IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 16 September 2022, accepted 27 September 2022, date of publication 30 September 2022,
date of current version 10 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3211072

RESEARCH ARTICLE

Virtual LiDAR Simulation as a High Performance Computing Challenge: Toward HPC HELIOS++

ALBERTO M. ESMORÍS[1], MIGUEL YERMO[1], HANNAH WEISER[2], LUKAS WINIWARTER[2,3], BERNHARD HÖFLE[2,4], AND FRANCISCO F. RIVERA[1]

[1]Centro Singular de Investigación en Tecnoloxías Intelixentes, CiTIUS, University of Santiago de Compostela, 15782 Santiago de Compostela, Spain
[2]3DGeo Research Group, Institute of Geography, Heidelberg University, 69120 Heidelberg, Germany
[3]Integrated Remote Sensing Studio (IRSS), Faculty of Forestry, The University of British Columbia, Vancouver, BC V6T 1Z4, Canada
[4]Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, 69117 Heidelberg, Germany

Corresponding author: Alberto M. Esmorís (alberto.esmoris.pena@usc.es)

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG), German Research Foundation, in the frame of the projects SYSSIFOSS, under Grant 411263134; in part by VirtuaLearn3D under Grant 496418931; in part by the Ministerio de Ciencia e Innovación, Spain, under Project PID2019-104834GB-I00; and in part by the Consellería de Cultura, Educación e Ordenación Universitaria of Xunta de Galicia (accr. 2019–2022 and reference competitive group 2019–2021) under Grant ED431G 2019/04 and Grant ED431C 2018/19 and 2022/16.

ABSTRACT The software HELIOS++ simulates the laser scanning of a given virtual scene that can be composed of different spatial primitives and 3D meshes with distinct granularity. The high computational cost of this type of simulation software demands efficient computational solutions. Classical solutions based on GPU are not well suited when irregular geometries compose the scene combining different primitives and physics models because they lead to different computation branches. In this paper, we explore the usage of parallelization strategies based on static and dynamic workload balancing and heuristic optimization strategies to speed up the ray tracing process based on a k-dimensional tree (KDT). Using HELIOS++ as our case study, we analyze the performance of our algorithms on different parallel computers, including the CESGA FinisTerrae-II supercomputer. There is a significant performance boost in all cases, with the decrease in computation time ranging from 89.5% to 99.4%. Our results show that the proposed algorithms can boost the performance of any software that relies heavily on a KDT or a similar data structure, as well as those that spend most of the time computing with only a few synchronization barriers. Hence, the algorithms presented in this paper improve performance, whether computed on personal computers or supercomputers.

INDEX TERMS HELIOS++, HPC, KDTree, LiDAR simulation, parallel computing, ray tracing.

I. INTRODUCTION

HELIOS++ is an open source software written in C++11 that computes virtual point clouds by simulating LiDAR (Light Detection and Ranging) scanning [1]. For this purpose, a scanner is virtually mounted either on a static or a dynamic platform. The typical static platform is a tripod, and supported dynamic platforms are ground vehicles, aerial vehicles, or simple linear trajectories. The mounted virtual scanner is used to simulate the sensing of a given geometry by ray-tracing, thus generating an output 3D point cloud.

Scanning a virtual geometry defined by a set of primitives requires computing a huge number of ray intersection checks. Failing to handle this procedure appropriately will lead to a prohibitive computational cost. For this, a k-dimensional tree data structure (KDT) is used to speed up data access and computations [2]. The KDT is a type of binary tree that exploits the idea of binary space partition and can be used to speed up spatial queries. In doing so, a partition axis is chosen at each tree node to determine its best split position. Afterward, the set of objects is split into two children nodes. On the one hand, objects at the left of the split position belong to the left node. On the other hand, objects at the right of the split position belong to the right node. For instance, finding the nearest neighbor with brute force presents an average

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

complexity of $\mathcal{O}(n)$ while using a KDT leads to an average complexity of $\mathcal{O}(\log n)$.

Since 3D simulation software, such as HELIOS++, strongly depends on spatial queries to perform necessary ray tracing operations, it benefits from using an adequately built KDT. In fact, without a KDT or a similar data structure to speed up spatial computations, the simulation problems solved by HELIOS++ would be intractable. Therefore, improving the efficiency of the KDT should lead to a significant improvement in performance.

