Received July 30, 2021, accepted August 9, 2021, date of publication September 1, 2021, date of current version September 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3109744

A Parallelization Algorithm for Real-Time Path Shortening of High-DOFs Manipulator

JI HWAN SEO*, HYUNTAE LEE*, AND KYOUNG-DAE KIM, (Member, IEEE)
Department of Information and Communication Engineering, Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu 42988, South Korea

Corresponding author: Kyoung-Dae Kim (kkim@dgist.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant through Korean Government by the Ministry of Science and ICT (MSIT) under Grant 2019R1F1A1059496, and in part by the Daegu Gyeongbuk Institute of Science and Technology (DGIST) Research and Development Program of MSIT under Grant 21-CoE-IT-01.

*Ji Hwan Seo and Hyuntae Lee contributed equally to this work.

ABSTRACT The paths generated by sampling-based path planning are generally not smooth and often generate multiple unnecessary robot posture changes in the task space. To mitigate such issues with a planned path from sampling-based path planners, shortcut-based path shortening algorithms are commonly adopted in the field of robot manipulator path planning as a post-processing step. In this paper, we analyze shortcut-based algorithms and propose a new approach based on the idea of parallelism for faster path shortening so that it can be more applicable in environments where a path has to be generated as quickly as possible to avoid collisions with other moving objects around the manipulator. Through performance comparisons in simulations, it is shown that the proposed approach can obtain a well-shortened as well as much smooth path compared to the original path faster than conventional shortcut-based algorithms and an optimization-based approach developed for collision-free path generation.

INDEX TERMS Path shortening, path smoothing, parallelization, manipulator.

I. INTRODUCTION

Research on collaborative robotics is very active these days due to its great usefulness in the era of the Fourth Industrial Revolution. Robot path planning is one of the key areas in the field of collaborative robotics research. In path planning, the sampling-based approach is the most commonly used technique especially for industrial robot manipulators, whose configuration space dimensions are generally high.

Sampling-based path planning algorithms generate a collision-free path by connecting sampled robot configurations in the configuration space. This approach has been widely used in robot path planning because it has better ability to find a path in high degree of freedom (DOF) configuration space, compared to existing techniques. Probabilistic Roadmap (PRM) [1], one of the most representative sampling-based planners, finds a path from start to goal by connecting some of the sampled configuration nodes (or simply, nodes) created in the collision-free space in advance. Rapidly Exploring Random Tree (RRT) [2] is another representative planner in this category. RRT generates a path by repeating the process of finding the closest node to a randomly generated sample in the configuration space, creating a new node at a certain distance, and performing collision checking. Afterwards, to cope with static and dynamic environments, various sampling based path planners have been introduced, such as Dynamic Roadmap (DRM) [3], Dynamic RRT [4], g-Planner [5], and so on. However, one common drawback of sampling-based path planning is that the planned paths are generally not smooth and often generate unnecessary robot posture changes in task space [6]. This is mainly due to the difference between the task space and the configuration space where a path is originally planned. A common approach to address this issue is to perform post-processing on the planned path such as smoothing around corners or eliminating unnecessary posture changes.

Parametric curves such as a polynomial curve, Bézier curve, or splines can be used to smooth the curvature of a path. These techniques generate a curved path based on the given control points. For instance, Wang *et al.* [7] generated a smooth trajectory between two adjacent points by planning the joint acceleration as a 4th-order polynomial. Huang *et al.* [8] proposed a method of smoothing the corners

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

of a given path to have G^2 continuity using cubic B-spline. Han and Liu [9] performed path smoothing with Bézier curve after finding a collision-free path for a 6-DOF manipulator using RRT. However, since sampling-based path planners typically generate a path with unneeded control points, there is a possibility that a path smoothed by these techniques alone still contains unnecessary motions. For this reason, removing unnecessary posture changes from a given path is more effective for improving path quality.

One way to improve the quality of the planned path is to use an optimization technique. For example, Covariant Hamiltonian Optimization for Motion Planning (CHOMP) [10] defines a cost function with a combination of a collision term and a smoothness term, and finds the optimal path using a covariant gradient descent method. Stochastic Optimization for Motion Planning (STOMP) [11] is similar to CHOMP, but it performs optimization even if the gradient of the cost function cannot be calculated. Trajectory Optimization for Motion Planning (TrajOpt) [12] was introduced to address the complexity issue of CHOMP by formulating a trajectory optimization problem as a sequential quadratic programming. However, these methods typically take a long time to get the optimal path, and so it may not be applicable in environments where a path has to be generated as quickly as possible to avoid collisions with other moving objects.

Another approach is to shorten the middle part of a given path by using path-pruning or shortcut-based algorithms. Path-pruning is a sequential process starting from the beginning node of a given path to find the other farthest node in the path that can be connected without collision [13], [14]. However, due to its sequential nature of the algorithm, a path-pruning algorithm cannot sufficiently shorten a given path in many cases. A Shortcut algorithm was proposed to overcome the limitation of the original path-pruning algorithm [15]. Roughly speaking, the shortcut algorithm is an iterative path-pruning approach which repeats a two-step process, a random selection of a partial segment of the given path and then path-pruning for the selected path segment, until the given path is sufficiently shortened. Because of its simplicity and effectiveness, the shortcut approach is still widely used for path post-processing [16], [17].

In this paper, we propose a new shortcut-based path shortening algorithm, named as *Parallelized Shortcut (ParaSC)*, that can shorten a path much faster than conventional shortcut-based algorithms. The proposed algorithm is designed based on the idea of parallelism to overcome some limitations of existing algorithms which we identified through in-depth analysis of existing algorithms with supporting evidence obtained via simulations. We also present the results of performance comparison between ParaSC and other algorithms to demonstrate the improved path shortening performance of our algorithm. In particular, an optimization-based method, which is another representative approach commonly used to generate a smooth optimal path, is also considered for comparison. The main contributions of this work can be summarized as follows.

Algorithm 1 Shortcut Algorithm (Π: Planned Path)

```
1: loop
2:     $\Pi_s \leftarrow$ Random selection of a partial segment of $\Pi$
3:     $\Pi'_s \leftarrow$ Perform shortening on $\Pi_s$ via linear interpolation
4:     if $\Pi'_s \in C^{\text{free}}$ then
5:         Replace $\Pi_s$ with $\Pi'_s$
6:     end if
7: end loop
```

- *Effective path shortening through parallelization*
  To the best of our knowledge, the proposed method is the first shortcut-based algorithm utilizing parallelization strategy applied for the path post-processing of high-DOF robot manipulators. Since ParaSC divides a given path into multiple segments and considers all possible shortening methods for each segment, it is possible to generate a shortened path where unnecessary posture changes are almost eliminated in a few iterations.
- *Parallel path shortening speed improvement using GPU*
  It utilizes the parallel processing power of modern GPUs to expedite the path shortening process for high-DOF manipulators. According to the simulation results, ParaSC can shorten a path in less than a few tens of milliseconds in most cases even within an environment with many surrounding obstacles, which outperforms other path post-processing algorithms in terms of shortening speed. Such computation time of ParaSC for path shortening is fast enough to be used alongside humans within the same workspace and it is expected that ParaSC can be potentially useful in the field of collaborative robotics.

The remainder of this paper is organized as follows. Section II reviews the representative shortcut-based path shortening approaches. Our proposed approach is presented in Section III. Implementation details of the proposed algorithm are presented in Section IV. Performance evaluation and comparisons are shown in Section V. Finally, this work is concluded in Section VI.

II. SHORTCUT-BASED ALGORITHMS

The basic steps of the shortcut algorithm are shown in Algorithm 1 where Π is a path generated by a sampling-based path planning algorithm consisting of a sequence of nodes and edges connecting adjacent nodes, and C^{free} is the collision-free configuration space.

A simple but naive approach for the random selection of a partial path segment Π_s, shown in line 2 of Algorithm 1, is to use only existing nodes of the planned path Π. However, this simple strategy may limit the number of selectable path segments if the number of nodes constituting path Π is not large enough, and hence eventually impact on the quality of path shortening. To overcome this limitation, several approaches have been proposed. In [18], the midpoint of an edge was

Algorithm 2 Adaptive Partial Shortcut Algorithm (Π: Planned Path, Π_{ref}: Reference Path)

```
1: loop
2:     $\Pi_s \leftarrow$ Random selection of a partial segment of $\Pi$
3:     Update AWD using $\Pi$ and $\Pi_{\text{ref}}$
4:     Select a fixed number of joints according to AWD
5:     $\Pi'_s \leftarrow$ Perform shortening on $\Pi_s$ via linear interpolation on selected joints
6:     if $\Pi'_s \in C^{\text{free}}$ then
7:         Replace $\Pi_s$ with $\Pi'_s$
8:     end if
9: end loop
```

used to increase the number of selectable path segments for the given path. Hsu *et al.* [19] put additional nodes on the planned path in order to achieve the same goal. In addition to these approaches, the shortcut algorithm has been extended in various directions such as a meta-algorithm that combines the shortcutting concept and a path hybridization technique [20], and a shortcutting approach for jerky trajectories bounded in velocity, acceleration and so on [21], [22].

In general, through the iterative path-pruning process, the shortcut algorithm can generate a well-shortened path in terms of path length in configuration space. However, the shortest path found in configuration space is not always desirable in task space since it may contain unnecessary joint movements when moving the manipulator from one configuration to another. To address this problem of the original shortcut algorithm, Partial Shortcut (PSC) algorithm was proposed in [23]. The overall structure of PSC algorithm is same as Algorithm 1. But the main idea which makes PSC algorithm different from the original shortcut algorithm is to focus on one joint motion when it shortens a path segment instead of considering all joints. Referring to line 3 in Algorithm 1, PSC algorithm first selects a joint and then performs the shortening process on Π_s only for the selected joint. At each iteration step, a joint is selected randomly according to a pre-defined weight distribution over all joints that approximately describes how much each of the joints can influence to the robot posture change.

Since PSC algorithm only considers one joint motion for each path shortening iteration step, one can easily see that the algorithm will inevitably take more iteration steps than the original shortcut algorithm to obtain a well-shortened path. Furthermore, as the path is updated at each iteration step, the pre-defined weight distribution for joint selection does not appropriately reflect the effectiveness of each joint for path shortening. To address these issues of PSC algorithm, a new algorithm, called Adaptive Partial Shortcut (APSC) algorithm, was proposed recently in [24]. The basic steps of the APSC algorithm are shown in Algorithm 2 where Π_{ref} is the straight line path in configuration space that connects the first node and the terminal node of Π without considering obstacles.

