The pan and tilt angles are discretized by using a user-defined parameter A, so that the step size is fixed to the value $\frac{\pi}{A}$. It follows that the pan angle α can take N_α=2A different values that range in $[0,2\pi[$. Moreover, given that the cameras are placed above the points to be covered and are oriented downward, the values in $]\pi, 2\pi[$ are not needed for the tilt angle β. Finally, any camera location with pan angle α and tilt angle $\beta \leq \frac{\pi}{2}$ will be identical to the camera location with pan angle $\alpha' = \alpha + \pi$ and tilt angle $\beta' = \pi - \beta$. As a consequence, the tilt angle β can be limited to $N_\beta = \lfloor A/2 \rfloor + 1$ different values.

The afore mentioned optimal camera placement problem can be easily stated as a unicost set covering problem (USCP) in the following way. The points of the monitored area are the set E of elements to be covered. Each camera location covers a subset of E, and we consider here the collection S of subsets of E that correspond to all possible camera locations. Thus, optimizing the camera placement comes down to find the minimum subset of S that covers E.

At first, we define the decision variables:

$$\forall c \epsilon S, x_c = \begin{cases} 1 & \text{if camera location } c \text{ is used,} \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Then, the optimization problem can be written as follows:

$$Min \sum_{c \epsilon S} x_c \quad (2)$$

subject to

$$\forall p \epsilon E, \sum_{c \epsilon S: p \epsilon c} x_c \geq 1 \quad (3)$$

$$\forall c \epsilon S, x_c \epsilon \{0,1\}. \quad (4)$$

The objective function (2) minimizes the total number of cameras. The set of constraints (3) ensures that each point is covered by at least one camera. Equation (4) gives the binary constraints for the decision variables (1).

B. Problem Instances

All experimentations reported in this article are conducted on the problem instances defined in Table I, where X_a, Y_a, and Z_a refer to the size of the monitored area (in units of length), and Z_c refers to the height (in units of length) where the cameras can be placed in the grid.

TABLE I: LIST OF INSTANCES

Instance	X_a	Y_a	Z_a	Z_c	H	D	A
1	10	10	4	5	65	10	4
2	20	20	4	5	65	10	4
3	30	30	4	5	65	10	4
4	40	40	4	5	65	10	4
5	50	50	4	5	65	10	4
6	10	10	4	5	65	20	4
7	20	20	4	5	65	20	4
8	30	30	4	5	65	20	4
9	40	40	4	5	65	20	4
10	50	50	4	5	65	20	4

III. VISIBILITY PREPROCESSING

Once the size of the monitored area and the camera specifications are known, the set of points visible from each possible camera location has to be computed, since it is the input data needed for solving the USCP defined in Section II.

A. Visibility Test

For each point of the monitored area and for each possible camera location, it has to be checked if this point lies or not inside the pyramid of vision of this camera. This visibility test is performed according to the method proposed in [6]. For each tested point $P = [x, y, z]$, new coordinates $P' = [x', y', z']$ are computed by using the following geometric transformations:

$$P' = PTR_Z R_Y \quad (5)$$

where
$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_c & -y_c & -z_c & 1 \end{bmatrix} \quad (6)$$

$$R_Z = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (7)$$

$$R_Y = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8)$$

where T is a translation such that $[x_c, y_c, z_c]$ are the coordinates of the considered camera c, and R_Z and R_Y are a rotation about the Z axis by angle α, and a rotation about the Y axis by angle β, respectively.

Thus P' is now in the coordinate system centered on the pyramid of vision of the considered camera (see Fig. 3), and P' is visible from this camera if all the conditions given below are satisfied:

$$0 \leq x' \leq D \quad (9)$$
$$|y'| \leq \frac{w}{2} \times \frac{x'}{D} \quad (10)$$
$$|z'| \leq \frac{h}{2} \times \frac{x'}{D} \quad (11)$$

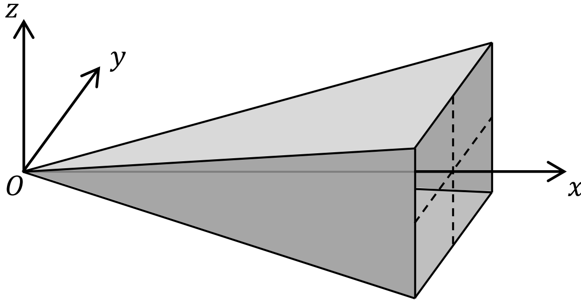

Fig. 3. Coordinate system centered on the pyramid of vision of the camera.