In this paper, we suggest combining two solutions to improve the efficiency of the KDT. The first is using heuristic strategies to reduce the computational cost of traversing the KDT. The second is the implementation of tree-building algorithms based on parallel computing techniques for both shared and distributed memory systems [3]. One typical third approach to the ray tracing problem is the implementation of GPU accelerated ray tracing algorithms. However, this case was not studied for two main reasons. The first one is that HELIOS++ is built as a very flexible simulator that supports different input geometries ranging from point clouds and detailed voxels (voxels with associated physical properties) to digital terrain models and classical Wavefront OBJ files. This input is translated to a different set of primitives with no significant constraints. Thus, it is possible to have primitives with very different sizes within the same scene and even typical plane primitives such as triangles combined with volumetric primitives such as voxels, with each subset of voxels performing a different physical calculus on ray intersection depending on its physical properties and operating mode. The aforementioned concerns make it difficult to implement a GPU-based ray tracing algorithm that works well for the general case because it is not trivial to properly balance the workload among warps of threads when the scene can present an extremely diverse composition. The second reason is that HELIOS++ is expected to run as a general-purpose tool on a typical CPU without requiring a GPU.

For the heuristic KDT building strategy, two greedy algorithms based on the Surface Area Heuristic (SAH) are proposed and parallelized. Both require a single user-specified argument defining the number of iterations to approximate the optimum split position. The algorithms themselves are well suited for the general case. When tested on different scenes with different sizes and types of primitives (voxels and triangles), they offered a significant improvement for all cases. When working with the new KDT implementations, the simulations showed low efficiency with a speedup of less than one when using a one task per thread strategy. Consequently, three alternative parallelization strategies are proposed and analyzed on shared memory computers. All of them are based on building chunks of tasks and distributing them among available threads, achieving shorter simulation times compared with the baseline implementation based on the one task per thread paradigm.

The proposed improvements lead to a significant boost in performance. Nonetheless, performance varies depending on the scene characteristics, the survey configuration, and the selected strategy. Some configurations can work well with big scenes but worse with smaller ones and vice versa. Specific survey settings, such as pulse frequency and leg characteristics, determine the computational burden of each simulation stage. Therefore, they also have a considerable impact on strategy selection.

In summary, we explore the application of well-known heuristics to optimize ray tracing algorithms based on advanced data structures from computer graphics and other domains. There is enough empirical evidence in the literature showing that these heuristics work well with regular meshes whose primitives are of the same type and solve physical models with uniform workload (e.g., standard lighting models), but this is not the case with HELIOS++ because it combines different input types with potentially different physical models. We also provide a theoretical justification for the heuristics' increased performance and application independence under mild assumptions on the distribution of rays. Besides, we explore the application of high-performance computing (HPC) techniques to our case study because they allow us to design load-balancing algorithms that work well for irregular and varying workloads.

The paper is organized into five sections after this introduction. First, we discuss the different strategies for KDT building in section II. Second, we present the parallelization of the KDT building process in section III. Then, we describe the parallelization of the simulation itself in section IV. In section V, we study the performance of the different strategies and algorithms, providing detailed results for all of our experiments. Afterward, in section VI we discuss under which conditions our algorithms are well suited for other related LiDAR simulators, further application scopes for our proposals outside laser scanning simulation, and the different works we are considering for the near future. Finally, we concisely present the main conclusions in section VII.

II. K-DIMENSIONAL TREE STRATEGIES

Standard KDT building procedures are based on taking a different axis at each depth and splitting it at the median point. More formally, let d be the current tree depth during the KDT building process and n the dimensionality of the space, so each point belongs to $\mathbb{R}^n$. The split axis ϕ is selected in a round-robin fashion, but then it can be expressed as a function of the tree's depth simply by $\phi \equiv d \mod n$. Let the set of objects (geometrical primitives) be $O = \{o_1, \ldots, o_m\}$. Besides, consider $a_i \in \mathbb{R}^n$ as the minimum vertex of the object's axis-aligned bounding box and $b_i \in \mathbb{R}^n$ as the maximum vertex. Thus, the split position is denoted as p_ϕ such that the left and right partition subsets of O can be respectively defined as $L = \{o_i : a_{i\phi} \leq p_\phi\}$ and $R = \{o_i : b_{i\phi} \geq p_\phi\}$. When p_ϕ is the median of the number of objects, the KDT building strategy is illustrated in Figure 1 and will be mentioned from now on as the Simple KDT.