In APSC algorithm, the first issue of PSC algorithm is easily addressed by increasing the number of joints to be considered for path shortening. As described in line 4 of Algorithm 2, the number of joints to be selected is pre-determined and fixed. Also, to reflect the effectiveness of each joint for path shortening as the path Π is being updated, APSC algorithm selects joints according to the weight distribution which is updated at each iteration step, called the adaptive weight distribution (AWD). AWD is updated so that joints which have large differences between Π and Π_{ref} are to be selected more likely than others.

III. PARALLELIZATION FOR FAST PATH SHORTENING

As explained in Section II, APSC can generate a well-shortened path faster than PSC. However, it is still not fast enough to be used in dynamic environments. In this section, we discuss the limitations of APSC algorithm in terms of computation time and propose a new shortcut-based path shortening framework that can achieve much faster convergence toward a well-shortened path enough to be used in real-time path planning situations.

A. LIMITATIONS OF APSC ALGORITHM

Since APSC algorithm is essentially a shortcut-based algorithm, it still relies on an iterative process for path shortening. Therefore, it is necessary to increase the number of iterations to get a well-shortened path using APSC. Referring to Algorithm 2, one main step that contributes to the number of iterations is the collision checking in lines 6 and 7. The probability that the shortened path segment Π'_s be collision-free is highly dependent on both the path segment Π_s selected in line 2 and also the set of joints selected in line 4. Thus, it is critically important for path shortening to wisely choose a set of joints for a given path segment Π_s. Also, even though APSC algorithm selects a fixed number of joints according to AWD, there is still a substantial possibility that Π'_s fails to pass the collision checking step in line 6. We argue that this is because the set of joints selected for a given path segment Π_s is generally not the best choice for Π_s in terms of the number of joints as well as the combination of selected joints. This claim is supported by the simulation results of the 6-DOF manipulator shown in Figure 1 and Table 1.

Figure 1a shows an exemplary path Π generated by a sampling-based path planning algorithm and three partial path segments Π_s of Π are shown in Figure 1b, 1c, and 1d. As shown in the figure, the planned path Π consists of 15 nodes along the path starting from the node 1 and ending at the node 15. In Table 1, we present the results of our exhaustive simulations for path segment shortening by considering all possible combinations of joints for a given number of selected joints. As shown in the table, the maximum number of joints that can generate a collision-free shortened path segment Π'_s can be different if the given path segment Π_s is different. For example, for the case of path segment Π_s in Figure 1b, the maximum number of selectable joints for collision-free Π'_s is six while it is only two for the

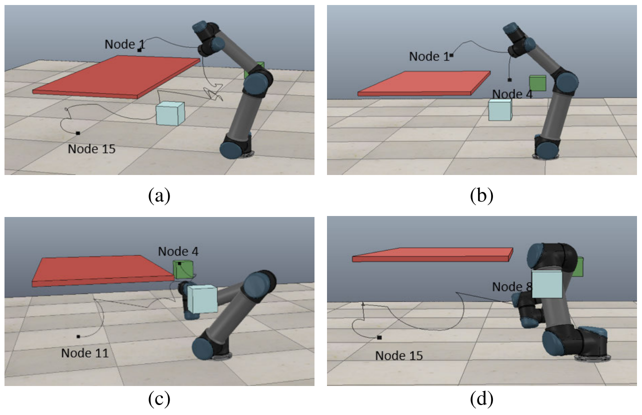


FIGURE 1. An example of path generated by a sampling-based path planning for a 6-DOFs UR-5 manipulator. (a) Planned path Π. (b) A path segment Π_s from node 1 to node 4. (c) A path segment Π_s from node 4 to node 11. (d) A path segment Π_s from node 8 to node 15.

TABLE 1. Number of joint combinations for collision-free Π_s' for given path segment Π_s.

# selected joints	# joint combinations	# joint combinations for $\Pi_s' \in \mathcal{C}^{\text{free}}$		
		Figure 1b	Figure 1c	Figure 1d
1	6	5	3	2
2	15	11	2	2
3	20	13	0	2
4	15	9	0	1
5	6	4	0	0
6	1	1	0	0

case of path segment Π_s in Figure 1c. Also, we note that if we fix the number of joints to be selected as in APSC algorithm, say to three for example, then the collision checking step in line 6 of APSC algorithm will fail and hence increase the number of iterations. Furthermore, even if we fix the number to two, one can argue that there are still some issues with APSC algorithm. Specifically, in our result for the case of path segment Π_s in Figure 1c, the two pairs of joint combinations for successful collision-free path segment Π_s' generation are turned out to be (1, 6) and (2, 6). However, since APSC algorithm selects joints randomly according to AWD, there is no guarantee that the algorithm will select one of these joint combinations. Also, if we choose joints with a pre-defined fixed number, it is easy to see that the algorithm eliminates any possibility to generate a better shortened path segment with either smaller or larger number of joints.

B. PARALLELIZATION STRATEGY FOR FAST PATH SHORTENING

In previous section, we discussed the limitation of the existing state-of-the-art path shortening algorithm, APSC, in terms of computation time. In this section, we propose our approach which is also a shortcut-based algorithm but with a parallelism framework to overcome the above mentioned limitation of other algorithms and make it fast enough to be used in dynamic environments.

First, one intuitive way to reduce the number of iterations is to perform path shortening on multiple path segments simultaneously. For this, we select a set of partial path segments that are non-overlapping each other except the first and the last nodes in each path segment. With this parallelized processing over multiple path segments, one can easily expect that the total number of iterations can be substantially reduced.

As we pointed out above, the strategy that selects a fixed number of joints randomly according to weight distribution (AWD in the case of APSC) can be problematic. To overcome this limitation, we also propose a parallelization scheme in joint selection. As shown in our simulation results, since the probability of success in the collision checking process is highly dependent on the number of joints to be selected as well as the combination of selected joints for a given number of joints, it is very important to choose a set of joints for a given path segment in the way to increase the success probability in collision checking. With a parallelization scheme in mind, we propose to explore all possible combinations of joints simultaneously and choose one of them that is collision-free and has the best shortening effect.

Figure 2 shows the overall structure of the proposed parallelization framework, named as Parallelized Shortcut (ParaSC) algorithm, that integrates the two parallel processing strategies explained above. The computation steps of the proposed path shortening process are outlined in Algorithms 3 and 4. For the purpose of parallelized

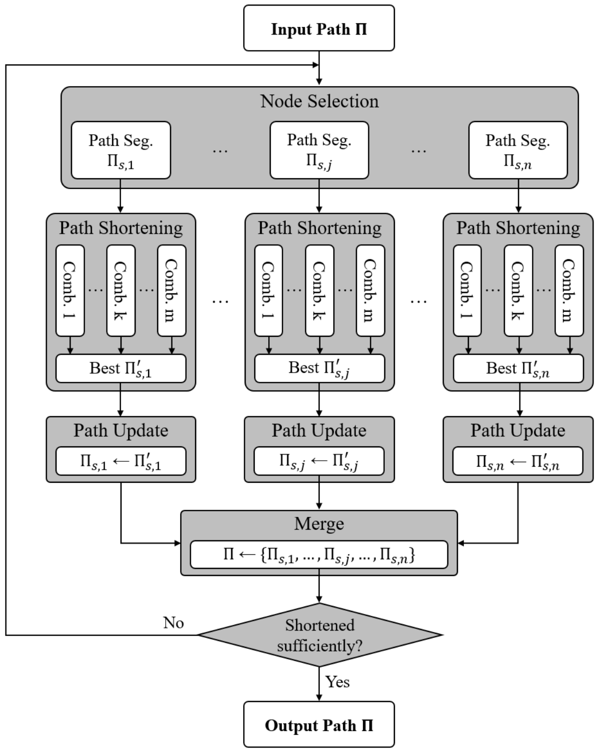


FIGURE 2. Proposed parallelization framework for shortcut-based path shortening.

Algorithm 3 Parallelized Shortcut (ParaSC) Algorithm (Π: Planned Path)

```
1:  Let $\gamma_1, \gamma_2, \ldots, \gamma_m$ be the set of all possible joint combination
2:  loop
3:      $\{\Pi_{s,1}, \Pi_{s,2}, \ldots, \Pi_{s,n}\} \leftarrow$ Partition $\Pi$ into $n$ path segments
4:      $\sum_i = \emptyset, \forall i \in \{1, 2, \ldots, n\}$
5:      Run In Parallel: $\forall i \in \{1, 2, \ldots, n\}, \forall j \in \{1, 2, \ldots, m\}$
6:          $\Pi'_{s,i} \leftarrow$ Path shortening on $\Pi_{s,i}$ (Algorithm 4)
7:          $\sum_i = \sum_i \cup \Pi'_{s,i}$
8:      For each $\sum_i$ s.t. $\sum_i \neq \emptyset$
9:          Choose one $\Pi'_{s,i}$ from $\sum_i$ w.r.t. a measure for path shortening quality
10:         Replace $\Pi_{s,i}$ with $\Pi'_{s,i}$
11: end loop
```

Algorithm 4 Parallelized Path Shortening on $\Pi_{s,i}$ for the jth Joint Combination Case

```
1:  Let $Q_j$ be the set of joints in the $j$th joint combination
2:  $\Pi'_{s,i} \leftarrow$ Linear interpolation of joints in $Q_j$ from the first node to the last node of $\Pi_{s,i}$
3:  Run In Parallel: $\forall l \in \{1, 2, \ldots, L\}$ where $L$ is the user-specified maximum number of interpolated node samples for collision checking
4:      $c_l \leftarrow$ Generate an interpolated node sample on $\Pi'_{s,i}$
5:      if $c_l \in C^{\text{free}}$ then $r_l = 0$
6:      else $r_l = 1$
7:      end if
8:  if $r_l = 0$ for all $l$ then return $\Pi'_{s,i}$
9:  end if
```

shortening of multiple path segments, the algorithm first partitions the given path Π into n path segments as shown in line 3 of Algorithm 3. This can be done easily by randomly selecting $n + 1$ nodes from the path and dividing the path into n path segments based on these selected nodes. Then, as the main step for the parallelized path shortening process shown in lines from 4 to 7, the algorithm applies the path shortening calculation for each path segment $\Pi_{s,i}$ and for all possible joint combinations γ_j where $i \in \{1, , 2, \cdots, n\}$ and $j \in \{1, 2, \cdots, m\}$. Here, the outline of path shortening calculation process on each path segment $\Pi_{s,i}$ is presented in Algorithm 4 where the linear interpolation is performed first for shortening the path segment and then collisions are checked along the shortened path segment as shown in lines from 3 to 7 of the algorithm. Next, if the shortened path segment is clear from collisions, then the shortened path segment $\Pi'_{s,i}$ is returned to the main algorithm and stored in the set Σ_i in Algorithm 3 for further processing where the subscript i represents the path segment number. After completion of path shortening for all i and j, the algorithm chooses the best shortened path segment $\Pi'_{s,i}$ for each path segment i w.r.t. a certain path shortening quality measure. As the main purpose of the process is to shorten a path, we chose to use the path length as the measure for path shortening quality. Once these steps are done in lines from 8 to 9 for each path segment, then the algorithm replaces the original path segment $\Pi_{s,i}$ with the shortened one $\Pi'_{s,i}$ if there is any. Finally, this process is repeated until the entire path Π is shortened sufficiently.