B. Sequential Implementation

The simple sequential algorithm that performs the visibility preprocessing is given in Table II. This algorithm has been implemented on a computer with an Intel Core processor i5-3330 (3.00GHz) and with 4 GB of RAM. Table VI reports the run times (in seconds) observed for the instances defined in Table I.

The first drawback of this sequential algorithm is that the geometric computations needed for the visibility test are costly and the number of visibility tests (given by $|E| \times |S|$) grows quickly when the size of the instance is increasing. The second drawback is that, even without the geometric computations of the visibility test, the two nested loops needed to build S are time-consuming.

On the one hand, all the visibility tests are independent, and on the other hand, GPU devices can execute computations in parallel on multiple data (thanks to their SIMD architecture). In the next section, we propose two GPU implementations of the visibility preprocessing in order to overcome the drawbacks listed above.

TABLE II: SEQUENTIAL VERSION OF VISIBILITY PREPROCESSING

Input:	The set C of possible camera locations.
	The set E of points to be covered.
Output:	The collection S of subsets of E covered by the camera locations of C, i.e. $S = \{cov(c): c \in C\}$, where $\forall c \in C, cov(c) = \{p \in E: c \text{ covers } p\}$.

1	**For** each $c \in C$ **do**
2	**For** each $p \in E$ **do**
3	Compute new coordinates of p according to (5), (6), (7) and (8).
4	**If** conditions (9), (10), and (11) are satisfied
5	Add p in $cov(c)$.
6	**End if**
7	**End for**
8	**End for**

C. GPU Implementations

GPU devices have a highly parallel SIMD architecture, and dedicated parallel computing platforms (like CUDA for NVIDIA GPU devices) allow to easily program general purpose computations with high-level languages. By using the CUDA platform, it is possible to perform heterogeneous parallel computations, where the CPU executes the main program from which parallel subprograms (so-called kernels) can be launched on the GPU. When called from the main program, the code of a kernel is duplicated on the GPU in order to be executed in parallel on multiple data. All these kernel duplicates are executed by CUDA threads, which are organized in groups (called blocks) that contain the same number of threads.

Now, getting back to the optimal camera placement problem, the USCP input data provided by the visibility preprocessing can be presented as a zero-one matrix, where the rows are the elements of E, and the columns are the sets of S: a one in row i and column j indicates that the j-th set of S covers the i-th element of E. Two methods are proposed below in order to fill this matrix with parallel computations performed by threads on a GPU device.

The first idea is to ask a kernel to generate a boolean matrix M in the global memory of the GPU so that each CUDA thread computes one visibility test. Then, M is copied to the RAM, and S is built sequentially by iterating through the matrix (see Table III). It is worth noting that, if the full boolean matrix can not be stored in the GPU global memory or in the RAM, then M (and thus S) can be computed gradually in several iterations according to the available memory space. This method overcomes the first drawback of the sequential implementation and it allows an almost 5-time speedup, as shown by column GPUv1 of Table VI, which reports the run times (in seconds) and the speedup (within brackets). These experimentations have been performed on a computer with an Intel Core processor i5-3330 (3.00GHz), with 4 GB of RAM, and with a NVIDIA GeForce GTX680 GPU device.

TABLE III: FIRST GPU VERSION OF VISIBILITY PREPROCESSING(GPU V1)

Input:	The set C of possible camera locations.
	The set E of points to be covered.
Output:	The collection S of subsets of E covered by the camera locations of C, i.e. $S = \{cov(c): c \in C\}$, where $\forall c \in C, cov(c) = \{p \in E: c \text{ covers } p\}$.

