2017 IEEE International Conference on Robotics and Automation (ICRA)
Singapore, May 29 - June 3, 2017

Parallel Collision Check for Sensor Based Real-Time Motion Planning

Massimo Cefalo, Emanuele Magrini, Giuseppe Oriolo

Abstract— **In this paper we present a real-time collision check algorithm based on the parallel computation capabilities of recent graphics card's GPUs. We show an effective application of the proposed algorithm to solve the task-constrained real-time motion planning problem for a redundant manipulator. We propose a proof-of-concept motion planner based on fast collision check of predicted robot motion over a given planning horizon. Obstacles are avoided exploiting the redundancy of the robot. Reactive velocities are computed for some control points placed on the robot and projected in the null space of the task Jacobian. The approach is validated through simulations in V-Rep environments and experiments on the KUKA LWR-IV 7-DoF manipulator.**

I. Introduction

The presence of robots in everyday life is rapidly growing as well as their applications, which span in increasingly different contexts, always more frequently involving interactions with humans, especially in service robotics. The success of new robotics frontiers involving human robot interactions largely depends on the safety levels of coexistence and collaboration between humans and robots.

Currently, all technologies (no one of which may be still considered mature) are based on the environmental monitoring to acquire information and apply, on the base of a proper strategy, a modification of the robot behavior when necessary to achieve safety. The approaches that lie at the extremes of the possible strategies for the real time applications are *pure reaction* and *fast replanning*. Pure reaction methods consist in avoiding possible collisions by making the robot reactive to the obstacles, for instance by means of repulsive potential fields. Typical problems of these methods are local minima and unstable oscillations in presence of narrow corridors or sudden changes of the environment. Fast replanning methods are indeed planning algorithms based on the idea to re-plan, as fast as possible, a new robot motion whenever possible collisions are detected.

Artificial Potential Fields ([1], [2], [3] - a pioneering work -, [4] , [5] and [6] among others) and variations are amid the first developed and still the most commonly applied strategies for achieving pure control reactiveness.

Opposed to the control-based approaches, methods based on fast replanning are on-line motion planning approaches where paths and trajectories are calculated in the configuration $\times$ time space in real time as proposed in [7]. In [8] the authors use virtual springs and damping elements for a Cartesian impedance controller that generates motion trajectories. In [9] both the robot and the human are represented by a number of spheres and a collision-free trajectory is obtained exploring possible end-effector movements in predefined directions. In [10] a method to model objects using surfaces that closely imitate their shape is studied. The collision cone approach is used to determine collisions between moving objects, whose shape can be modeled by certain type of surfaces. Another method that uses simpler collision cones is presented in [11], where the concept of virtual plane is introduced.

All these works are limited by the computation time required to generate the proposed solution, and consequently are also limited by the dimensionality of the configuration space. Thus, they can only be applied to simple contexts.

Last decade developments in the electronic field has given rise to the multi-core processor technology and, in the same time, to the parallel programming paradigm. In 2007, NVIDIA introduced the CUDA architecture, an hardware and software system for supporting GPU programming, that allows to easily demand certain computations to the GPU cores. The CPUs are nowadays multi cores systems, made of dozens of cores, but GPUs have thousands of cores and implement a much higher degree of parallelism.

More recent works, like [12] and [13], exploit the availability of software libraries for image and point cloud processing, recently implemented on GPU, to propose real time algorithms. Date back only a few years ago the first attempts to exploit GPU programming for realizing real-time motion planners, but the results were still far from being optimal. In [14] the authors propose an optimization-based motion planner by computing collision avoidance objective functions on GPU. The planner provided a solution in a few seconds. In [15] the authors propose a parallel algorithm based on GPUs computations for fast collision detections that, nevertheless, is not intended for real time applications.

Here we address the real-time Task-Constrained Motion Planning (TCMP) problem in dynamic environments for redundant robots. We present a proof-of-concept planner that exploits a novel parallel collision detection algorithm running completely on GPU. The planner and the collision detection algorithm have been implemented and tested in simulation and on a real robot, providing exciting results. The proposed approach opens to several possible extensions and applications.

The paper is organized as follows. Section II and III, respectively, introduce the proposed collision detection algorithm and the proposed planner. Section IV shows some

The authors are with the Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma, Via Ariosto 25, 00185 Rome, Italy. E-mail: {cefalo,magrini,oriolo}@diag.uniroma1.it . This work is supported by the EU H2020 project COMANOID.

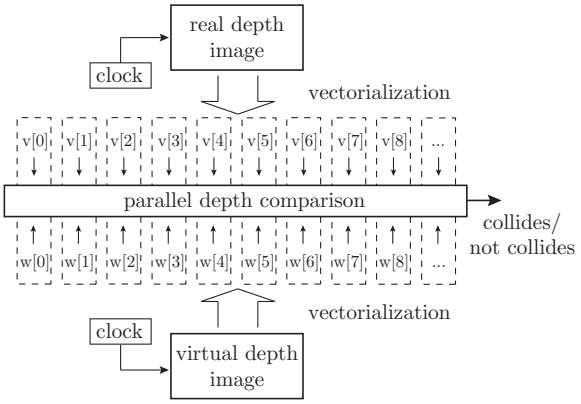

Fig. 1. The parallel collision check.