IV. IMPLEMENTATION

In this section, we present the details on the implementation of the parallelization steps in ParaSC algorithm, i.e., lines 5 and 6 in Algorithm 3, on a Graphics Processing Unit (GPU). A GPU usually has more than hundreds of cores and this makes a GPU more suitable than a Central Processing Unit (CPU) for applications that need to process a massive amount of data rapidly in parallel. There are two representative frameworks for GPU programming: Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). Since CUDA usually requires less programming effort than OpenCL [25], CUDA was used in this work. Below, we first provide a brief overview of the GPU architecture, and then describe how we parallelize the path shortening process in consideration of GPU architecture.

A. GPU ARCHITECTURE

In the GPU architecture, a *thread* is defined as a sequence of operations, and the GPU is specifically designed to excel at executing thousands of threads in parallel. Each thread executes a user-defined function called a *kernel*, and the same kernel can be executed in parallel by multiple threads.

In CUDA, a thread hierarchy can be configured by specifying two configurations: The number of thread blocks (or simply, block) and the number of threads per block. A thread block is a group of threads. Both can be configured in the form of 1D, 2D, or 3D. On a current GPU, a block can contain up to 1024 threads. Once the thread hierarchy for a kernel is configured, threads are executed by the streaming multiprocessors (SMs) of the GPU, which are components of GPU, that contain several CUDA cores. The SM executes threads in groups of 32 threads called a *warp*. If the threads allocated to the SM are not multiples of 32, virtual threads are generated so that the number of threads is a multiple of 32, and they are filled in the last warp [26]. Since virtual threads do not perform any valid operations, it is desirable to configure the thread hierarchy so that the number of threads per block can be a multiple of 32 in order to maximize the utilization of GPU capacity [27]. NVIDIA recommends the number of threads per block in multiples of 64 such as 192 or 256 [28].

There are 6 types of memory in a GPU that can be accessed by threads. Figure 3 shows the hierarchy of GPU memory and Table 2 shows the characteristics of each memory [29]–[31]. *Register* is mainly used to store variables which are frequently used by each thread [32]. *Local memory* is used when each thread needs more space than the total

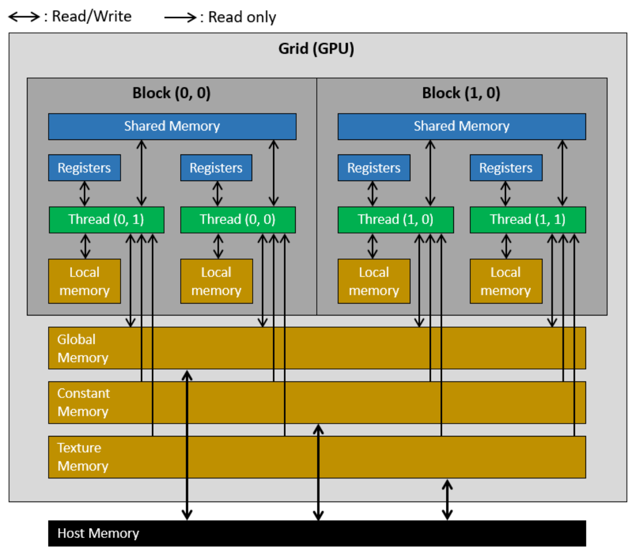


FIGURE 3. **GPU memory hierarchy.**

TABLE 2. **Characteristics of each GPU memory.**

Memory	Scope	Access latency	Persistent
Register	Thread	0-4 cycle	No
Local memory	Thread	400-600 cycles	No
Shared memory	Block	1-6 cycles	No
Global memory	Global	400-600 cycles	Yes
Constant memory	Global	400-600 cycles Few cycles (if cached)	Yes
Texture memory	Global	400-600 cycles Few cycles (if cached)	Yes

allocated registers. *Shared memory* allows multiple threads in each thread block to share intermediate operation results. *Global memory* is a space where data is exchanged between the host PC and GPU. This memory is typically used when previous data is required for the next kernel run [32]. *Constant memory* and *texture memory* can also receive data from the host PC, but data stored in them can only be read. Threads can access data relatively quickly in some cached areas of these memories. Constant memory is optimized for broadcasting data to multiple threads [33], and texture memory can be used to improve computing performance when the memory access pattern has spatial locality [34]. To increase computing efficiency, it is important to consider these characteristics when storing data in each memory available in the GPU.

B. PARALLEL INTERPOLATED NODE GENERATION

As shown in Algorithms 3 and 4, ParaSC algorithm has a hierarchy of parallelizations. Once the path segment $\Pi_{s,i}$ and the set of joints Q_j for path shortening are selected, the algorithm tries to shorten the selected path segment $\Pi_{s,i}$ via linear interpolation. However, to be a valid shortened path segment, it is necessary that the shortened path segment is collision-free. Since this is one of the most computationally expensive tasks in general, to check whether a path is free from any collisions, ParaSC algorithm utilizes another level of parallelization strategy for collision checking of a given path segment, which corresponds to lines from 3 to 7 in Algorithm 4. Our strategy for collision checking of a path is to generate node samples along the path as many as possible and check whether any of the node samples is in collision with surrounding obstacles. If all node samples are collision-free, then the algorithm declares that the linearly interpolated path segment is also collision-free and returns it as a valid shortened path segment $\Pi'_{s,i}$.

In this section, we first describe the details of our implementation for concurrent generation of node samples that will be used for collision checking later. Figure 4 shows the overall process of the proposed parallel interpolated node sample generation. In our implementation for this process, we configure the thread hierarchy as follows. First, each joint combination is assigned to each (thread) block so that the jth joint combination corresponds to block j. Thus blocks are configured in 1D form. Second, each block is configured in 2D form so that threads within a block can process the same computation simultaneously but working on a different path segment as well as different joint. For example, the blue thread shown in Figure 4 processes its calculations for the fifth joint in the second edge of the given path. Here, it is worth noting that, as shown in Figure 4, a path segment for shortening consists of a group of consecutive path edges.

The first step of the parallel interpolated node sample generation process is to determine the number of node samples to be generated for each path edge, and then generate node samples along the path. In the figure, these steps are represented as Steps 1 and 2 respectively. Specifically, in Step 1, a thread associated with the ith joint and the jth path edge calculates the angle change of the ith joint from the starting node and the ending node of the jth path edge of the path. Once all threads within a block finish their computation, the length of each path edge in configuration space is approximated as follows: For the jth path edge,

$$\Delta\theta_j = \left(\sum_i \Delta\theta_{i,j}^2\right)^{1/2} \tag{1}$$

where $\Delta\theta_{i,j}$ is the angle change calculated by the thread associated with the ith joint and the jth path edge.

Finally, the number of node samples N_j to be generated on the jth path edge is determined by

$$N_j = \left\lfloor \frac{\Delta\theta_j}{\delta\theta} \right\rfloor \tag{2}$$

where $\lfloor a \rfloor$ is the maximum integer not exceeding a and $\delta\theta$ is the user-defined constant for the resolution of node sample distance in configuration space. Clearly, the smaller $\delta\theta$, the more samples generated on a path edge because the distance between samples is smaller. Once N_j is determined for all path edges, N_js are stored in the shared memory within the block so that it can be used for further processing in the next step.

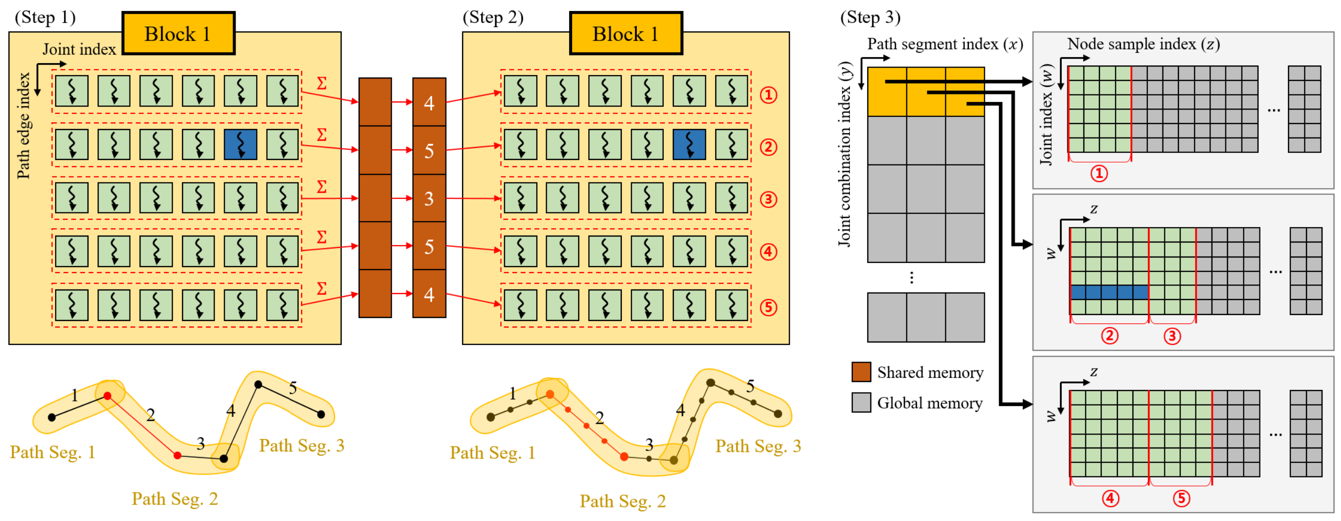


FIGURE 4. Parallel interpolated node sample generation process. (The path example shown in the figure consists of five path edges and is divided into 3 path segments. The small box with arrow represents thread and the brown box represents shared memory in each block.)