Splitting at the object median point has been a widely used approach, as can be seen in Heckbert's work on color image

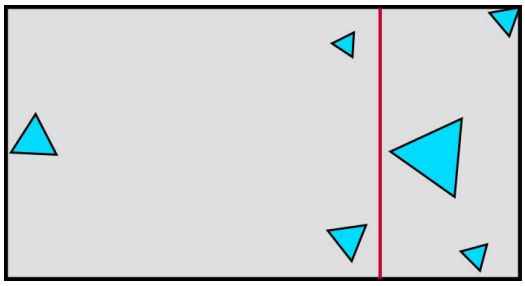


FIGURE 1. KDT splitting at object median with $r > 0.5$.

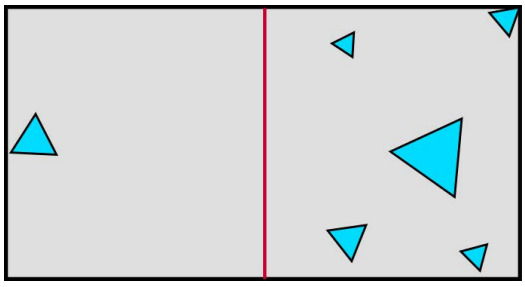


FIGURE 2. KDT splitting at spatial median with $r = 0.5$.

quantization, which proposes a median cut algorithm that creates a KDT as an alternative for the well-known former algorithm [4]. Alternatively, the spatial median can be used instead, leading to a split position as shown in Figure 2. The latter would be the case of Glassner's work [5], where the amount of ray intersection checks is reduced using an octree that divides the objects into compartments. The octree can be seen as a KDT using the spatial median as splitting criteria. The main difference would be that each octree partition splits into eight nodes while each KDT partition splits just into two. According to Samet, the KDT is an improved quadtree because it leads to a smaller branching factor. Thus, it can also be seen as an improved octree [6]. Therefore, it is easy to notice that the spatial median criteria of the octree can be efficiently applied to the KDT as it is significantly faster to compute than the object median. The reason is that the former implies computing the mean between min and max vertices, while the latter requires either sorting primitives or a histogram-based computation.

The reason why the spatial median criteria can lead to a similar efficiency as the object median is very intuitive if left and right partitions are similar for both cases. First, assume a uniform ray distribution [7]. Then, for the sake of simplicity, assume a unitary size bounding box too. Thus, the split position matches the space distribution ratio $0 \leq r \leq 1$. For a given number of rays k, the cost of ray intersection checks for the general case can be expressed as $k[r|L| + (1 - r)|R|]$. Ceteris paribus and ignoring the problem of straddling objects (those whose surface lies at both extremes of the split), it can be proven that both cases lead to the same cost. First, consider the spatial median case where $r = 1/2$, for which the cost is $k\,(|L| + |R|)\,/2$. Now, consider the object median scenario which cost is $k|L|$ because $|L| = |R|$, but then $k|L| = k\,(|L| + |L|)\,/2 = k\,(|L| + |R|)\,/2$ which is the same cost than for the spatial median case.

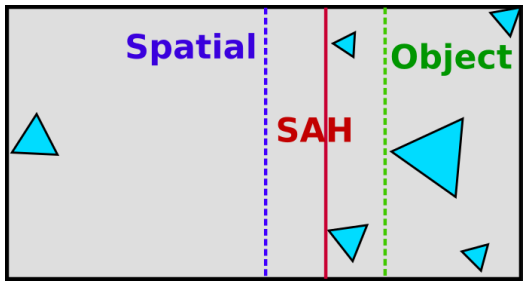


FIGURE 3. Surface area heuristic-based splitting in red color. Blue and green dashed lines represent the spatial and object median splits, respectively.

A. THE SURFACE AREA HEURISTIC

Although changing the object median based implementation by a spatial median one decreases KDT building time, it does not reduce the computational burden of ray intersection checks. A surface area heuristic (SAH) based approach is proposed to deal with this issue. The SAH is a heuristic based on the observation that, when using a KDT for ray tracing, most of the workload comes from tree usage, while building cost tends to be insignificant in comparison. The idea was introduced by MacDonald and Booth, who suggested governing the KDT construction by the costs of traversing the internal nodes and computing the intersection checks on leaf nodes [7].