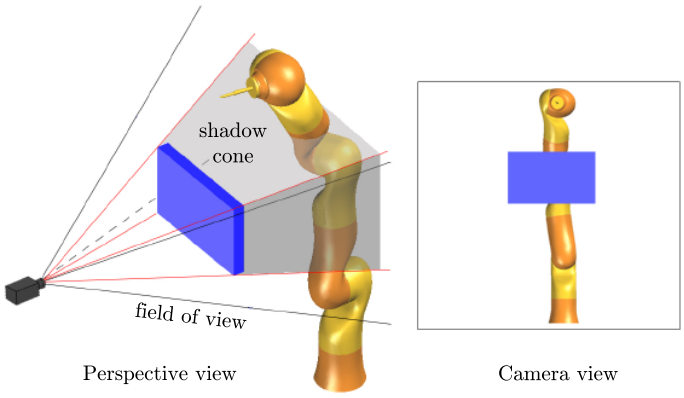

Fig. 2. The robot is in the shadow cone of the obstacle.

preliminary experimental results, while conclusions and possible future developments are discussed in Section V.

II. The Collision Detection Algorithm

A. Overview

The proposed collision detection algorithm is based on the visual feedback provided by a 2.5D image sensor. A 2.5D image sensor is a special type of range camera, that associates to each pixel of a bidimensional image, a value that corresponds to the distance of the 3D point in the Cartesian space to which the pixel refers to, from the camera image plane (depth information). Nevertheless, due to relative occlusions between bodies, not all points in a scene can be associated to a depth information. This is why most of the users consider not completely appropriate to call such sensors 3D sensors and call them 2.5D sensors.

In the last years, some depth image cameras have become very popular mainly due to their diffusion in the gaming world. One the most famous is the Kinect, nowadays largely used also in robotics thanks to its low cost and relatively high reliability.

Our approach consists in processing two 2.5D images with the same resolution: the *real depth image*, representing the obstacles, and the *virtual depth image*, representing only the robot in a future configuration to be tested. They are generated in real time using a Kinect and the robot CAD models. Section II-B describes in details the virtual and real depth images building process.

The two images are loaded into the GPU memory as row vectors with the depth information. Components in the two vectors with the same index correspond to the same pixel in the depth images. A parallel comparison of the depth information for any pair of corresponding components is than executed to detect collisions (see Fig. 1).

If two corresponding pixels of the row vectors result to be in collision an atomic increment is done into a shared memory GPU variable used to count the pixels in collision.

A collision is detected whenever the robot touches an obstacle or is covered (even partially) by an obstacle. We distinguish two possibile cases of mutual occlusions between the robot and the obstacles. The first case of occlusion arises when the robot is in the shadow cone of an obstacle (see Fig. 2). Also in this case, it is not possible to know if the robot is in effect in contact or not with any obstacle, therefore the entire shadow cone generated by the obstacle is prudently considered as part of the obstacle itself and a collision is reported.

The second case occurs when, with respect to the camera point of view, the robot is in front of some obstacles. In this case, in general, using a single static image it is not possible to deduce if the robot is in collision or not. However, in dynamic situations, making some hypothesis on the relative velocities between the robot and the obstacles, it is possible to predict if a collision has occurred or not. The proposed algorithm is able to recognize if the robot is in collision or not in situations like this, using information relative to the previous state and the guess that at the beginning of the motion the robot in a safe configuration.

B. Virtual and real depth images

The virtual depth image is an image of the robot without obstacles artificially created in the depth space. A CAD model of the robot is loaded into a virtual environment using OpenGL. A scene object collects hierarchically the movable components of the robot (for instance, the links of a manipulator) as distinguished nodes, to which it is possible to apply motion commands. The nodes in the scene object are mesh elements that contain all necessary information for the rendering, like vertices position and the orientation of the normal vectors to the surfaces. The goal is to obtain an image that could be subtracted from the camera image to cancel the robot and build a map containing only the obstacles.

Once the CAD model has been loaded, by using the direct kinematics, we move the virtual model to match the real robot configuration. At this point we apply a sequence of linear transformations defined by matrices to obtain a 2.5D image with the same resolution and relative to the same point and field of view of the Kinect. In particular, we first apply a transformation to map the point coordinates from a local reference frame (defined in the CAD model) to a world reference frame, with respect to witch all objects in the environment are defined (M_{model}). Then, we transform the world coordinates into a view-space coordinates making