After the completion of Step 1, threads are now ready to generate node samples along the path edge assigned to each thread. Specifically, if a thread is associated with the ith joint and the jth path edge, the thread samples N_j number of uniformly spaced joint angles over the interval of $[\theta^1_{i,j}, \theta^1_{i,j} + \Delta\theta_{i,j})$ where $\theta^1_{i,j}$ is the ith joint angle of the first node in the jth path edge. We note here that, if we let $\Theta_{i,j}$ be the set of sampled joint angles, then elements in $\Theta_{i,j}$ are ordered starting from the first node of the path edge and $\theta^1_{i,j}$ has to be the first element in the set. Once the set $\Theta_{i,j}$ are generated for all i and j, node samples generated along the path can be easily constructed later by collecting joint angles from each joint. For example, if we denote the kth node sample along the jth path edge as q^k_j, then q^k_j for the 6-DOF robot manipulator can be determined as a vector formed by

$$q^k_j = \left[\theta^k_{1,j}\ \theta^k_{2,j}\ \theta^k_{3,j}\ \theta^k_{4,j}\ \theta^k_{5,j}\ \theta^k_{6,j}\right]^{\mathrm{T}} \tag{3}$$

where $\theta^k_{i,j}$ is the kth element in $\Theta_{i,j}$.

The last step in the node sample generation process is to store data in $\Theta_{i,j}$ for all i and j in global memory so that they can be used later in the parallel collision checking process which is done in line 5 of Algorithm 4. For this, we use a 4D array data structure as shown in Figure 4 so that each thread can access the memory efficiently without causing any synchronization issues. Let $a_{x,y,z,w}$ be the element in the 4D array located at the position with (x, y, z, w) indices where x, y, z, w corresponds to the index number associated with each path segment, joint combination, node sample, and joint, respectively. For example, the blue-colored elements in the 4D array shown in the figure for Step 3 are accessed by the thread associated with the second path segment ($x = 2$), the first joint combination case ($y = 1$), and the fifth joint ($w = 5$). In addition, each of these blue-colored elements corresponds to the joint angle of the fifth joint in each of five node samples along the first path edge of the second path segment. As shown in the example path in Figure 4, the second path segment consists of two path edges and its first path edge has five node samples on it. In a case where there are multiple path edges within a path segment, the index in z-dimension is determined as follows: Let m_e be the number of path edges within a path segment and n_j be the number of node samples on the jth path edge where $j \in [1, 2, \ldots, m_e]$. Then the z index for the kth node sample of the jth path edge is

$$z = \sum_{i=1}^{j-1} n_i + k. \tag{4}$$

Once the target array indices are determined as described above, the node sample data generated by each thread is stored in their corresponding locations. In this work, the maximum number of node samples to be generated in one path segment is set to 1024.

C. PARALLEL COLLISION CHECKING

In this section, we explain how the proposed parallel collision checking process in line 5 of Algorithm 4 is implemented. This calculation process starts immediately after completion of the node sample generation process and the generated node samples data is stored in the global memory of GPU. For concurrent collision checking calculation, we first configure the thread hierarchy so that the blocks are configured in 3D form and the threads in each block are configured in 1D form. An example configuration of this thread hierarchy is shown in Figure 5 when a path is divided into 3 path segments, all joint combination cases are 63, and the maximum number of node samples on each path segment for collision checking is set to 1024. In this configuration, a block at (x, y, z') location consists of threads working on the xth path segment, the yth joint combination case, and node samples indexed as $256 \times (z' - 1) + l$ where $z' \in [1, 4]$ and $l = [1, 256]$. Note that the

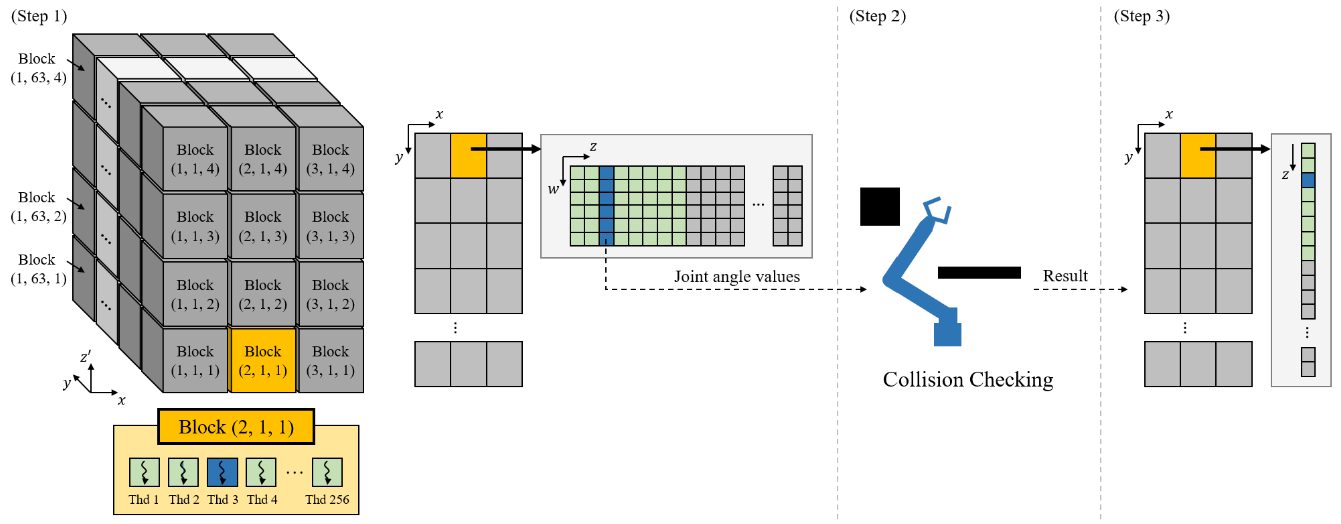


FIGURE 5. **Parallel collision checking process.**

upper bound for z', which is 4 in this example, is determined by the fact that we set the maximum number of node samples per path segment is set to 1024 and also the number of threads per block is set to 256 as recommended by NVIDIA.

In this thread hierarchy configuration, the lth thread in the (x, y, z')th block can now access the node sample data stored in the global memory to determine the robot manipulator's pose for collision checking in the following manner. Recall that the node sample data is stored as a 4D array in the global memory such that the element $a_{x,y,z,w}$ in the array contains the joint angle for the wth joint of the manipulator at the zth node sample in the xth path segment and the yth joint combination case. Thus the lth thread in the (x, y, z')th block can determine the pose of the manipulator by using angle values stored in elements $a_{x,y,z,w}$ for all $w \in [1, 6]$ where $z = \left(256 \times (z' - 1) + l\right)$. If we let $q^{l}_{x,y,z'}$ be the node sample processed by the lth thread in the (x, y, z')th block, then

$$q^{l}_{x,y,z'} = \left[\theta^{1}_{x,y,z}\ \ \theta^{2}_{x,y,z}\ \ \theta^{3}_{x,y,z}\ \ \theta^{4}_{x,y,z}\ \ \theta^{5}_{x,y,z}\ \ \theta^{6}_{x,y,z}\right]^{\mathrm{T}} \qquad (5)$$

where $\theta^{w}_{x,y,z}$ is the joint angle value stored in $a_{x,y,z,w}$ and z is determined as mentioned above based on z' and l. Therefore, collision checkings for all cases of path segment x, joint combination y, and node samples z on path segment can be processed in parallel based on the thread hierarchy configured in the GPU.

The last step of the parallel collision checking process is to store the collision checking results in global memory so that they can be used to determine whether a shortened path segment $\Pi'_{s,i}$ is free from collisions or not, which is done in line 8 of Algorithm 4. For this purpose, we use a 3D array variable to store the result for each node sample. Let $b_{x,y,z}$ be the element of the 3D array located at the position with (x, y, z) indices where x, y, z corresponds to the index number associated with each path segment, joint combination, and node sample, respectively. Then the lth thread in the (x, y, z')th block stores its collision checking result, which is 0 if no collision, 1 otherwise, at $b_{x,y,z}$ where z is determined by z' and l in the same way as before. Thus, each thread can store its calculation result without causing any synchronization issues.

V. PERFORMANCE EVALUATION

In this section, we present the simulation results of the proposed path shortening algorithm. To verify the effectiveness of the parallelization strategy, the performance was analyzed in terms of convergence rate and computation time.

A. SIMULATION SETUP

In order to evaluate the performance of the proposed parallelization framework for path shortening in Algorithms 3 and 4, we ran simulations with a 6-DOF robot manipulator that has 63 possible joint combinations. An approximated robot model was implemented based on the Oriented Bounding Boxes (OBB) tree approach for collision detection. The OBB tree shows relatively better performance than other collision detection algorithms such as AABB, sphere-tree, even when two objects are close to each other [35]. Robot Operating System (ROS) [36] was used as a software development platform to implement the proposed parallelization algorithm. The parallelization process of the ParaSC algorithm was implemented on an NVIDIA GeForce GTX 1080 Ti.