SAH can be understood considering Figure 3 as an example. If the split position based on SAH is compared with the split position based on the spatial median, then the left and right partitions have the same number of primitives. However, the SAH-based split implies that the partition with fewer primitives occupies more space than it would if using the spatial median split criteria instead, as shown in Figure 2. Without loss of generality, assume $|L| < |R|$. If it is not possible to avoid $|L| = |R|$ because of primitives distribution, then there is no benefit from using a SAH-based approach. But this is an uncommon scenario. In consequence, proving that SAH split is better than both, object and spatial median splits, requires showing $k[r|L| + (1 - r)|R|] < k(|L| + |R|)/2$ from $|L| < |R|$ and $r > 1 - r$. It is easy to see that previous inequality is equivalent to $(r - 1/2)|L| < (r - 1/2)|R|$. Moreover, as $r > 1 - r$ then $r - 1/2 > 0$. Therefore, it is possible to divide both sides of the inequality by $r - 1/2$ without changing its sign, leading to $|L| < |R|$, which is already known to be true. Notice the same process could be applied for the case where $|L| > |R|$ which implies $r < (1 - r)$ and recall it was shown before that spatial median and object median criteria have equivalent costs, so the proof is completed.

The SAH implementation uses the cost function defined in (1), where C_i, C_l, and C_o are the costs of traversing an interior node, traversing a leaf node, and performing a ray-object intersection check, respectively. Also, N_i and N_l are the number of interior and leaf nodes, while $N_o(l)$ is the number of objects in the l-th leaf node. Finally, $S(i)$ is the surface area of the i-th interior node, $S(l)$ is the surface area

of the l-th leaf node, and $S(R)$ is the surface area of the root node.

$$C_T = \frac{1}{S(R)}\left[C_i\sum_{j=1}^{N_i}S(j) + C_l\sum_{j=1}^{N_l}S(j) + C_o\sum_{j=1}^{N_l}S(j)N_o(j)\right] \quad (1)$$

Let r be the normalized position of the splitting hyperplane for any node N. Therefore, $r = 0$ is the lower limit, $r = 1$ is the upper limit, and $r = \frac{1}{2}$ is the spatial median. Moreover, let L_r and R_r be the left and right parts for the r split position and $N_o(L_r)$ and $N_o(R_r)$ the number of objects at the left and right splits, respectively. In consequence, the loss function $\mathcal{L}(r)$ arises as in (2).

$$\mathcal{L}(r) = S(L_r)N_o(L_r) + S(R_r)N_o(R_r) - S(N)N_o(N) \quad (2)$$

Consider the term $-S(N)N_o(N)$ as the amount of work saved by making the node an interior one. Hence, it can be treated as a constant, and thus, the loss function can be simplified as shown in (3).

$$\begin{aligned}\mathcal{L}(r) &= S(L_r)N_o(L_r) + S(R_r)N_o(R_r)\\ &= rS(N)N_o(L_r) + (1-r)S(N)N_o(R_r)\\ &= S(N)\left[rN_o(L_r) + (1-r)N_o(R_r)\right]\end{aligned} \quad (3)$$

HELIOS++ implements the SAH as a greedy algorithm used at each node to evaluate the split decision based on local information. Thus, exploiting the fact that $S(N)$ is constant when working at the node level, a computationally simpler version of the loss function can be used, as introduced in (4).

$$\mathcal{L}_2(r) = rN_o(L_r) + (1-r)N_o(R_r) \quad (4)$$

Note that the loss function involves $N_o(L_r)$ and $N_o(R_r)$, which are both discontinuous functions of r. MacDonald and Booth proposed a smart analysis noticing that the number of objects in the left partition cannot decrease as r increases [7], because of that $\forall r \in [0, 1]$, $\frac{d}{dr}N_o(L_r) > 0$. Thus, considering $N_o(R_r) = N_o(N) - N_o(L_r)$, the loss function can be differentiated with respect to r as shown in (5).

$$\frac{d\mathcal{L}}{dr} = [2N_o(L_r) - N_o(N)]\frac{d}{dr}S(L_r) + [S(L_r) - S(R_r)]\frac{d}{dr}N_o(L_r) \quad (5)$$