1	Call the kernel that performs the visibility preprocessing and store the results in a boolean matrix M.

2	Copy M from GPU global memory to CPU main memory.
3	**For** each $c \in C$ **do**
4	**For** each $p \in E$ **do**
5	**If** M indicates that c covers p
6	Add p in $cov(c)$.
7	**End if**
8	**End for**
9	**End for**

As we can see in Table III, geometric data about the points and the camera locations are not transferred from CPU main memory to GPU global memory before calling the GPU kernel. Actually, each point p of E (with integer coordinates $[x, y, z]$ in the discrete grid) is labelled with an integer id_p by following the rule below:

$$id_p = x \times (Y_a + 1) \times (Z_a + 1) + y \times (Z_a + 1) + z \quad (12)$$

In this way, all points of the monitored area have consecutive identification numbers that are directly related to their geometric positions in the space. Thus the i-th line of M corresponds the point p such that $id_p = i$.

Then, the same method is used to label each possible camera location c:

$$id_c = x \times (Y_a + 1) \times (Z_{maxC} - Z_{minC} + 1) \times N_\alpha \times N_\beta + y \times (Z_{maxC} - Z_{minC} + 1) \times N_\alpha \times N_\beta + (z - Z_{minC}) \times N_\alpha \times N_\beta + k_\alpha \times N_\beta + k_\beta \quad (13)$$

where Z_{minC} and Z_{maxC} refer respectively to the minimal height and the maximal height (in units of length) allowed in the grid for any camera location, where c is placed at a point with integer coordinates $[x, y, z]$ in the discrete grid, and c is oriented with pan angle $k_\alpha \times \frac{\pi}{A}$ and tilt angle $k_\beta \times \frac{\pi}{A}$, with $k_\alpha \in \{0, \dots, N_\alpha - 1\}$ and $k_\beta \in \{0, \dots, N_\beta - 1\}$. It means that the j-th column of M corresponds the camera location c such that $id_c = j$.

In addition to that, matrices can only be transferred in linear form between CPU main memory and GPU global memory. It means that the 2D matrix M is stored as a 1D matrix. Thus, the result of the visibility test regarding point p and camera location c is stored in the id_t-th element of this 1D matrix, where:

$$id_t = |S| \times id_p + id_c \quad (14)$$

Now, when the kernel is executed, each thread knows the index of its block, and its own thread index in this block. It can thus easily compute the id_t of the visibility test it has to perform. Then, from (14), it can determine the corresponding id_p and id_c. Finally, it can retrieve all geometric information of p and c from (12) and (13). This allows to perform the visibility preprocessing without transferring to the GPU the geometric data of all points and camera locations. According to these explanations, the outline of the visibility preprocessing kernel is given in Table IV.

TABLE IV: OUTLINE OF THE VISIBILITY PREPROCESSING KERNEL

	Input:	$\lvert E\rvert, \lvert S\rvert, Y_a, Z_a, Z_{minC}, Z_{maxC}, N_\alpha, N_\beta, w, h, D$.
	Output:	The boolean matrix M, in linear form.
1	Compute id_t from thread index and block index.	
2	Compute id_p and id_c according to (14).	
3	Compute coordinates of p according to (12).	
4	Compute coordinates and orientation angles of c according to (13).	
5	Compute new coordinates of p according to (5), (6), (7), and (8).	
6	**If** conditions (9), (10), and (11) are satisfied	
7	Set element id_t of M to True.	
8	**Else**	
9	Set element id_t of M to False.	
10	**End if**	

The second idea is to save time when S is built by iterating through M. Actually, M is sparse and there is no need to iterate through the whole matrix in order to build the sets of S. To this aim, an integer matrix M' is used instead of the boolean matrix M. Now, if a camera location c covers a point p, then the element $id_t$of M' is set to id_c (instead of *True*), and to -1 otherwise (instead of *False*). Once the visibility preprocessing kernel terminates, another kernel is called in order to reduce in parallel each line of M' to a list of camera location identification numbers. Each thread of this so-called reduction kernel deals with exactly 1 line of M', and it iterates through the line in order to make all encountered id_c adjacent, so that it forms a list starting at the beginning of the considered line and ending with a -1 value. Afterwards, M' is copied to the RAM, and the sets of S are built by iterating through the resulting lists contained in the lines of M', instead of iterating through the sparse lines of M. The outline of this second GPU version of the visibility preprocessing is given in Table V. This method overcomes the second drawback of the sequential implementation and it allows to achieve a 15-time speedup when compared to the sequential version (see column GPUv2 of Table VI).