Figure 6 shows three different scenarios that we used in our simulations for path shortening where the goal configuration of the robot is represented in orange, the green-colored boxes represent obstacles around the manipulator, and the blue curve and red curve are the end-effector's trajectory along the path generated by a sampling-based path planner and its shortened path, respectively. For all scenarios, a path Π was generated using PRM which is one of the commonly used

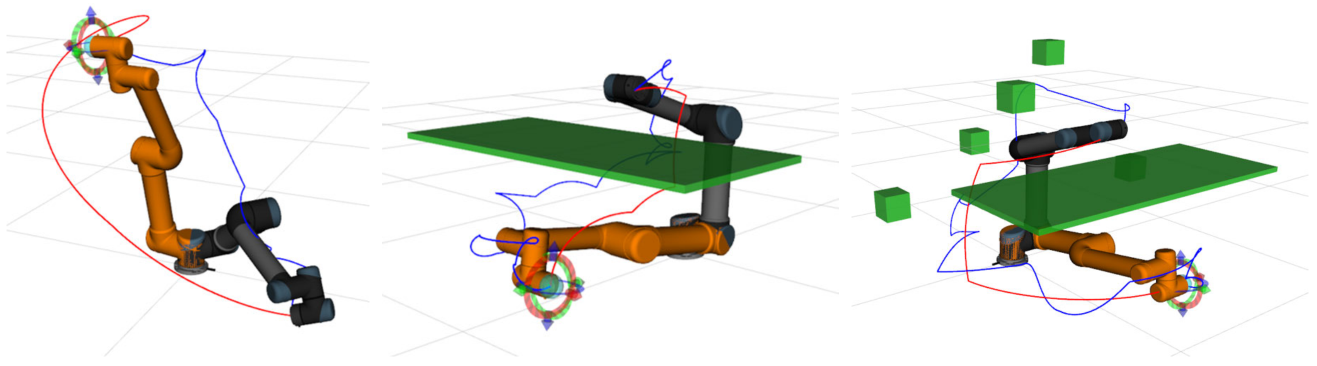


FIGURE 6. **Motion planning scenarios with a 6 DOFs UR-5 manipulator.**

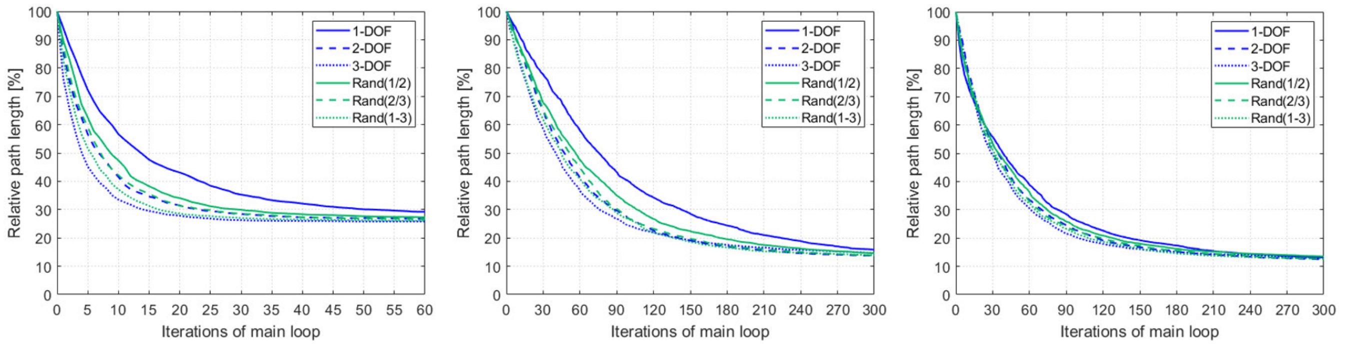


FIGURE 7. **Path length reduction rate of 6 variants of APSC algorithm.**

sampling-based path planning algorithms, and the number of path segments for path shortening was set to 3.

We evaluated the path shortening performance of the proposed ParaSC algorithm by comparing the results with APSC algorithm. Note that the results shown below is the average of the results obtained from 100 simulation runs for each scenario. We believe that these statistical results can show the difference in performance more accurately since, due to the randomness involved in APSC algorithm, the outcomes of APSC algorithm are generally different every time even if it is under the same condition such as the given planned path, environment, the number of iterations, the number of selected joints, etc. Thus, for a given planned path Π for each scenario, we ran the path shortening simulation 100 times for each algorithm and computed their average as the performance measure for each algorithm.

B. COMPARISON OF CONVERGENCE RATE

First of all, as it is necessary to pre-determine the number of joints to be selected in APSC algorithm, we ran simulations using only APSC algorithm with variations of the number of joints to be selected for path shortening and compared their performance to determine the best case for APSC algorithm. For this, we considered three different fixed number of joints to be selected which are denoted as 1-DOF, 2-DOF, and 3-DOF for one joint, two joints, and three joints to be selected respectively. Also, we let the algorithm can select the number of joints randomly within a given range such as [1, 2], [2, 3], and [1, 3] which are denoted as Rand(1/2), Rand(2/3), and Rand(1-3), respectively. As an example for this case, when the range [2, 3] is given, APSC algorithm selects 2-DOF or 3-DOF joints randomly out of 6-DOF at each iteration step as the algorithm proceeds.

Figure 7 shows the results that compare the rate of path length reduction at each iteration step as each algorithm proceeds with path shortening. In the figure, the relative path length represents the path length of the shortened path at the current iteration step compared to the path length of the original planned path Π. In Figure 7a, the convergence rate of APSC algorithm for the 3-DOF case is the fastest, followed by Rand (2/3), 2-DOF, Rand (1-3), Rand (1/2), and 1-DOF. The results for scenarios 2 and 3 in Figure 7b and 7b show similar results but had more iterations than scenario 1. This is because surrounding obstacles frequently hindered the generation of new paths, resulting in a higher chance of collision checking failure when shortening a path. An interesting observation from this result is that the performance difference among the variants of APSC algorithm became smaller as

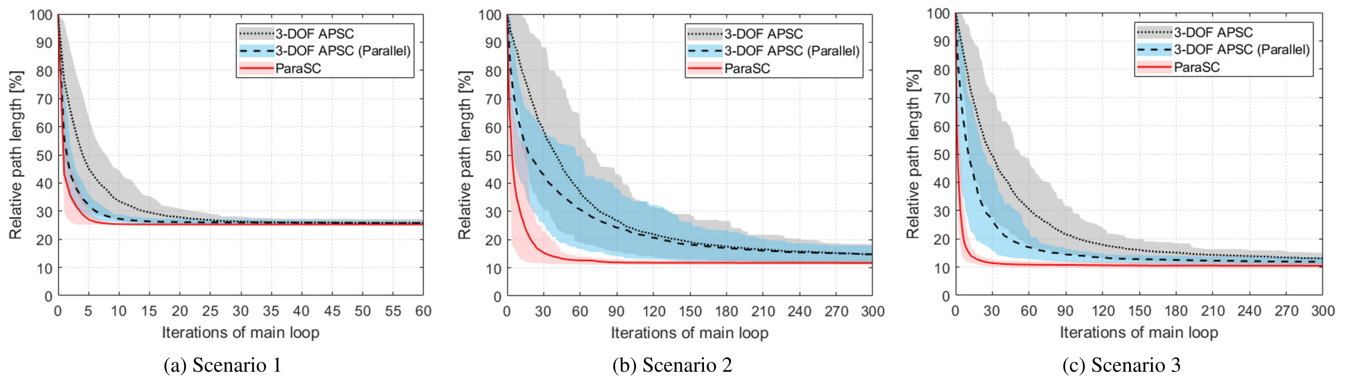


FIGURE 8. Comparison of path length reduction rate.

the environment for manipulator path planning became more complicated. This is because the benefit of using multiple joints for shortening a path segment in each iteration step is disappearing as it becomes more difficult to shorten a path in multiple DOFs joint space than in one DOF joint space due to surrounding obstacles. Therefore, as shown in Figure 7, the convergence curves of APSC algorithm with multiple DOFs joint selection approach the one DOF joint selection case, and hence the overall performance of APSC algorithm grows worse as the scenario becomes more challenging.

To evaluate the performance of the proposed ParaSC algorithm, we compared the path shortening results of the proposed algorithm against APSC algorithm for each scenario. For this, we used the conventional APSC with 3-DOF which has the fastest convergence rate for all scenarios. Also, we considered another variation of APSC algorithm for comparison to show the effects of two different parallelization strategies, as discussed in Section III-B, which are parallelizations for multiple path segments and all possible joint combinations. This APSC algorithm variant performs path shortening for multiple path segments simultaneously as in ParaSC algorithm, but each path segment is shortened by the conventional APSC algorithm.

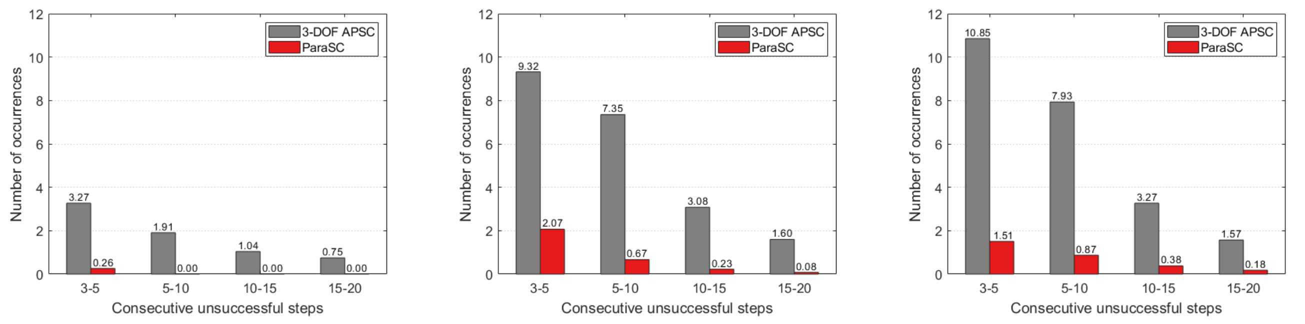
Figure 8 shows the results of convergence rate comparison for the three path shortening algorithms, where the line and translucent areas represent the average and a range between the 10th and 90th percentiles of the relative path length according to iteration, respectively. In this figure, 3-DOF APSC is the case where APSC algorithm was applied to only the middle path segment out of three path segments as in its conventional algorithm. On the other hand, 3-DOF APSC (Parallel) is the case of APSC algorithm with the parallelization strategy for multiple path segments as described above. As shown in the figure, the effect of simultaneous path shortening of multiple path segments is clearly noticeable because the relative path length reduction of 3-DOF APSC (Parallel) is much faster than 3-DOF APSC case in all three scenarios. In particular, the performance of 3-DOF APSC (Parallel) was similar to that of ParaSC in Scenario 1. From these results, we can confirm that the strategy of

TABLE 3. Comparison between ParaSC and 3-DOF APSC.