First, assume that the object median lies somewhere satisfying $r < 1/2$. Therefore, at the left of the object median $\frac{d\mathcal{L}}{dr} < 0$ because $N_o(L_r) < N_o(N)/2$ and $S(L_r) < S(R_r)$. Also, at the right side of the spatial median $\frac{d\mathcal{L}}{dr} > 0$ because $N_o(L_r) > N_o(N)/2$ and $S(L_r) > S(R_r)$. In consequence, when the object median is to the left of the spatial median, the minimum must be somewhere between the object and spatial medians. Note that a similar argument applies for the case where the object median is to the right of the spatial median, which implies the optimum split position must lie between the object and spatial medians for any case.

Once the optimal split position is known to be inside $[\mu, \omega]$, where μ denotes the spatial median and ω the object median, it is possible to define an iterative process to approximate it accurately. Before describing the process, note that the optimal split position might lie inside $[\omega, \mu]$ instead. Nevertheless, this case will be omitted for simplicity, as it is very straightforward to figure out one from another since it is only necessary to swap start and end points.

Let a and b be the minimum and maximum vertices of the current KDT node, and let n be the number of iterations so $t \in [1, n]$. While $\mu \in [a, b]$ is always satisfied, it might happen that $\omega \notin [a, b]$ because primitives can be only partially contained in the node. In this case, force either $\omega = a$ or $\omega = b$ depending on which limit is exceeded. Afterwards, define the normalized spatial and object medians as $\hat{\mu} = (\mu - a)(b - a)^{-1}$ and $\hat{\omega} = (\omega - a)(b - a)^{-1}$. Also, let $\phi_1 = \hat{\mu}$ be the first tested split position in the iterative process. Considering the function shown in (6), it is possible to define the optimum split position approximation process as depicted in (7) so ϕ_n will be the best found split position, where n is the iteration index. Note that the unnormalized split position can be obtained as $\phi_n(b - a) + a$. Finally, notice that $r = \phi_n$ and thus, the greedy SAH algorithm approximates the optimum r.

$$\varphi(t) = \hat{\mu} + t\frac{\hat{\omega} - \hat{\mu}}{n - 1} \quad (6)$$

$$\phi_{t>1} = \begin{cases}\varphi(t), & \mathcal{L}_2\left[\varphi(t)\right] < \mathcal{L}_2(\phi_{t-1})\\ \phi_{t-1}, & \mathcal{L}_2\left[\varphi(t)\right] \geq \mathcal{L}_2(\phi_{t-1})\end{cases} \quad (7)$$

Here proposed greedy algorithm based on SAH requires one input argument n, the number of iterations to approximate the optimum split position. However, the problem of building an optimum SAH for the general case cannot be reduced to one simple argument, as more variables could be modified, such as the split axis. For instance, it is easy to see that an aerial scanning of a big mass of land would have more primitives along the ground plane, let us say the one defined by x and y axes, than in the orthonormal direction of the ground plane, let us say z axis. Hence, distributing more splits on the x and y axes than on the z axis could lead to a more efficient space partition in this particular case. Furthermore, expected scenes from LiDAR sensing can be of very different types. For instance, works using HELIOS++ (or its predecessor HELIOS for Java) range from industrial applications [8] to forestry applications [9]. Scenarios of real laser scanning range from mobile laser scanning (MLS) for autonomous driving [10] or indoor mapping [11], to large-scale airborne laser scanning [12]. Laser scanning can also be used in rural and urban environments such as [13] and [14] who used terrestrial laser scanning (TLS) and MLS, respectively, or even in industrial facilities [15].

Due to the various possible characteristics of different scenes, there is no evident constraint shared by different surveys that can be used to improve the performance of the greedy SAH algorithm in the general case. That is why the

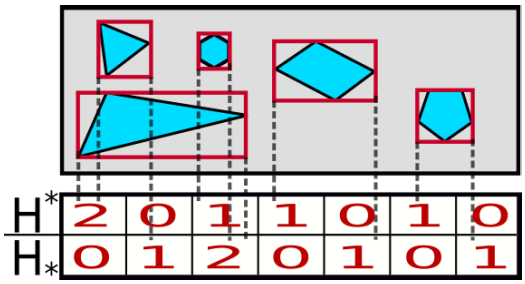


FIGURE 4. Min-max histograms population. Dotted lines connect the minimum and maximum vertices of each primitive with the histogram of minimums H_* and the histogram of maximums H^*, respectively.

number of nodes is left as an input argument. Because it is simple and flexible and allows the control of the compromise between precision and building time.