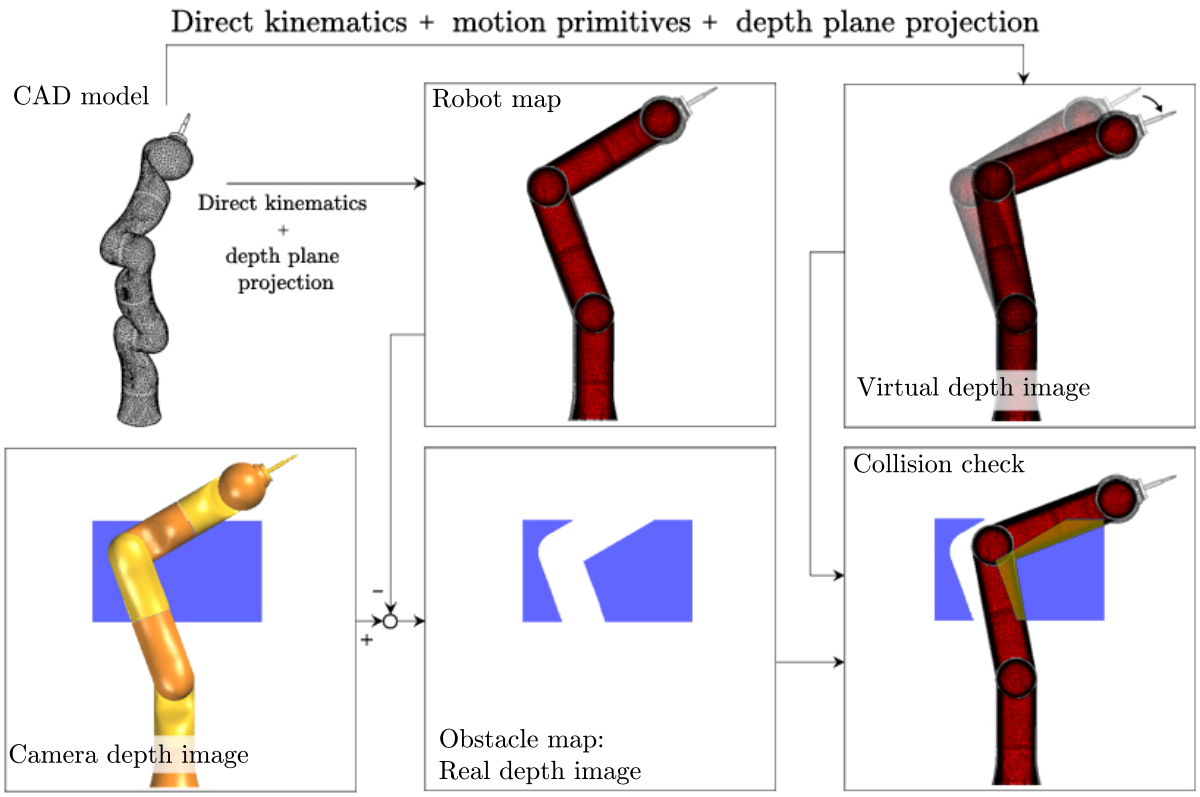

Fig. 3. Collision check process.

everything relative to the camera point of view (M_{view}). This mapping transforms 3D points into 2D points with a depth information. The third transformation determines what is visible and what is not in a given field of view. This mapping projects the coordinates from the view-space to the clip-space (M_{pr}). The last transformation determines the screen point coordinates of a 2D frame (M_{scr}).

The coordinates of a point in the virtual depth image are therefore determined applying the following formula to the 3D points of a CAD model:

$$\boldsymbol{p}_{vdi} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} = M_{scr} \cdot M_{pr} \cdot M_{view} \cdot M_{model} \cdot \boldsymbol{v}_{local} \quad (1)$$

where vdi stands for virtual depth image and $\boldsymbol{v}_{local} = (v_x \ v_y \ v_z)^T$.

The real depth image is an image containing only the obstacles. It is obtained subtracting to the camera image a virtual depth image representing the robot only. Any inaccuracy in the geometric model arises in a non perfect cancellation of the robot in the camera image. Other factors that may also influence the robot cancellation are the calibration of the virtual camera (used to generate the virtual depth image) w.r.t. the real one. In order to filter the imperfections and obtain a robust process, the virtual depth image is dilated a few pixels before the cancellation. This operation produces an enlargement of the geometric models corresponding to displace only the points on the surfaces, keeping unchanged the position of any interior point and their relative distances. The result is similar to assign a thickness to the surfaces. If a pixel representing the robot has a shorter depth than its corresponding pixel in the camera image, then the latter is deleted from the camera image. The result is a map containing only the obstacles that are visible to the Kinect. Finally a maximum depth is assigned to each pixel belonging to the holes left by the robot cancellation. This process can be considered as a first level of image filtering.

A second filter acts on the counter of the pixels in collision. If the counter has a value greater than a given threshold, a robot collision is returned. The aim of this threshold is to filter image imperfections generated by the camera and thus skip false collision results. This parameter can be properly chosen according to the obstacles' geometry, bearing in mind that if an obstacle is placed in the scene so as to appear very thin and be represented only by a few pixels in the obstacle map, high threshold values imply false negatives and the inability to detect collisions with such obstacle.

Finally we mention the possibility to define an obstacle clearance. This can be implemented considering in the depth plane, for any pixel to be tested for collisions, a set of further four points placed at the vertices of a square centered with the pixel. The size of the square, may change accordingly to the depth of the pixel in which it is centered, in order to realize a fixed tolerance in the Cartesian space. All pixels of the square than must be tested for collisions (recall that this may be done in parallel on the GPU).

Fig. 3 summarizes the whole collision check process.

C. Tools for parallel computing

The proposed collision detection algorithm has been implemented as a GPU distributed program and is able to

exploit the parallelism of a graphic board. In particular, it is based on the CUDA framework and the OpenGL library. OpenGL is a platform independent rendering library that provides hardware accelerated rendering functions. CUDA is a framework for mass parallel programming based on the NVIDIA architecture. Differently from the CPU, in a GPU each core has the capability to execute hundreds of processes simultaneously, thanks also to a high speed memory closely interconnected to the cores. This high degree of parallelism gives to any GPU based application a huge performance boost.

In order to develop scalable and parallel applications that are able to exploit the power of a GPU, new parallel programming models are required. CUDA architecture is an emerging technology that gives to the developers the possibility to access GPU resources for general purposes, making the GPU an open architecture similarly to the CPU. We used the GLSL shading language to directly program some tasks in the GPU (like, the robot augmentation, the transformations defined in (1) and the collision tests) and the OpenGL for CAD model manipulation.

III. A Real Time Validation-Based Planner

In this section we present a real-time planner to solve in real time the Task-Constrained Motion Planning problem presented in [16]. We will adopt the same terminology and framework therein introduced. Differently from that work, here, for simplicity, we assume that the robot is not subject to nonholonomic constraints.

Consider a robot whose generalized coordinates are represented by an n_q-dimensional vector $\boldsymbol{q}$ which takes values in the configuration space $\mathcal{C}$. The robot is assumed to move in a workspace $\mathcal{W}$, subset of $I\!R^2$ or $I\!R^3$, populated by possibly moving obstacles.