Path	Relative path length	Iterations		Ratio
		ParaSC	3-DOF APSC	
Scenario 1	50%	1	5	20.0%
	40%	2	7	28.6%
	30%	5	14	35.7%
	26%	8	60	13.4%
Scenario 2	50%	5	41	12.2%
	40%	7	55	12.7%
	30%	11	76	14.5%
	20%	20	141	14.2%
	15%	32	286	11.2%
Scenario 3	50%	2	30	13.4%
	40%	3	43	7.0%
	30%	4	61	6.6%
	20%	7	101	6.9%
	15%	14	184	7.6%

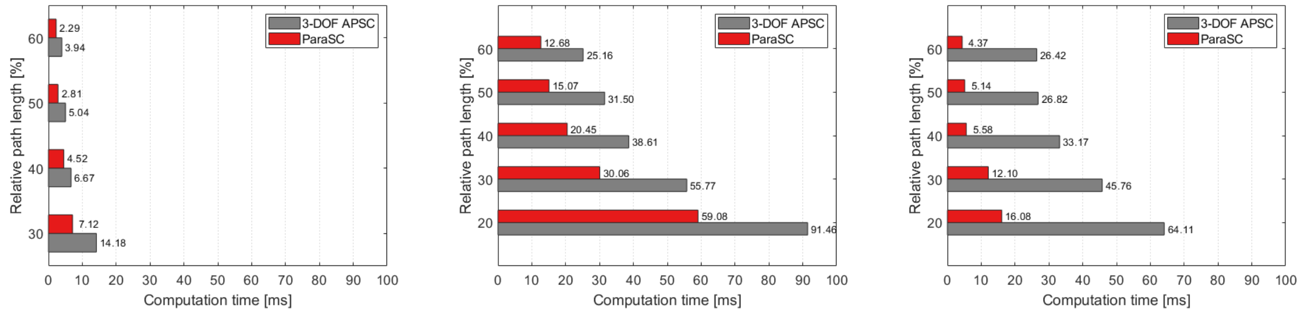
multiple path segments parallelization alone is certainly effective for faster path shortening process. Another important observation from these results is that the performance gap between ParaSC and the other two algorithms became larger in Scenarios 2 and 3 than the case of Scenario 1. Specifically, ParaSC showed a smaller deviation from the average convergence rate, and reached the lowest relative path length in fewer iterations than the others in most trials. This implies that the second parallelization scheme proposed in this paper, which considers all joint combinations simultaneously, is much more effective in situations when the environment for path planning becomes more complicated.

In Table 3, we specifically compared the performance between the proposed algorithm and the APSC algorithm with 3-DOF joint selection. As shown in the table, for the same quality of path improvement in terms of the relative path length, the number of loop iterations of ParaSC was significantly smaller than that of 3-DOF APSC in all cases. For example, to generate a shortened path with 30% relative path length, ParaSC took roughly 35.7%, 14.5%, 6.6% of the loop iterations needed by 3-DOF APSC in Scenarios 1, 2, and 3,



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

FIGURE 9. Number of consecutive non-successful steps before saturation.



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3

FIGURE 10. Comparison of execution time of ParaSC and 3-DOF APSC.

respectively. This clearly indicates that the proposed ParaSC parallelization framework can generate a well-shortened path with much fewer iterations and its path-shortening performance is substantially more robust to the environment complexity than APSC algorithm.

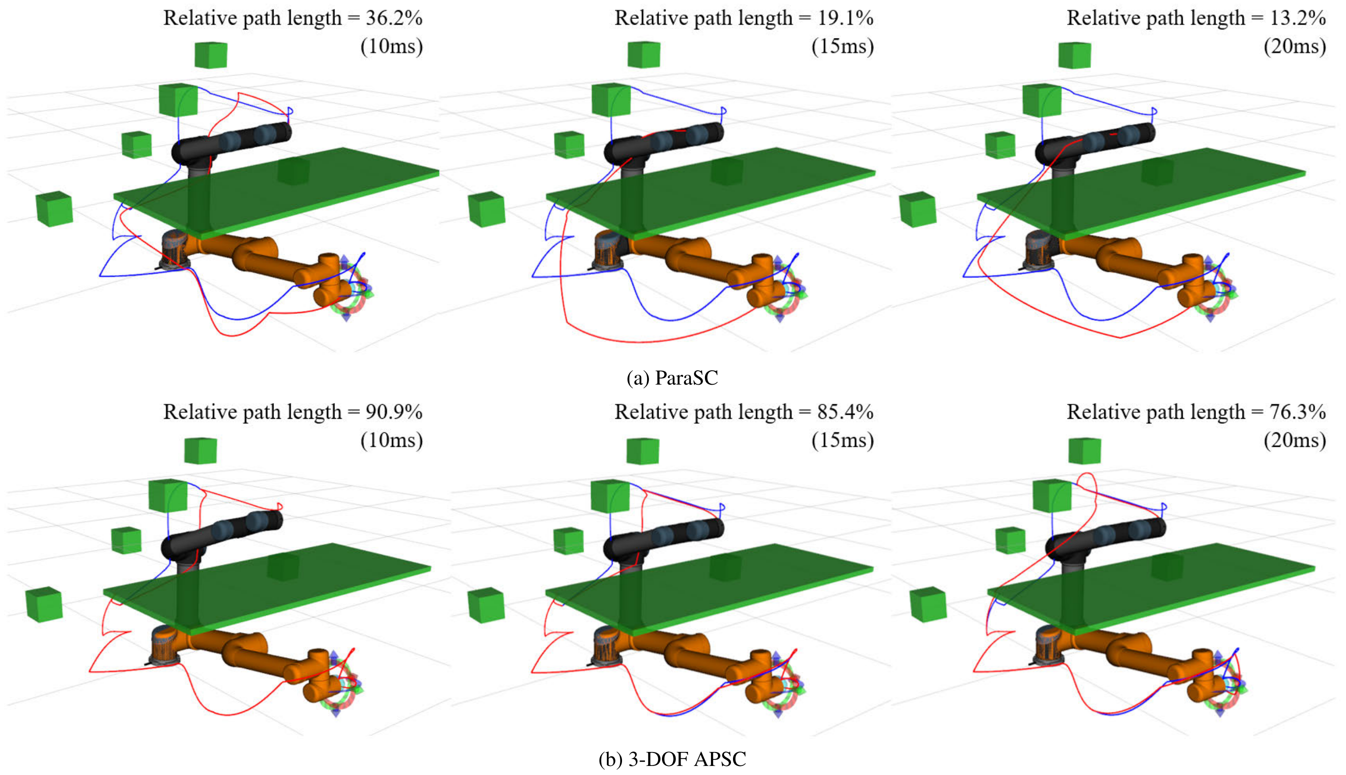
C. ANALYSIS OF THE IMPROVED CONVERGENCE RATE

Another way to evaluate the performance of the proposed parallelization algorithm for path-shortening is to count the number of consecutive iteration steps with an unsuccessful path segment update. An unsuccessful path segment update occurs when there are collisions, or when the selected path segment has already been shortened in the previous steps. Since the unsuccessful path segment update is most likely to occur due to collisions until the path shortening process converges, we measured the number of unsuccessful consecutive iteration steps until the algorithm converged. Figure 9 shows a comparison of ParaSC and 3-DOF APSC. The number of occurrences in the figure is the average number of occurrences from 100 simulation runs for each algorithm. As we can see from the figure, ParaSC algorithm has a much lower number of unsuccessful consecutive path segment update steps compared to APSC algorithm. This is one of the crucial reasons for the much faster convergence performance of the proposed algorithm.

D. COMPARISON OF COMPUTATION TIME

In Figure 8, we compared the path shortening performance of ParaSC and 3-DOF APSC based on the number of loop iterations. This comparison shows that ParaSC converges to a well-shortened path with much fewer loop iterations than 3-DOF APSC. Thus it is reasonable to conjecture that ParaSC takes much less "time" to shorten a path than 3-DOF APSC as well. To verify this conjecture, we implemented the parallelization parts of ParaSC algorithm on a GPU as described in Section IV, and compared the computation times of ParaSC with those of 3-DOF APSC for all three scenarios. As before, the results shown below are the average of 100 simulation runs for each scenario.

For comparison, the average execution time for each algorithm was measured for the same path length. All simulations were conducted on a computer with Intel Core i7-7700K CPU, 4.2GHz, 64GB RAM, and NVIDIA GTX 1080 Ti. We note that GPU memory allocation time was excluded from the execution time of ParaSC. This is because we implemented the parallelization parts of ParaSC on a GPU so that all parameters required for GPU memory allocation, such as the number of nodes constituting the given path, the number of path segments to be divided, and the number of joint combinations, can be determined before ParaSC starts to run on the GPU.



Relative path length = 36.2%
(10ms)
Relative path length = 19.1%
(15ms)
Relative path length = 13.2%
(20ms)
(a) ParaSC
Relative path length = 90.9%
(10ms)
Relative path length = 85.4%
(15ms)
Relative path length = 76.3%
(20ms)
(b) 3-DOF APSC

FIGURE 11. Comparison of end-effector path for ParaSC and 3-DOF APSC for the same computation time spent for path shortening in Scenario 3. (Blue line is the original path and red line is the shortened path at that moment.)

Figure 10 shows the time taken for each algorithm to shorten a given path. In Scenario 1, we can see that both algorithms can generate a well-shortened path in less than 15 milliseconds. Note that Scenario 1 is the case where there are no obstacles around the manipulator. Thus, in this case, the shortened path is most likely to be collision-free for any joint combination selected for path shortening except the case of self-collision. This is why the computation times for both algorithms are fairly small in this scenario. The result also shows the effectiveness of the parallelization in ParaSC since it took only less than half of the time taken by 3-DOF APSC for a well-shortened path even in this simple situation.

As one can anticipate from the results shown in Figure 8, the computation time difference between ParaSC and 3-DOF APSC was larger when there were obstacles around the manipulator. This result is shown in Figures 10b and 10c. In these situations, unlike in Scenario 1, it is more likely to have collisions, due to surrounding obstacles, when shortening a path segment via linear interpolation. Thus APSC algorithm was forced to spend more time repeating the steps selecting a path segment, shortening the path segment, and collision checking of the shortened path segment until it successfully found a path segment that could be shortened without incurring collisions. On the other hand, thanks to the parallelization for both multiple path segment processing and all joint combinations in ParaSC, it was substantially less likely to waste time during the path shortening process. This is why the computation time for ParaSC is substantially smaller than those of 3-DOF APSC in both scenarios, supporting the idea that the parallelization strategy proposed in ParaSC is much more effective for faster path shortening in practice when a manipulator is operated around obstacles.