B. THE FAST SURFACE AREA HEURISTIC

While the SAH improves the efficiency of the simulation, it also implies a higher KDT building time. Therefore, in certain use cases, the increase in building time might overpass the decrease in simulation time coming from reducing the number of needed ray intersection checks. But even if the building time does not exceed the decrease in simulation time, it is still possible to reduce the total time by decreasing the building time even more. This is what the fast surface area heuristic (FSAH) does at the expense of precision.

A strategy based on min-max histograms similar to the one proposed in [16] is used to achieve the aforementioned goal. Let us start by defining H_* as the histogram of minimums and H^* as the histogram of maximums, with both having the same bin limits. If a_i is understood as the start point of the i-th bin and b_i as its end point, then the set of histogram intervals would be $\{[a_1, b_1), \ldots, [a_{m-1}, b_{m-1}), [a_m, b_m]\}$. Notice that the last interval is closed because histograms divide node space among partition axis in m bins, so the upper limit must be inside boundaries too. Once the histograms are built from a priori node information, the set of primitives is analyzed one time to populate both histograms as in Figure 4. Thus, the sorting stage required by the SAH implementation with a $\mathcal{O}(n \log n)$ computational complexity is substituted by a histogram-based implementation with $\mathcal{O}(n)$ complexity. Note that minimum and maximum vertices will belong to the minimum and maximum histograms, respectively. Thus, H_* can be seen as the set of minimum vertices while H^* would be the set of maximum vertices.

Once these histograms are known, the SAH approximation becomes a simple minimization problem. For this purpose, let $L(i) = |\{x \in H_* : x < b_i\}|$ be the number of primitives on left partition if split is placed on i-th bin and $R(i) = |\{x \in H^* : x \geq b_i\}|$ its analogous for right partition. Now, finding the approximated optimum split position r_{opt} can be done by solving (8). This computation has $\mathcal{O}(m)$ complexity. The FSAH offers good results for the general case when using $m = 32$, which is the same number of bins that Shevtsov *et al.* suggest using at any level [16]. There is also evidence that using 32 bins is a good choice when using a pigeonhole sorting algorithm based on histograms to approximate the optimum split position [17]. However, switching to exact SAH when the number of primitives is $N_o(N) \leq m$ did not bring any significant improvement for HELIOS++, unlike in the work of Shevtsov *et al.* [16]. Instead, it is a good choice to use a fixed minimum number of primitives such that a KDT node is only split when reached. Although considering a greater m leads to a more accurate search of the optimum split position, it significantly increases computational cost. Additionally, a fine grain search does not offer a proportional improvement in the cost of KDT queries, while the computational burden of the building does. That is why using a higher value such as $m > 100$ is strongly discouraged, at least for the general case.

$$r_{\text{opt}} = \arg\min_i \ \frac{i}{m} L(i) + \left(1 - \frac{i}{m}\right) R(i) \tag{8}$$

C. THE ILOT: INTERNAL LEAF OBJECT TOTAL CACHE STRATEGY

Both SAH and FSAH strategies benefit from a new cache technique called the Internal Leaf Object Total (ILOT) cache. Recall that SAH is based on the idea of governing the KDT construction through the cost function shown in (1). However, directly computing C_T at each split leads to an intractable problem because it requires traversing the partially built tree for each split decision. To address this problem, let us define t_0 as the KDT initial state composed only by the root node. But then, it is also possible to define a time-based version of the cost function in such a way that the initial cost is as shown in (9). Let us also define a speculative cost function like the one in (10) that computes the cost of the KDT after splitting the root node. The idea behind the speculative cost function is to compute the new cost by removing the cost of the current node as a leaf node and aggregating the cost of the current node as an interior node. In addition, the expected new cost considers the two children nodes as leaves. If $C_S(0) \geq C_T(0)$, then the root node will not be split because the cost would increase, and the KDT construction finishes. Otherwise, the KDT root node will be split.