Since there are no differential constraints, the kinematic model of the robot consists of simple integrators

$$\dot{\boldsymbol{q}} = \boldsymbol{v} \tag{2}$$

reflecting the fact that the generalized coordinates $\boldsymbol{q}$ can move arbitrarily in $\mathcal{C}$.

The robot is assumed to be constrained to perform assigned tasks expressed in m-dimensional coordinate vectors $\boldsymbol{y}$, associated to the space $\mathcal{Y}$. Task coordinates and configuration coordinates can be related through the kinematic map $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{q})$ which, differentiated, assumes the form

$$\dot{\boldsymbol{y}} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}},$$

where $\boldsymbol{J} = \partial \boldsymbol{f}/\partial \boldsymbol{q}$ is the $m \times n$ task Jacobian. For reasons that will be soon clear, we will assume that the robot is kinematically redundant with respect to the assigned task, that is $m < n$.

Let $\boldsymbol{y}_d(t) \in \mathcal{Y}$ be a non-derogable assigned task, i.e. a task that cannot be violated for any reason. The possibility to relax the task constraint in presence of unavoidable collisions is outside the scope of this paper. The robot redundancy is therefore required to provide the robot with the necessary flexibility to continue the execution of the task in presence of obstacles.

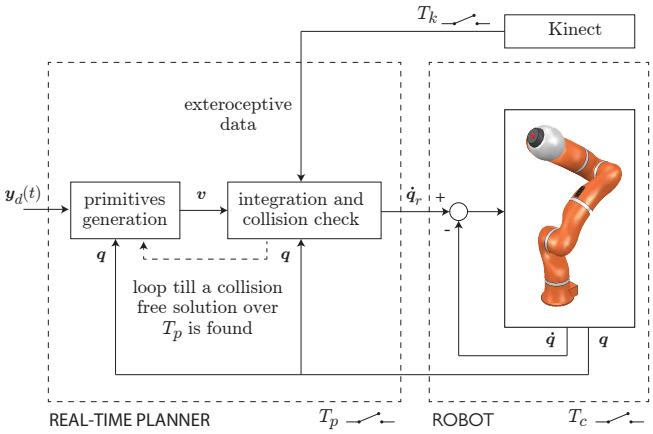

Fig. 4. General scheme of the proposed real-time planner.

The robot is also assumed to be equipped with a depth sensor fixed in the environment, that provides real-time information on the depth of any object that falls within a given field of view (e.g. a Kinect camera).

The problem consists in real-time planning a robot trajectory in $\mathcal{C}$ that is collision-free and realizes the desired task. Observe that the assigned task is specified as a function of time, therefore, there is no possibility for the planner to find a solution simply scaling in time a motion that results in collision. In case a collision free motion cannot be found, the robot stops and the planer returns a failure.

A. The algorithm

The proposed planner is an anytime task-constrained motion planner that generates collision free motion on a *planning horizon* T_h using a kinematic controller (the motion generation scheme, further discussed in this Section). Figure 4 synthesizes the scheme of the planner.

Joint velocities are chosen to guarantee the exact and safe execution of the task during the period T_h by setting

$$\boldsymbol{v} = \boldsymbol{J}^{\dagger}(\dot{\boldsymbol{y}}_d + \boldsymbol{K}\boldsymbol{e}_{\boldsymbol{y}}) + (\boldsymbol{I} - \boldsymbol{J}^{\dagger}\boldsymbol{J})\boldsymbol{w} \tag{3}$$

where $\boldsymbol{J}^{\dagger}$ is the pseudoinverse of the task Jacobian, $\boldsymbol{K}$ is a positive definite matrix, $\boldsymbol{e}_{\boldsymbol{y}} = \boldsymbol{y}_d - \boldsymbol{y}$ is the task error and $\boldsymbol{w}$ is an $(m-n)$-dimensional auxiliary velocity vector which can be arbitrarily specified without affecting the task (recall that $\boldsymbol{I} - \boldsymbol{J}^{\dagger}\boldsymbol{J}$ is the orthogonal projection matrix in the null space of $\boldsymbol{J}$).

Vector $\boldsymbol{w}$ produces self motions that do not influence the position or the orientation of the end-effector. It can be freely defined, and is usually exploited to accomplish a secondary task compatible with the main one. In Section III-B we propose a possible strategy for choosing $\boldsymbol{w}$ that allows collision avoidance without affecting the assigned task.

The motion generation scheme generates the joint velocities defined by equation (3) using a proper choice for the auxiliary vector $\boldsymbol{w}$. Than the kinematic model of the robot is integrated with a sample period T_c, to predict the motion of the robot over the planning horizon T_h. T_c is the sample

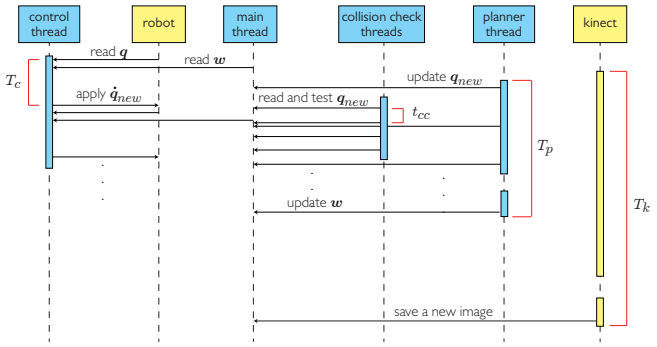

Fig. 5. Time sequence chart.

rate of the low-level controller which is responsible for the actuation of the commanded joint velocities to the real robot.