To check whether the path shortening speed was improved using ParaSC, we conducted a statistical hypothesis test. For this, we stated the following null hypothesis (H_0) and alternative hypothesis (H_1) where μ_{t_1} and μ_{t_2} are the average execution time of 3-DOF APSC and ParaSC, respectively.

$$H_0 : \mu_{t_1} = \mu_{t_2}$$
$$H_1 : \mu_{t_1} > \mu_{t_2}$$

Both algorithms shortened the same initial path for each scenario, and the simulation was run 100 times for each relative path length. Thus, it can be assumed that the simulation data we obtained are large, independent samples. Table 4 shows the results of the hypothesis test at the 5% level of significance ($z_{0.05} \approx 1.65$ in this test). In this hypothesis test, H_0 were rejected for all relative path lengths in scenarios 1, 2, and 3. These results imply that ParaSC shortens the path faster than 3-DOF APSC.

Figure 11 shows the snapshot images that compare the quality of the path generated by ParaSC and 3-DOF APSC as the path shortening computation progressed for the case

TABLE 4. Hypothesis test results at the 5% level of significance for speed improvement verification by ParaSC. (P: ParaSC, A: 3-DOF APSC).

Scene	Rel. Path Len.	Alg.	# of trials	Mean [ms]	St. Dev.	Test Stats.	Reject H_0
1	60%	P	100	2.29	0.24	8.05	Yes
		A	100	3.94	2.03		
	50%	P	100	2.81	1.27	8.66	Yes
		A	100	5.04	2.23		
	40%	P	100	4.52	2.52	5.90	Yes
		A	100	6.67	2.64		
	30%	P	100	7.12	4.15	5.54	Yes
		A	100	14.18	12.05		
2	60%	P	100	12.68	7.02	8.32	Yes
		A	100	25.16	13.25		
	50%	P	100	15.07	13.17	7.81	Yes
		A	100	31.50	16.40		
	40%	P	100	20.45	17.90	7.08	Yes
		A	100	38.61	18.40		
	30%	P	100	30.06	20.53	5.26	Yes
		A	100	55.77	44.35		
	20%	P	100	59.08	32.56	4.32	Yes
		A	100	91.46	67.48		
3	60%	P	100	4.37	1.84	18.77	Yes
		A	100	26.42	11.60		
	50%	P	100	5.14	2.80	14.2	Yes
		A	100	26.82	15.01		
	40%	P	100	5.58	3.51	15.84	Yes
		A	100	33.17	17.07		
	30%	P	100	12.10	6.07	10.44	Yes
		A	100	45.76	31.69		
	20%	P	100	16.08	9.93	13.49	Yes
		A	100	64.11	34.18		

of Scenario 3. As one can see from this result, ParaSC was able to generate a well-shortened path in less than 20 milliseconds even in an environment with many surrounding obstacles, while the path generated by 3-DOF APSC after 20 milliseconds of path shortening computation was still not much different from the original one.

E. COMPARISON WITH OPTIMIZATION-BASED METHODS

In this section, the path shortening performance of ParaSC is compared with that of an optimization-based method. As discussed in Section I, there are various optimization-based path planners available as open-source, such as CHOMP, STOMP, and TrajOpt. In this work, we chose CHOMP and TrajOpt as representative optimization-based methods, which smooth the given path using functional optimization and sequential quadratic programming, respectively. Especially, since TrajOpt uses sequential convex optimization, we believe that it is relatively fast among optimization-based methods if an initial collision-free path is given. In this work, we used the TrajOpt and CHOMP algorithms implemented in MoveIt motion planning framework [37]. A new scenario was configured,

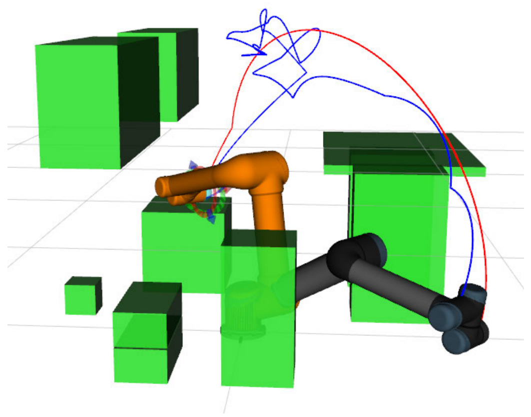


FIGURE 12. Scenario 4: A new scenario for comparison path shortening performance of ParaSC, 3-DOF APSC, CHOMP, and TrajOpt.

TABLE 5. Computation time comparison for ParaSC, 3-DOF APSC, TrajOpt, and CHOMP.

Rel. Path Len.	Algorithm	Mean [ms]	St. Dev.	Min [ms]	Max [ms]
50%	ParaSC	3.23	0.46	2.65	6.06
	3-DOF APSC	16.20	7.75	3.06	41.56
	TrajOpt	274.91	6.04	263.02	291.23
	CHOMP	7279.57	87.08	7133	7487
40%	ParaSC	3.62	1.41	2.53	8.44
	3-DOF APSC	18.58	8.51	3.50	42.18
	TrajOpt	278.85	6.38	266.89	296.24
	CHOMP	9390.48	96.72	9251	9688
30%	ParaSC	4.40	2.23	2.49	12.47
	3-DOF APSC	21.79	11.72	3.84	47.69
	TrajOpt	281.33	6.80	267.09	299.62
	CHOMP	13311.84	120.48	13040	13532
20%	ParaSC	6.48	4.07	2.53	17.76
	3-DOF APSC	30.18	13.04	7.89	69.56
	TrajOpt	282.71	7.09	269.75	302.81
	CHOMP	22199.96	216.08	21676	22639

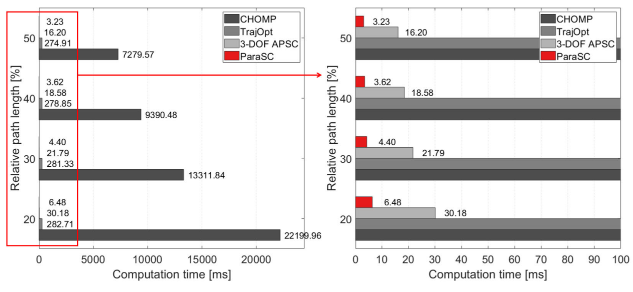


FIGURE 13. Computation time comparison between ParaSC, 3-DOF APSC, TrajOpt, and CHOMP.

as shown in Figure 12, in order to evaluate the performance of each algorithm in a more complex environment than other scenarios used in the previous sections.

Table 5 and Figure 13 are the comparisons of the computation time taken to perform path shortening for each

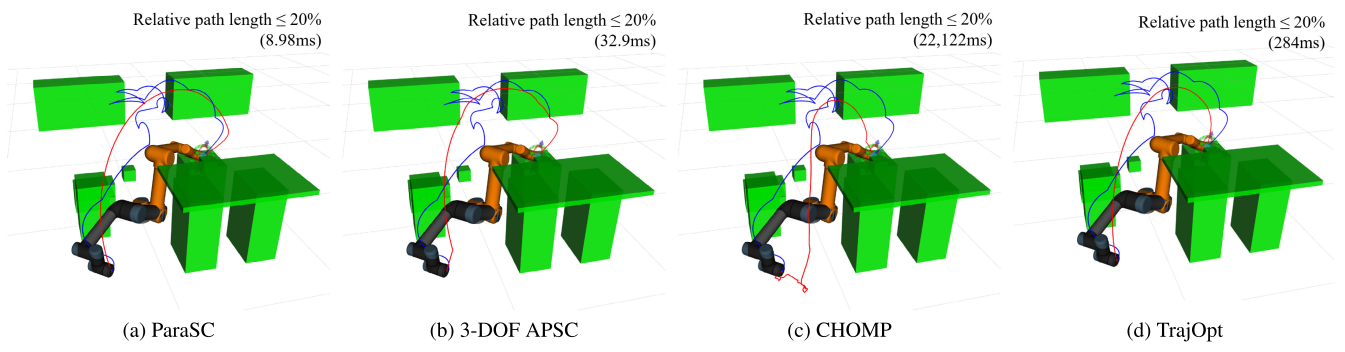

Relative path length ≤ 20%
(8.98ms)
Relative path length ≤ 20%
(32.9ms)
Relative path length ≤ 20%
(22,122ms)
Relative path length ≤ 20%
(284ms)
(a) ParaSC
(b) 3-DOF APSC
(c) CHOMP
(d) TrajOpt

FIGURE 14. Comparison of end-effector path and computation time for ParaSC, 3-DOF APSC, CHOMP, and TrajOpt in Scenario 4. (Blue line is the original path and red line is the shortened path.)

algorithm. Note that the results presented in the table and the figure are the averages of 100 simulation runs for each relative path length reduction. As shown in the figure, ParaSC shortened the initial path the fastest on average. Furthermore, comparing the min/max times in the table, no matter how fast CHOMP and TrajOpt performed path smoothing, they took longer than the maximum run time of 3-DOF APSC and ParaSC. In Figure 14, we also compared the path quality of each algorithm in terms of end-effector path after 20% relative path length shortening. As one can see from the figure, all of the algorithms successfully generated a collision-free shortened path. However, the result shows that the end-effector movement in CHOMP is less smooth than the other algorithms even if it took a substantially longer time to generate the shortened path. In the comparison between ParaSC and 3-DOF APSC, we can see that ParaSC can generate a shortened path much faster than 3-DOF APSC similar to the results in the other scenarios.