TABLE V: SECOND GPU VERSION OF VISIBILITY PREPROCESSING (GPUV2)

	Input:	The set C of possible camera locations. The set E of points to be covered.
	Output:	The collection S of subsets of E covered by the camera locations of C, i.e. $S = \{cov(c): c \in C\}$, where $\forall c \in C, cov(c) = \{p \in E: c \text{ covers } p\}$.
1	Call the kernel that performs the visibility preprocessing and store the results in an integer matrix M'.	
2	Call the reduction kernel that transforms the lines of M' into lists.	
3	Copy M' from GPU global memory to CPU main memory.	
4	**For** each $p \in E$ **do**	
5	**For** each id_c in line id_p of M' **do**	
6	Add p in $cov(c)$.	
7	**End for**	
8	**End for**	

TABLE VI: RUN TIMES OF VISIBILITY PREPROCESSINGS

Instance	$\lvert E\rvert$	$\lvert S\rvert$	CPU	GPUv1	GPUv2
1	605	2 904	0.236	0.078 (3.03)	0.037 (6.38)
2	2 205	10 584	2.912	0.623 (4.67)	0.250 (11.65)
3	4 805	23 064	13.701	2.825 (4.85)	0.988 (13.87)
4	8 405	40 344	41.742	8.568 (4.87)	2.725 (15.32)
5	13 005	62 424	99.780	20.377 (4.90)	6.287 (15.87)
6	605	2 904	0.228	0.051 (4.47)	0.035 (6.51)
7	2 205	10 584	2.920	0.637 (4.58)	0.271 (10.78)
8	4 805	23 064	13.765	2.892 (4.76)	1.057 (13.02)
9	8 405	40 344	41.952	8.762 (4.79)	2.902 (14.46)

10	13 005	62 424	100.268	20.687 (4.85)	6.583 (15.23)

IV. Reduction Preprocessing

A. Description

The visibility preprocessing of Section III provides the input data needed for solving the optimal camera placement problem as a standard USCP. However, before the optimization process, the size of S can be reduced by removing dominated sets of S. Actually, any possible camera location c corresponds to a set $cov(c)$ of S: if there exists another camera location c' such that $cov(c) \subseteq cov(c')$, then c and $cov(c)$ are said to be dominated by c' and $cov(c')$ respectively. In practice, it means that camera location c is not interesting since camera location c' covers the same points as c, and even more. Thus c should not be considered as a possible camera location, and $cov(c)$ can be removed from S before the optimization process.

B. Sequential and Parallel Implementations

The proposed sequential algorithm of the reduction preprocessing is given in Table VII. Firstly, each set $cov(c)$ of S is marked as dominated if we can find a larger set $cov(c')$ of S such that $cov(c) \subseteq cov(c')$, as depicted by lines 1 to 9 in Table VII. Then, we iterate through S in order to build the collection S' of non-dominated sets of S (see lines 10 to 14 in Table VII). The most time-consuming part of this algorithm is obviously the two nested loops of lines 2 and 3, since there can be $O(|S|^2)$ iterations in the worst case.

TABLE VII: Sequential Version of the Reduction Preprocessing

	Input: S.
	Output: The collection S' of non-dominated sets of S.
1	Compute the list T of all sets of S sorted by increasing order of their cardinality.
2	**For** each element $cov(c)$ in T **do**
3	**For** each element $cov(c')$ after $cov(c)$ in T **do**
4	**If** $cov(c) \subseteq cov(c')$
5	Mark $cov(c)$ as dominated.
6	Break for loop
7	**End if**
8	**End for**
9	**End for**
10	**For** each element $cov(c)$ in S **do**
11	**If** $cov(c)$ is not dominated
12	Add $cov(c)$ in S'
13	**End if**
14	**End for**

For this reason, a distributed version of this algorithm has been implemented, where the iterations of the first for loop (line 2 in Table VII) are distributed to n computers of a cluster by using the standard message passing interface (MPI). Since this loop iterates though the list T of all sets of S sorted by increasing order of their cardinality, it is worth noting that the work load will be significantly different from iteration to another. That is why each iteration i of this loop is executed by the $(i \ mod \ n)$-th computer of the cluster: it is a simple way to regularly distribute the work load to the nodes of the cluster.