Figure 5 shows a diagram sequence of the main parts of the algorithm that may run in parallel. Each part is represented in the chart as a separate thread. The main thread represents the core of the algorithm put in charge of the communications with all other processes. The control thread is responsible for updating, every T_c seconds, the joint velocity commands on the robot actuators. Such commands are computed using equation (3) where $\boldsymbol{w}$ is updated by the planner every T_p seconds. The collision check process is executed on the GPU and is run in parallel to any other process. This process runs continuously, performing the collision tests on each point of a camera image. Being the collision check much faster than the camera refresh rate, the same test is repeated many times on the same image, but without influencing the behavior of the planner. Whenever a new camera image becomes available, the collision tests are automatically performed on the new image. This process communicates with the main thread to signal if a collision has been detected or not. In the end, the planner thread is the part of the algorithm responsible for generating the auxiliary velocity vector $\boldsymbol{w}$ using an appropriate strategy.

During the integration process, new robot configurations are checked for collisions with the obstacles detected at the beginning of the period T_h. In this preliminary work we consider the obstacles fixed during the period T_h. This hypothesis works fine also in case of obstacles that move with low velocities with respect to the robot and to the camera sampling frame.

If the integration of the kinematic controller can be computed without collisions over the entire period T_h, then the pseudo velocities $\boldsymbol{w}$ are used to apply new velocity commands to the robot. The vector $\boldsymbol{w}$ is kept constant for a *planning period* $T_p < T_h$, which should be long enough to give to the planner the time to compute new collision-free velocities but also as short as possible so as to update $\boldsymbol{w}$ with high frequency. In particular, fixed T_c, T_p and T_h must satisfy

$$T_p \geq q \cdot t_{cc} \cdot T_h / T_c \tag{4}$$

where t_{cc} is the collision check time, T_h/T_c is the number of integration steps and q is the number of different alternatives for the auxiliary vector $\boldsymbol{w}$ to be tested. Figure 6 shows the

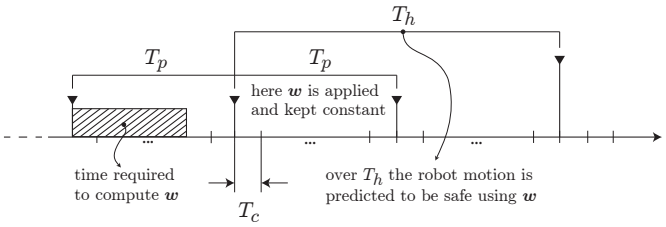

Fig. 6. Planning time T_p, planning horizon T_h and control period T_c.

relation between T_c, T_p and T_h. In equation (4), q may be put to 1 if the different choices for $\boldsymbol{w}$ can be tested in parallel.

B. Primitive-based motion generation scheme

The idea at the heart of our planner is to use the proposed real-time collision check algorithm to predict if a given choice for $\boldsymbol{w}$, held constant, generates safe motions over the planning horizon T_h. This is obtained by integrating equation (3) over the interval T_h and checking the obtained configuration space path for collisions with the obstacles detected at the beginning of the planning phase.

We assume that the obstacles do not disturb the motion of the end-effector along the assigned trajectory. If not, the assigned planning problem has no solutions.

A safe motion requires to move the manipulator as much as possible far from the obstacles. Exploiting the redundancy this can be achieved while pursuing the primary task. Taking inspiration from a classical approach (see, for instance, [17]), we exploit the null space projection to obtain self motions that stave off some robot points from the obstacles. In particular, we consider a set of p control points along the robot structure. We use the control points c_i to compute the distances with the obstacles and consequently generate an auxiliary velocity vector $\boldsymbol{w}$ in such a way to increases such distances and move the robot far from the obstacles.

In general, if $f(\boldsymbol{q})$ is a performance function defined in the configuration space, to maximize f, in accordance with the gradient descent method, the configuration space velocities can be defined as

$$\dot{\boldsymbol{w}} = \alpha \nabla f$$

where $\nabla f = (\partial f/\partial q_1, \ldots, \partial f/\partial q_n)^T$ and α is a positive gain. On the base of this general result, we propose the following definition for the auxiliary velocity vector $\boldsymbol{w}$:

$$\boldsymbol{w} = \sum_{i=1}^{p} \dot{\boldsymbol{q}}_{c_i} \tag{5}$$

where $\dot{\boldsymbol{q}}_{c_i}$ is the velocity in the joint space of the control point c_i defined as

$$\dot{\boldsymbol{q}}_{c_i} = \begin{cases} 0 & \text{if } d_i > d_{th} \\ (d_{th} - d_i)/d_{th}\ \alpha_i \boldsymbol{J}_{c_i}^{\dagger} \nabla d_i & \text{if } d_i \leq d_{th} \end{cases} \tag{6}$$

in which, ∇d_i is the gradient of the shortest distance between the control point c_i and the obstacles in the Cartesian space, $\boldsymbol{J}_{c_i}$ is the Jacobian matrix associated to the control point c_i and d_{th} is a threshold, beyond which the reaction to the obstacles is null. The term $(d_{th} - d_i)/d_{th}$ helps to obtain

a smooth transition with the motion guided only by the pseudoinverse ($\boldsymbol{w} = 0$).

Equation (5) defines the auxiliary robot joint velocities as the sum of p reactive velocities applied to p control points. The reactive velocity of a single control point is zero if it is far enough from any obstacle. Using equation (5), the robot moves on the assigned task keeping the distances between the control points and the obstacles always within an assigned threshold, compatibly with the primary task.