$$C_T(0) = \frac{1}{S(R)} \left[C_l S(R) + C_o S(R) N_o(R) \right] \tag{9}$$

$$C_S(0) = \frac{1}{S(R)} \left[C_i S(R) + C_l \sum_{j=1}^{2} S(j) + C_o \sum_{j=1}^{2} S(j) N_o(j) \right] \tag{10}$$

Once the initial state is established, the evolution of the cost function can be modeled for any integer $t > 0$. For the sake of understanding, let us use the expressions defined in (11). Notice that the three can be computed at the node level, so there is no need to traverse the tree, and thus, the computational cost of the greedy algorithm is not significantly increased. Also, let the internal, leaf and object cached

costs for any $t > 0$ be $C_I(t) = C_i \sum_{j=1}^{N_i} S(j)$, $C_L(t) = C_l \sum_{j=1}^{N_l} S(j)$ and $C_O(t) = C_o \sum_{j=1}^{N_l} S(j)N_o(j)$. Hence, the speculative function to be applied after the initial state is the one shown in (12). If $C_S(t) < C_T(t-1)$ the node will be split and the new KDT cost will be $C_T(t) = C_S(t)$. In consequence, caches can be updated with no need of traversing the whole tree, so $C_I(t) = C_I(t-1) + k_1$, $C_L(t) = C_L(t_1) + k_2$ and $C_O(t) = C_O(t-1)+k_3$. Using an incremental implementation to minimize the computational burden of calculating the tree cost function was also proposed in other works [18].

$$\begin{aligned} k_1 &= C_i S_A(N) \\ k_2 &= C_l\,[S(L_r) + S(R_r)] - C_l S(N) \\ k_3 &= C_o\,[S(L_r)N_o(L_r) + S(R_r)N_o(R_r)] \\ &\quad - C_o S(N)N_o(N) \end{aligned} \tag{11}$$

$$\begin{aligned} C_S(t) = \frac{1}{S(R)}\Big[& C_I(t-1) + k_1 \\ & + C_L(t-1) + k_2 \\ & + C_O(t-1) + k_3 \Big] \end{aligned} \tag{12}$$

The ILOT cache is used together with the minimum number of required primitives to split. Both criteria must be satisfied for a KDT node to be split. Otherwise, the branch stops at the current node. First, the KDT cost function based on ILOT cache prevents splitting nodes that will increase tree cost, thus effectively governing KDT construction by (1), as suggested by MacDonald and Booth. Second, the constraint on the minimum number of primitives prevents splitting nodes that, despite decreasing KDT cost, increase tree depth and thus memory consumption without significantly improving performance. In such cases, the cost decrease is often relatively small compared to the decrease at lesser tree depth. Finally, a global representation of the ILOT cache logic is provided in Figure 5.

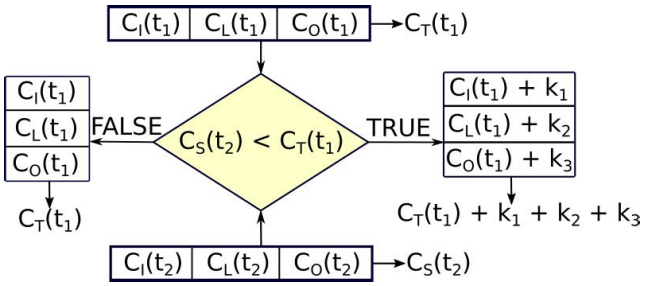


FIGURE 5. ILOT cache logic.

D. COMPLEXITY ANALYSIS

The computational complexity analysis of the different algorithms to govern the construction of the KDT is reduced to the complexity analysis of finding the split position. Any KDT building algorithm must iterate over primitives, instantiate nodes, and compute a few similar boilerplate operations. However, considering these operations does not help to compare our algorithms because all strategies share them. It makes more sense to study the complexity of the decision on whether to place the split position because it is the main difference between the different proposals.

For the object median split case, it is necessary to sort the primitives to find the exact object median. This sort process presents a computational complexity $\mathcal{O}(m\log(m))$ for m primitives. For the spatial median split case, it is enough to compute the mid-range. Consequently, it is necessary to iterate one time over the primitives to find the minimum and maximum vertices and then compute one addition and one division, leading to a complexity $\mathcal{O}(m+2)$. Since this complexity is dominated by the linear term m, it is a linear complexity $\mathcal{O}(m)$.