Table VIII shows the size of the resulting S' for all instances, the run times (in seconds) and the speedups (within brackets) for $n \in \{1,2,4,6,8\}$. Firstly, these experimentations confirm the high importance of the reduction preprocessing for the optimal camera placement problem, since Table VIII shows that between 25 % and 80 % of the possible camera locations are dominated (for instances 5 and 7 respectively), and can thus be discarded. Secondly, the run times reported here correspond to the time spent in the two nested loops of lines 2 to 9 in Table VII, so that it clearly shows the benefit due to the parallel implementation: for $n = 8$, a speedup greater than 6 times (in average, depending on the instance) can be achieved by the proposed distributed approach.

V. General Parallel Approach for the Preprocessings

This section aims at combining some additional sequential preprocessings with the two parallel methods presented in Section III and Section IV, in order to provide a general parallel approach for the preprocessing of optimal camera placement problems.

Firstly, before removing the dominated camera locations as proposed in Section IV, it can be interesting to discard the so-called blind camera locations, i.e. camera locations that can not cover any point of the monitored area. It means that all the empty sets of S have to be removed, and it can be done sequentially in linear time by iterating through S.

Secondly, if a point p of E is covered by only one set $cov(c)$ of S (i.e. a so-called mandatory set), then p and $cov(c)$ can be removed from E and S respectively, since $cov(c)$ is necessarily part of the final solution: the problem can thus be optimized without p and $cov(c)$, and p and $cov(c)$ will be added in the solution at the end of the optimization process. This preprocessing can also be performed sequentially in a time linear with the coverage data, and we can note that there is no mandatory set for the problem instances provided in Table I.

Finally, a general heterogeneous parallel approach can be designed for accelerating the preprocessing computations by using a cluster of computers with GPU devices. Table IX presents the outline of this general method. To the best of our knowledge, it is the first parallel approach combining distributed and GPU computing that is proposed for the preprocessing of optimal camera placement problems.

VI. Optimization with a Set-Based Differential Evolution Algorithm

This section first presents the original differential evolution algorithm. Then, it introduces a set-based version of this algorithm from the literature, which is designed to solve general combinatorial optimization problems. Finally, it investigates the efficiency of this approach in order to solve the optimal camera placement problems from Table I.

A. DE for Continuous Optimization

Differential evolution (DE) is an evolutionary algorithm designed for solving continuous optimization problems [7]. According to [8], it is able to provide high-quality solutions for various theoretical and real-world optimization problems. DE aims at converging on the global best solution by using a population of individuals (i.e. candidate solutions), which are

TABLE VIII: RUN TIMES OF REDUCTION PREPROCESSINGS

Instance	$\|S\|$	$\|S'\|$	$n = 1$	$n = 2$	$n = 4$	$n = 6$	$n = 8$
1	2 904	1 401	0.078	0.045 (1.73)	0.023 (3.39)	0.016 (4.88)	0.012 (6.50)
2	10 584	6 778	0.989	0.540 (1.83)	0.315 (3.14)	0.185 (5.35)	0.153 (6.46)
3	23 064	16 180	5.313	2.813 (1.89)	1.712 (3.10)	0.969 (5.48)	0.913 (5.82)
4	40 344	29 549	19.989	10.591 (1.89)	6.718 (2.98)	3.595 (5.56)	3.400 (5.88)
5	62 424	46 907	54.332	27.996 (1.94)	18.020 (3.02)	9.379 (5.79)	9.056 (6.00)
6	2 904	1 292	0.080	0.046 (1.74)	0.023 (3.48)	0.016 (5.00)	0.012 (6.67)
7	10 584	2 179	1.823	0.947 (1.93)	0.531 (3.43)	0.325 (5.61)	0.267 (6.83)
8	23 064	7 889	10.656	5.507 (1.93)	3.172 (3.36)	1.951 (5.46)	1.581 (6.74)
9	40 344	16 864	35.199	18.213 (1.93)	10.688 (3.29)	6.169 (5.71)	5.586 (6.30)
10	62 424	29 071	87.244	44.647 (1.95)	26.937 (3.24)	14.941 (5.84)	13.457 (6.48)