Equation (5) generates a single choice $\bar{\boldsymbol{w}}$ for $\boldsymbol{w}$. The planner integrates the motion generation scheme using $\boldsymbol{w} = 0$ and $\boldsymbol{w} = \bar{\boldsymbol{w}}$. Therefore, in our implementation $q = 2$, but since the two integration processes may run in parallel on two different CPU threads, in equation (4) we can put $q = 1$. At the beginning, the planner tries to move the robot on the task using $\boldsymbol{w} = 0$. If the integration with $\boldsymbol{w} = 0$ can be completed without collisions over the planning horizon T_h, the choice $\boldsymbol{w} = 0$ is passed to the control thread which uses again equation (3) to compute and send velocity commands to the robot any T_c seconds. This behavior corresponds to move the robot on the task in absence of collisions with a minimum norm displacement in the configuration space (minimum energy).

If, on the contrary, the choice $\boldsymbol{w} = 0$ generates collisions, than the value $\bar{\boldsymbol{w}}$ is considered and tested and in the positive case passed to the control thread to generate the velocity commands for the robot.

Whenever a choice for $\boldsymbol{w}$ results to be collision free, than such value is considered in the next integration period as the first possible choice. This helps to maintain the same value for a primitive over a longer period of time, avoiding a jerky behavior of the robot that may instead result if the primitive is continuously changed.

C. Implementation issues

In order to compute the Cartesian distance $d(\boldsymbol{O}, \boldsymbol{P})$ between a robot point $\boldsymbol{P}$ and an obstacle point $\boldsymbol{O}$ using the depth information, we use the formula proposed in [5] that we recall here for convenience:

$$d(\boldsymbol{O}, \boldsymbol{P}) = \sqrt{v_x^2 + v_y^2 + v_z^2} \geq \sqrt{v_x^2 + v_y^2}$$

with

$$\begin{aligned} v_x &= \frac{(o_x - c_x)d_o - (p_x - c_x)d_p}{f \cdot s_x} \\ v_y &= \frac{(o_y - c_y)d_o - (p_y - c_y)d_p}{f \cdot s_y} \\ v_z &= d_o - d_p \end{aligned}$$

where (o_x, o_y) and (p_x, p_y) are the coordinates in the depth space of the points $\boldsymbol{O}$ and $\boldsymbol{P}$ respectively, d_o and d_p are their depth w.r.t. the camera, c_x and c_y are the pixel coordinates of the center of the image plane (on the focal axis), f is the focal length of the camera and s_x and s_y are the dimensions of a pixel in meters. The last five parameters constitute the intrinsic parameters of the camera and can be usually retrieved by the device manufacturer.

TABLE I

T_c	T_p	T_h	d_{th}	Kp	α_i
0.005 s	0.02 s	0.04 s	0.3 m	20	15

Equation (6) requires to compute the gradient of the distances between the control points and the obstacles. Exploiting the CUDA functionalities, it is possible to compute the distances between any control point and any point on the obstacle map in parallel, and than select the minimum d_i. All this computations can be done by an atomic operation, whose cost in terms of execution time, is almost zero.

Once the distances d_i have been calculated, the gradient in (6) may be computed as

$$\nabla d_i = R(v_x \quad v_y \quad 0)^T / d \tag{7}$$

where R is the rotation matrix necessary to align the robot frame with the camera frame.

During the integration of the motion generation scheme in the prediction phase, in order to speed up the computations, the integration step may be set up as a multiple of T_c, at the expense of a larger numerical error. Besides, as long as the planning horizon is chosen small enough (for instance less than 1 second), it is admissible to check for collisions only the last configuration obtained by the integration process. These considerations may be exploited to setup a longer planning horizon T_h with respect to the planning period T_p.

IV. Experimental results

We present here some preliminary results of the proposed planner applied to the KUKA LWR-IV 7-DOF manipulator. We tested the planner on V-REP, a software development kit for robotic simulations, and also on the real robot. The hardware platform was a 64-bit Intel Core i7-4790 CPU, equipped with 16 GB DDR3 RAM, running at 4 GHz. The high-performance graphic board had a NVIDIA GTX970 GPU, organized in 1884 CUDA cores and capable of 26624 concurrent threads.

Both in simulation and in the real experiment we assigned to the end-effector of the robot a linear task in the Cartesian space, with the time law $(x_0, y_0 + 0.2\sin(2\pi t/T), z_0)^T$, where $(x_0, y_0, z_0)^T$ was the position of the end-effector at the beginning of the motion and T was 4 seconds in VREP and 8 in the real experiment.

We considered three control points placed at the center of the joints 3, 4 (the elbow) and 5. Other points could be added along the structure, but any control point too close to the base or to the end-effector is useless for reconfiguring the manipulator.

Table I collects some planner parameters. We used the same values for the simulation and the real experiment. The average time for running the collision test was around 1.5 milliseconds on the entire camera image.

We dilated the robot CADs along the normals 3 cm before the cancelation and we fixed to 500 the threshold for the collisions counter. Finally, we fixed to 50 cm the obstacle clearance.

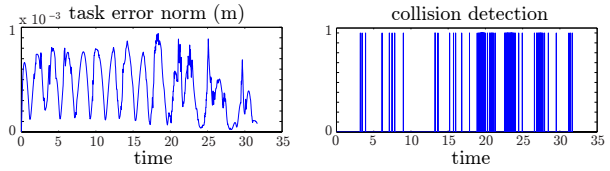

Fig. 7. Simulation: Error norm (on the left) and collision detections on the predicted motion (on the right).

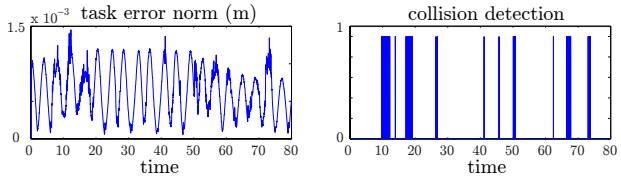

Fig. 10. Real experiment: Error norm (on the left) and collision detections on the predicted motion (on the right).