VI. CONCLUSION

In general, shortcut-based algorithms are effective for improving the quality of a path generated by sampling-based path planners. However, conventional shortcut-based path shortening techniques are not satisfactory in terms of computation time to be used in real-time path planning applications due to its high probability of unsuccessful path segment update caused by a random joint selection process. However, the parallelization framework proposed in this paper can maximize the efficiency of each step in shortcut-based approach for faster path shortening by finding the best joint combination using parallelism and by performing on several segments of a path at the same time. Simulation results showed that the proposed algorithm, Parallelized Shortcut (ParaSC), can generate a well-shortened path in a substantially shorter time than existing shortcut-based path shortening algorithms such as Adaptive Partial Shortcuts (APSC) through smaller loop iterations and GPU parallel computing. Also, from the results of performance comparisons with optimization-based path smoothing algorithms, Covariant Hamiltonian Optimization for Motion Planning (CHOMP) and Trajectory Optimization for Motion Planning (TrajOpt) in this work, it is demonstrated that ParaSC clearly outperformed CHOMP and TrajOpt in computation time while it can generate a well-shortened path with enough smoothness compared to the one generated by CHOMP in terms of end-effector movement.

In this paper, we showed that ParaSC is capable of generating a well-shortened path in less than a few tens of milliseconds in most cases of scenarios that we configured for simulations. Such computation time for path shortening is much faster than other algorithms, and is also fast enough to be used alongside human workers in practice. Based on these results, it is expected that ParaSC can be a useful algorithm to be used in part of a robot manipulator path planning frameworks for the purpose of collaborative robotics. Currently, it is still on-going work to integrate ParaSC with MoveIt motion planning framework so that it can be used in conjunction with various sampling-based path planning algorithms implemented in the framework. Also, we plan to implement an experimentation setup using an actual UR-5 robot manipulator to evaluate and demonstrate the performance of the proposed ParaSC algorithm in real-life collaborative robot operation tasks.

REFERENCES

[1] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Automat.*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[2] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Dept. Comput. Sci., Iowa State Univ., Ames, IA, USA, Tech. Rep. TR 98-11, 1998.

[3] P. Leven and S. Hutchinson, "A framework for real-time path planning in changing environments," *Int. J. Robot. Res.*, vol. 21, no. 12, pp. 999–1030, Dec. 2002.

[4] J. Bruce and M. Veloso, "Real-time randomized path planning for robot navigation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS)*, Lausanne, Switzerland, Jun. 2002, pp. 2383–2388.

[5] J. Pan, C. Lauterbach, and D. Manocha, "g-Planner: Real-time motion planning and global navigation using GPUs," in *Proc. AAAI Conf. Artif. Intell.*, vol. 24, no. 1, Atlanta, GA, USA, 2010, pp. 1245–1251.

[6] R. Guernane and N. Achour, "An algorithm for generating safe and execution-optimized paths," in *Proc. 5th Int. Conf. Autonomic Autonomous Syst.*, Valencia, Spain, Apr. 2009, pp. 16–21.

[7] H. Wang, H. Wang, J. H. Huang, B. Zhao, and L. Quan, "Smooth point-to-point trajectory planning for industrial robots with kinematical constraints based on high-order polynomial curve," *Mech. Mach. Theory*, vol. 139, pp. 284–293, Sep. 2019.

[8] J. Huang, X. Du, and L.-M. Zhu, "Real-time local smoothing for five-axis linear toolpath considering smoothing error constraints," *Int. J. Mach. Tools Manuf.*, vol. 124, pp. 67–79, Jan. 2018.

[9] B. Han and S. Liu, "RRT based obstacle avoidance path planning for 6-DOF manipulator," in *Proc. IEEE 9th Data Driven Control Learn. Syst. Conf. (DDCLS)*, Liuzhou, China, Nov. 2020, pp. 822–827.

[10] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, Kobe, Japan, May 2009, pp. 489–494.

[11] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "STOMP: Stochastic trajectory optimization for motion planning," in *Proc. IEEE Int. Conf. Robot. Autom.*, Shanghai, China, May 2011, pp. 4569–4574.

[12] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *Int. J. Robot. Res.*, vol. 33, no. 9, pp. 1251–1270, 2014.

[13] S. Berchtold and B. Glavina, "A scalable optimizer for automatically generated manipulator motions," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Munich, Germany, Sep. 1994, pp. 1796–1802.

[14] P. Isto, "Constructing probabilistic roadmaps with powerful local planning and path optimization," in *Proc. IEEE/RSJ Int. Conf. Intell. Robot. Syst.*, Lausanne, Switzerland, Sep. 2002, pp. 2323–2328.

[15] G. Sánchez and J.-C. Latombe, "A single-query bi-directional probabilistic roadmap planner with lazy collision checking," in *Robotics Research*. Berlin, Germany: Springer, 2003, pp. 403–417.

[16] P. S. Schmitt, W. Neubauer, W. Feiten, K. M. Wurm, G. V. Wichert, and W. Burgard, "Optimal, sampling-based manipulation planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, Singapore, May 2017, pp. 3426–3432.

[17] T. Zahroof, A. Bylard, H. Shageer, and M. Pavone, "Perception-constrained robot manipulator planning for satellite servicing," in *Proc. IEEE Aerosp. Conf.*, Big Sky, MT, USA, Mar. 2019, pp. 1–10.

[18] R. Guernane and M. Belhocine, "A smoothing strategy for PRM paths application to six-axes MOTOMAN SV3X manipulator," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Edmonton, AB, Canada, 2005, pp. 4155–4160.

[19] D. Hsu, J.-C. Latcombe, and S. Sorkin, "Placing a robot manipulator amid obstacles for optimized execution," in *Proc. IEEE Int. Symp. Assem. Task Planning (ISATP)*, Porto, Portugal, Jul. 1999, pp. 280–285.

[20] R. Luna, I. A. Sucan, M. Moll, and L. E. Kavraki, "Anytime solution optimization for sampling-based motion planning," in *Proc. IEEE Int. Conf. Robot. Autom.*, Karlsruhe, Germany, May 2013, pp. 5068–5074.

[21] K. Hauser and V. Ng-Thow-Hing, "Fast smoothing of manipulator trajectories using optimal bounded-acceleration shortcuts," in *Proc. IEEE Int. Conf. Robot. Autom.*, Anchorage, AK, USA, May 2010, pp. 2493–2498.

[22] R. Zhao and D. Sidobre, "Trajectory smoothing using jerk bounded shortcuts for service manipulator robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Hamburg, Germany, Sep. 2015, pp. 4929–4934.

[23] R. Geraerts and M. H. Overmars, "Clearance based path optimization for motion planning," in *Proc. IEEE Int. Conf. Robot. Autom.*, New Orleans, LA, USA, Apr. 2004, pp. 2386–2392.

[24] J. Polden, Z. Pan, N. Larkin, and S. van Duin, "Adaptive partial shortcuts: Path optimization for industrial robotics," *J. Intell. Robot. Syst.*, vol. 86, no. 1, pp. 35–47, Apr. 2017.

[25] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming productivity, performance, and energy consumption," in *Proc. Workshop Adap. Res. Manage. Sched. Cloud Comput. (ARMS-CC)*, Washington DC, USA, 2017, pp. 1–6.

[26] G. Ränger and T. Rauber, "General purpose GPU programming," in *Parallel Programming: For Multicore Cluster System*. 2nd ed. New York, NY, USA: Springer, 2013, pp. 404–406.

[27] X. Fei, K. Li, W. Yang, and K. Li, "CPU-GPU computing: Overview, optimization, and applications," in *Innovative Research and Applications in Next-Generation High Performance Computing*. Hershey, PA, USA: IGI Global, 2016, pp. 159–193.

[28] E. Wynters, "Parallel processing on NVIDIA graphics processing units using CUDA," *J. Comput. Sci. Colleges*, vol. 26, no. 3, pp. 58–66, Jan. 2011.

[29] Y. Munekawa, F. Ino, and K. Hagihara, "Design and implementation of the smith-waterman algorithm on the CUDA-compatible GPU," in *Proc. 8th IEEE Int. Conf. Bioinf. BioEngineering*, Athens, Greece, Oct. 2008, pp. 1–6.

[30] M. Humenberger, C. Zinner, M. Weber, W. Kubinger, and M. Vincze, "A fast stereo matching algorithm suitable for embedded real-time systems," *Comput. Vis. Image Understand.*, vol. 114, no. 11, pp. 1180–1202, 2010.

[31] C.-L. Hung and S.-W. Guo, "Fast parallel network packet filter system based on CUDA," *Int. J. Netw. Distr. Comput.*, vol. 2, no. 4, pp. 198–210, Oct. 2014.

[32] D. B. Kirk and W. H. Wen-Mei, "CUDA memories," in *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd ed. Waltham, MA, USA: Morgan Kaufmann, 2013, pp. 97–104.

[33] N. Wilt, "Memory," in *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. 1st ed. Upper Saddle River, NJ, USA: Pearson, 2013, pp. 156–157.

[34] J. Sanders and E. Kandrot, "Texture memory," in *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2010, pp. 115–137.

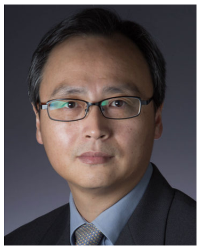
[35] C. Ericson, "Bounding volume hierarchies," in *Real-time Collision Detection*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2004, pp. 261–266.

[36] M. E. A. Quigley, "ROS: An open-source robot operating system," in *Proc. ICRA Open Source Softw. Workshop*, 2009, pp. 1–6.

[37] I. A. Sucan and S. Chitta. *Moveit*. Accessed: Jul. 12, 2021. [Online]. Available: https://moveit.ros.org/

JI HWAN SEO received the B.S. degree in transdisciplinary studies from Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu, South Korea, in 2019, where he is currently pursuing the Ph.D. degree with the Department of Information and Communication Engineering.

HYUNTAE LEE received the B.S. degree in physics from the School of Electrical and Electronics Engineering, Chung-Ang University, Seoul, South Korea, in 2018, and the M.S. degree from the Information and Communication Engineering Department, Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu, South Korea, in 2020. He researched real-time motion planning and robotic manipulator control.



KYOUNG-DAE KIM (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana–Champaign, USA, in 2011. He is currently an Assistant Professor with the Department of Information and Communication Engineering, DGIST. Prior to joining DGIST, he was an Assistant Professor with the Department of Electrical and Computer Engineering, University of Denver, USA. He was a Postdoctoral Research Associate with the Department of Electrical and Computer Engineering, Texas A&M University, USA. His research interests include developing theories, tools, and software frameworks to improve reliability and autonomy of cyber-physical systems, and their application to real systems, such as smart transportation systems, and collaborative robotic systems.