Evolving from generation to generation with the help of three evolutionary operators. A mutation operator is firstly applied: it generates a mutant individual by adding weighted differences to a reference individual. The basic DE mutation strategy, so-called DE/rand/1, is given hereafter. For each variable j of each individual i of the population Pop:

$$Mut_{i,j} = Pop_{r_1,j} + F \times \left(Pop_{r_2,j} - Pop_{r_3,j}\right) \quad (15)$$

where Mut refers to the mutant population, r_1 , r_2 and r_3 to three randomly chosen integers such that $r_1 \neq r_2 \neq r_3 \neq i$, and $F\epsilon[0,1]$ to the DE scaling factor. Then, the classical so-called binomial crossover operator is applied to each current individual and its corresponding mutant individual in order to generate a new trial individual. Finally, the selection operator compares each current individual with its corresponding trial individual, and it keeps the best of both (according to the considered continuous objective function).

TABLE IX: GENERAL PARALLEL PREPROCESSING APPROACH

	Input:	Problem instance number.
	Output:	The collection S' of non-blind, non-dominated, and non-mandatory sets of S. The subset E' of E that contains the points covered by the sets of S'.
	On each node of the cluster:	
1		Set the instance problem to be processed.
2		Perform the GPUv2 visibility preprocessing, so that each node has his own copy of S.
3		Remove all the empty sets from S, i.e. those corresponding to blind camera locations.
4		Compute the list T of all sets of S sorted by increasing order of their cardinality.
5		Decide if the sets of T are dominated or not, by using the distributed approach described in Section VIII.
6		Send the results of step 5 to the master node.
	On the master node of the cluster:	
7		Aggregate the results of step 5 sent by the other nodes.
8		Build S' according to lines 10 to 14 of Table VII.
9		Remove all mandatory sets from S'.
10		Build E' from S'.

B. Set-Based DE for Combinatorial Optimization

Several adaptations of DE to combinatorial optimization have already been proposed, but most of them can only solve permutation-based combinatorial optimization problems. Only a few works are able to deal with general combinatorial optimization problems and, among them, the set-based approach from [5] seems to be the most interesting to tackle the optimal camera placement problem formulated as a USCP. Furthermore, this approach has been already applied on the traveling salesman problem [5] and the capacitated centered clustering problem [9] with promising results.

By applying this approach to the optimal camera placement problem, a solution is defined as a subset of all the possible camera locations, and the mutation operator is adapted as follows. Arithmetic operations in (15) are replaced with operations on sets, so that for each individual i of the population Pop:

$$Mut_i = Sol_{rand} \cup F \cdot \left(Pop_{r_1} \oplus Pop_{r_2}\right) \quad (16)$$

where Sol_{rand} is a randomly generated feasible solution, $r_1$and r_2 are randomly chosen such that $r_1 \neq r_2 \neq i$, and $\oplus$ is the XOR operator on sets. In [5], it is suggested to set the value of F in order to control the size of Mut_i by using one of the strategies proposed in [10]. However, in the experimentations reported in the next section, F is set to 1, which means that it has no impact on the size of Mut_i, whatever the strategy chosen.

Then, the crossover operator consists in picking some sets from $Pop_i \cup Mut_i$ in order to get a new trial solution. Thus, the crossover operator comes down to solve a subproblem with a lot fewer possible camera locations, and the authors suggest using exact algorithms to get good trial solutions.

C. Experimentations

In the experimental study, the problem instances from Table I are used to compare the CPLEX optimizer, the greedy algorithm from [11] (Greedy, for short), and the set-based DE approach hybridized with CPLEX and Greedy as crossover operators (called DEset-CPLEX and DEset-Greedy respectively).

CPLEX 12.7.0 is used together with ILOG Concert Technology, and the CPLEX optimizer is set up so as to use only one thread: the algorithm is thus deterministic and runs sequentially [12]. Since Greedy is also deterministic, CPLEX and Greedy only need one run for each instance.