Figure 9 shows some snapshots of the simulation. The robot starts moving on the assigned task from a safe configuration and continues to move along the task using the pseudoinverse till its posture becomes too close to a human which is in its workspace (3-rd frame). In correspondence to that configuration, the predicted motion using the pseudoinverse generates collisions and the planner applies to the robot the auxiliary velocities $\boldsymbol{w}$ that tend to increase the distances between the control points and the human. Later, the human approaches to the manipulator (4-th frame) and the planner computes new auxiliary velocities, with larger values that yield a prompt reaction of the robot, which self-reconfigures while continuing to perform the assigned task.

Figure 7 shows, on the left, the norm of the error along the assigned task (always smaller than one millimeter) and, on the right, the results of the collision tests executed in real time during the prediction phase.

Similarly to the simulation, some snapshots of the experiment performed on real robot are shown in Figure 8. Starting from a collision free configuration, the robot moves along the assigned task. About 11 seconds later, a fixed obstacle is placed close to the robot, with a resulting collision on predicted motion. New reactive velocities $\boldsymbol{w}$ push away the control points from obstacles (2-nd and 3-th frame). In the second part of the experiment, a moving obstacle approaches to the robot. The collisions are still avoided by the robot, maintaining the end-effector on the desired task (black line). Also in this case the norm of the task error is almost always smaller than one millimeter. Figure 10 shows the results together with the collisions detected in real time on the predicted motion.

Note that compared to the methods purely reactive, as the method of the artificial potential fields, the solution found by the planner leads the robot to move away from obstacles only what is strictly necessary, passing near them as permitted by the settings on safe distance.

V. Conclusions

We developed a novel collision detection algorithm based on GPU parallel computing and a real-time planner for redundant robots. To the best of our knowledge, we present here the first a real-time task constrained motion planning algorithm based on parallel collision check. It is a proof-of-concept planner that prove the effectiveness of the our collision detection method.

Interesting future developments may concern the possibility to predict the obstacles' motion using a filter, or doing a 3D reconstruction based on camera images. A further extension may also concern the application of the presented algorithms to an experimental setup that uses more than a single Kinect, as in [18].

References

[1] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*. Springer-Verlag, 2009.

[2] J. Minguez, F. Lamiraux, and J. Laumond, *Motion planning and obstacle avoidance*, B. Siciliano, O. Khatib ed. Springer Handbook of Robotics, 2008.

[3] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *Int. J. of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986.

[4] P. Ogren, N. Egerstedt, and X. Hu, "Reactive mobile manipulation using dynamic trajectory tracking," in *2000 IEEE Int. Conf. on Robotics and Automation*, San Francisco, USA, 2000.

[5] F. Flacco, T. Kroger, and A. De Luca, "A depth space approach to human-robot collision avoidance," in *2012 IEEE Int. Conf. on Robotics and Automation*, Saint Paul, USA, 2012.

[6] S. Haddalin, S. Belder, and A. Albu-Schaeffer, "Dynamic motion planning for robots in partially unknown environments," *IFAC World Congress*, vol. 44, no. 1, pp. 6842–6850, 2011.

[7] J. Vannoy and J. Xiao, "Real-time adaptive motion planning (ramp) of mobile manipulators in dynamic environment with unforeseen changes," *IEEE Trans. on Robotics*, vol. 24, no. 5, pp. 1199–1212, 2008.

[8] S. Haddadin, H. Urbanek, S. Parusel, D. Burschka, J. Romann, A. Albu-Schffer, and G. Hirzinger, "Real-time reactive motion generation based on variable attractor dynamics and shaped velocities," in *2010 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, Taipei, Taiwan, 2010.

[9] L. Balan and G. M. Bone, "Real-time 3d collision avoidance method for safe human and robot coexistence," in *2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, Beijing, China, 2006.

[10] A. Chakravarthy and D. Ghose, "Generalization of the collision cone approach for motion safety in 3d environments," *Autonomous Robots*, vol. 32, pp. 243–266, 2012.

[11] F. Belkouche, "Reactive path planning in a dynamic environment," *IEEE Trans. on Robotics*, vol. 25, no. 4, pp. 902–911, 2010.

[12] K. Kaldestad, S. Haddadin, R. Belder, G. Hovland, and D. Anisi, "Collision avoidance with potential fields based on parallel processing of 3d-point cloud data on the gpu," in *2014 IEEE Int. Conf. on Robotics and Automation*, Hong Kong, China, 2014.

[13] S. Ueki, T. Mouri, and H. Kawasaki, "Collision avoidance method for hand-arm robot using both structural model and 3d point cloud," in *2015 IEEE/SICE International Symposium on System Integration (SII)*, 2015.

[14] R. Wagner, U. Frese, and B. Buml, "3d modeling, distance and gradient computation for motion planning: A direct gpgpu approach," in *2013 IEEE Int. Conf. on Robotics and Automation*, Karlsruhe, Germany, 2013.

[15] J. Pan and D. Manocha, "GPU-based parallel collision detection for fast motion planning," *Int. J. of Robotics Research*, vol. 31, no. 2, pp. 187–200, 2012.

[16] G. Oriolo and M. Vendittelli, "A control-based approach to task-constrained motion planning," in *2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, St. Louis, MO, 2009, pp. 297–302.

[17] A. Maciejewski and C. Klein, "Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments," *Int. J. of Robotics Research*, vol. 4, no. 3, pp. 109–117, 1985.

[18] F. Flacco and A. De Luca, "Real-time computation of distance to dynamic obstacles with multiple depth sensors," *IEEE Robotics and Automation Letters*, vol. 2, no. 1, pp. 56–63, 2017.