For hybridization with the set-based DE approach, CPLEX has a time limit of 10 seconds: thus, either it solves the subproblem, or it provides a feasible solution. On the contrary, there is no time limit for Greedy because it cannot provide a feasible solution before it terminates. Actually, it starts from an empty solution and then, it iteratively adds the best camera location (i.e. the one which covers the maximum number of so far uncovered points) until all points are covered. The parameters of the set-based DE are the following: the population size is set to 20 individuals, and $F = 1$. Moreover, since set-based DE algorithms are not deterministic, 30 runs per instance are performed to get significant statistics.

All algorithms are executed on a computer with an Intel Core i5-3330 processor (3.00GHz) and 4 GB of RAM, with a time limit of 1 000 seconds. Table X shows the results: best values are depicted in bold font, and columns Solution, Mean, and Best report the size of the solutions, i.e. the number of cameras needed to cover the monitored area.

CPLEX is able to find the optimal solution of instances 1, 6, 7, and 8 within the time limit. For instances 2, 3, 9, and 10, it can provide reasonable feasible solutions, but for instances 4 and 5, it gives no better solution than the number of possible camera locations available after the reduction preprocessing: unsurprisingly CPLEX can not be used for large problem instances. Greedy gets quickly a solution for each instance, but they are of poor quality in comparison with the other algorithms. DEset-Greedy is neither competitive with CPLEX on small instances, nor with Greedy on the largest ones. DEset-CPLEX, conversely, gets the better results: it beats the other algorithms for all instances, except for instances 2 and 9 where it is still competitive with CPLEX.

TABLE X: RESULTS AND STATISTICS FOR CPLEX, GREEDY, DESET-CPLEX, AND DESET-GREEDY

	CPLEX			Greedy		DEset-CPLEX			DEset-Greedy		
Instance	Solution	Lower bound	Time	Solution	Time	Mean	Best	Std	Mean	Best	Std
1	**7**	7.00	0.40	10	0.01	**7.00**	7	0.00	7.87	7	0.35
2	**21**	17.52	1 000.00	32	0.12	21.53	21	0.57	29.10	28	0.71
3	56	35.30	1 000.03	63	0.56	**48.07**	45	1.76	71.60	70	0.77
4	29 549	0.00	1 002.24	109	1.82	**92.80**	90	1.37	131.13	126	1.91
5	46 907	0.00	1 002.32	164	4.46	**155.80**	150	2.57	202.73	197	3.19
6	**7**	7.00	1.10	9	0.01	**7.00**	7	0.00	**7.00**	7	0.00
7	**5**	5.00	1.29	**5**	0.05	**5.00**	5	0.00	**5.00**	5	0.00
8	**9**	9.00	26.85	12	0.38	**9.00**	9	0.00	9.37	9	0.49
9	**14**	12.64	1 000.03	19	1.29	14.97	14	0.49	18.90	18	0.61
10	26	18.86	1 000.11	30	3.39	**23.43**	23	0.50	32.97	32	0.76

VII. CONCLUSION

In this paper, the optimal camera placement problem is stated as a USCP. The visibility preprocessing is presented in detail in order to propose an effective parallel version by using massively parallel computations on GPU and a significant 15-time speedup is achieved. It is also shown how to perform the reduction preprocessing with distributed computations and the proposed method leads to a 6-time speedup with a cluster of 8 nodes. A general heterogeneous approach is then given in order to bring together this preprocessings in an efficient way. Afterwards, 10 instances of the optimal camera placement problem are solved by two hybrid set-based DE algorithms, and this method gets promising results in comparison with CPLEX and a greedy algorithm.

For the USCP preprocessing, a first perspective would be to perform less visibility tests by first detecting (and discarding) blind cameras according to their geometric coordinates and orientation angles. It can also be considered to improve the reduction preprocessing by identifying dominated camera locations in a smaller neighborhood related to the considered camera location. Regarding the proposed optimization method, a deeper experimental study should be conducted in order to see the impact of the user-defined parameters (especially F). It will be interesting as well to compare this set-based approach with other efficient state-of-the art algorithms dealing with the optimal camera placement problem or the USCP.

REFERENCES

[1] Horster and R. Lienhart, "Optimal Placement of Multiple Visual Sensors," in H. Aghajan and A. Cavallaro, *Multi-Camera Networks: Principles and Applications*, Elsevier, 2009, ch. 5, pp. 117-138.

[2] J. Liu, S. Sridharan, and C. Fookes, "Recent advances in camera planning for large area surveillance: A comprehensive review," *ACM Computing Surveys (CSUR)*, vol. 49 no. 1, 2016.

[3] J. E. Beasley, "An algorithm for set covering problem," *European Journal of Operational Research*, vol. 31 no. 1, pp. 85-93, 1987.

[4] Z. Naji-Azimi, P. Toth, and L. Galli, "An electromagnetism metaheuristic for the unicost set covering problem," *European Journal of Operational Research*, vol. 205, no. 2, pp. 290-300, 2010.

[5] A. L. Maravilha, J. A. Ramirez, and F. Campelo, "A new algorithm based on differential evolution for combinatorial optimization," in *Proc. 2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence (BRICS-CCI & CBIC)*, Ipojuca, Brazil, 2013.

[6] H. Zhang, L. Xia, P. Wang, J. Cui, C. Tang, N. Deng, and N. Ma, "An optimized placement algorithm for collaborative information processing at a wireless camera network," in *Proc. 2013 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1-6, San Jose, CA, USA, 2013.

[7] R. Storn and K. Price, "Differential evolution – A simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11 no. 4, pp. 341-359, 1997.

[8] S. Das, S. S. Mullick, and P. N. Suganthan, "Recent advances in differential evolution – An updated survey," *Swarm and Evolutionary Computation*, vol. 27, pp. 1-30, 2016.

[9] A. L. Maravilha, J. A. Ramirez, and F. Campelo, "Combinatorial optimization with differential evolution: A set-based approach," in *Proc. the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO Comp'14)*,Vancouver, BC, Canada, 2014.

[10] R. S. Prado, R. C. P. Silva, F. G. Guimarães, and O. M. Neto, "Using differential evolution for combinatorial optimization: A general approach," in *Proc. 2010 IEEE International Conference on Systems Man and Cybernetics (SMC)*, Istanbul, Turkey, 2010.

[11] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4 no. 3, pp. 233-235, 1979.

[12] IBM. IBM Knowledge Center - global thread count. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSSA5P_12.7.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/Threads.html

Mathieu Brévilliers received in 2008 his PhD degree in computer science from the University of Haute-Alsace (UHA), Mulhouse, France. He spent one year at the Grenoble Intitute of Technology (Grenoble INP, France) as temporary lecturer and researcher, and then has been hired by the UHA in 2009 as associate professor. Since 2014, he is member of the "Metaheuristic and Combinatorial Optimization" research team at LMIA Laboratory. His main research interests include massively parallel and distributed hybrid metaheuristics and their applications.

Julien Lepagnot received his PhD in computer science in 2011 from University Paris 12, France. He is an associate professor in computer science at University of Haute-Alsace, Mulhouse, France, in which he belongs to the "Metaheuristic and Combinatorial Optimization" research team at LMIA Laboratory. His main research interests include massively parallel and distributed hybrid metaheuristics, dynamic optimization, and machine learning.

Julien Kritter obtained both his BSc. (2015) and his MSc. (2017) in computer science and applied mathematics from the University of Le Havre, France. His research interests include several branches of artificial intelligence and operational research, in which he is now working towards a PhD. His work in the "Metaheuristic and Combinatorial Optimization" research team at LMIA Laboratory revolves around the application of metaheuristic methods for large-scale optimization.

Lhassane Idoumghar received in 2012 his accreditation to supervise research from University of Haute-Alsace, Mulhouse, France. Since 2015, he is Full Professor with University of Haute-Alsace and he is head of "Metaheuristic and Combinatorial Optimization" research team at LMIA Laboratory. His research activities include dynamic optimization, single/multiobjective optimization, uncertain optimization by hybrid metaheuristics, distributed and massively parallel algorithms.