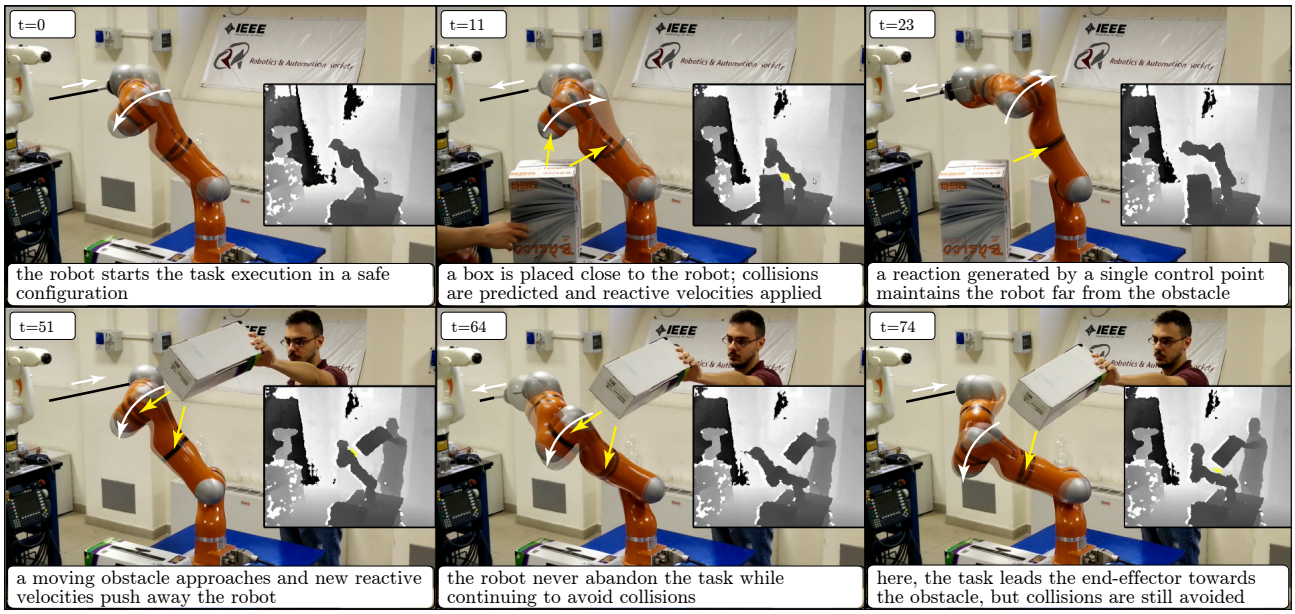

Fig. 8. Sample frames of the experiment on the real robot. The assigned task is highlighted in black, with its orientation specified by a white arrow. The robot motion is highlighted by white curved arrows. Collisions are highlighted with yellows dots in the depth image, on the right. Cartesian reactive velocities are represented by yellow arrows.

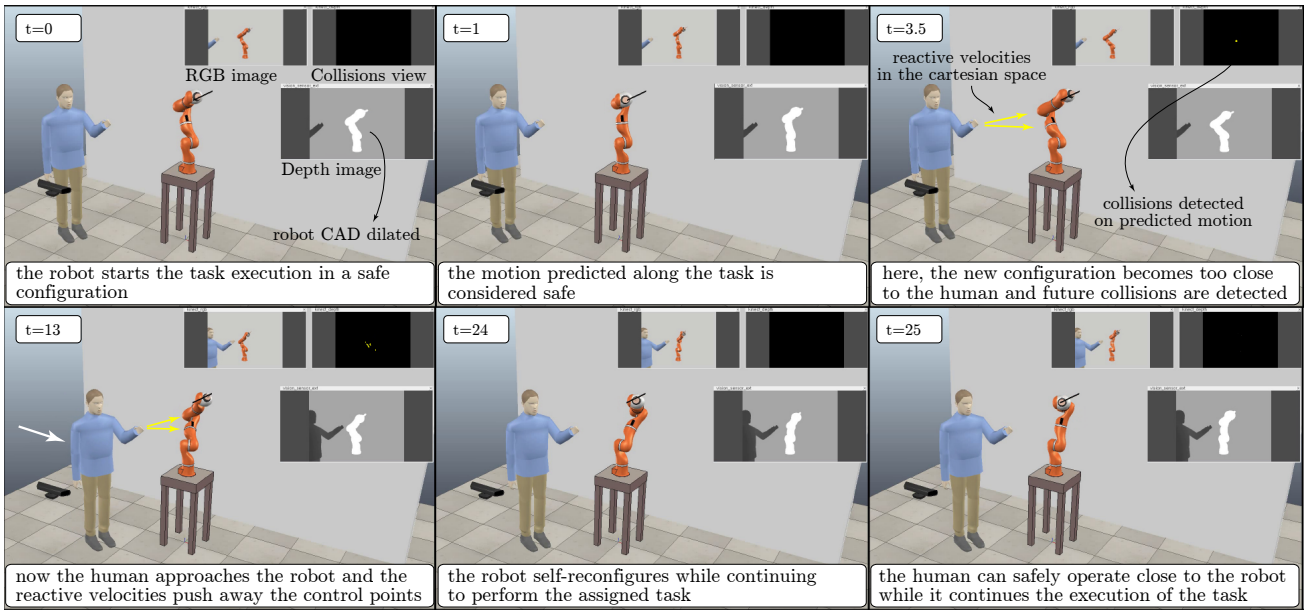

Fig. 9. Sample frames of the simulation run in V-REP. The assigned task is highlighted in black. The RGB image captured by the Kinect is on the center top. Collisions are highlighted with yellows dots in top right image. The depth image in represented in the box on the right. The Cartesian reactive velocities are represented by yellow arrows and human's approaching direction is represented by a white arrow.