The SAH computation requires a sorting process like the object median with complexity $\mathcal{O}(m\log(m))$ and the calculus of the mid-range like the spatial median with complexity $\mathcal{O}(m)$. It also requires the computation of an iterative process of n iterations with linear complexity $\mathcal{O}(n)$. Thus, the final complexity to compute the SAH is $\mathcal{O}(m\log(m)+m+n)$. This complexity is dominated by the term $m\log(m)$. Neglecting the other terms leads to a simplified complexity of $\mathcal{O}(m\log(m))$. As stated in previous subsections, the FSAH strategy approximates the SAH with a reduced computational complexity because it uses histograms built in linear time instead of a sorting process. The complexity of the FSAH is $\mathcal{O}(2m+n)$, where $2m$ comes from computing two histograms in linear time each and n comes from the iterative process to approximate the optimal split position. Considering the complexity of the FSAH is dominated by the variable m (which is expected to be greater than n), neglecting the other terms leads to a simplified complexity of $\mathcal{O}(m)$.

Two main questions arise from the previous complexity analysis. The first is deciding between FSAH-based and spatial median split criteria, given that both have linear complexity. Recall that the cost of building the KDT is significantly smaller than the cost of the ray tracing process. The idea of using heuristics tries to exploit this fact to build a KDT that allows for faster ray tracing. If the complexity of building the KDT is the same, the FSAH-based heuristic must be preferred because it will reduce the cost of using the KDT for ray tracing purposes more than the spatial median. The second question is whether the reduced complexity of the FSAH approach comes at the expense of performance during ray tracing or not. We studied this second question thoroughly and found that for simulations with a large number of primitives, the FSAH approach preserves the performance of the SAH strategy while having a smaller building time. Different experiments were performed on personal computers and supercomputers and in different operating systems. They are explained in detail in section V.

III. PARALLEL K-DIMENSIONAL TREE BUILDING

KDT is a data structure with a vast set of possible applications. In consequence, several different parallelization proposals for its construction already exist. For instance,

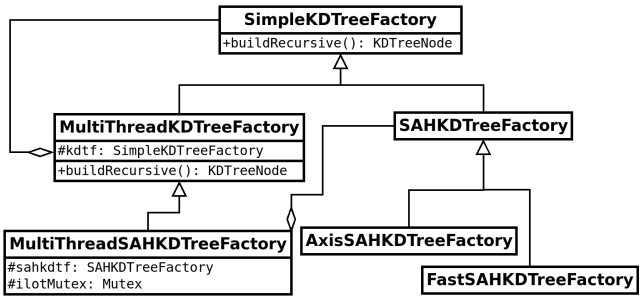


FIGURE 6. Parallel decorators for KDT factories.

a parallelization in two stages where a global binning is performed in the first stage to approximate medians quickly such that, in the second stage, it is possible to build a local KDT on each thread [16]. Concerning the streaming construction of KDT, it has been proposed to build a subtree in each different processor. This method exploits the fact that multiprocessor contexts have a separated cache for each processor, leading to fewer memory conflicts [18]. Other methods propose to apply geometry-level parallelism on upper nodes, and node-level parallelism on lower nodes because upper nodes contain a significant number of primitives, while lower nodes contain a smaller number. Alternatively, it is possible to use an in-place algorithm to minimize data movement by tracking at which tree depth primitives belong to [19]. There are also works using GPU versions of radix sort specifically designed to distribute data among threads. This approach prevents the typical unbalancing coming from large data chunks in upper tree levels and small data chunks in lower tree levels [20]. Other proposals focus on hybrid parallelization algorithms which can scale up to 50,000 cores building a large KDT through global redistribution of points among tree nodes. Then, data-level parallelism is exploited in each of these nodes until there are enough branches (generally at least ten times the number of threads) to proceed with thread-level parallelism, where each thread creates its local KDT from non-overlapping sets of points [21].

A. RECURSIVE PARALLELISM

In this work, a design inspired by the factory pattern is used to handle the building of different KDTs. Thus, the Simple KDT has its own factory while the SAH and FSAH each have their own factory too. A solution based on a balance between software design and HPC is proposed. Thus, only a single layer of parallelism is needed for the recursive building of the KDT. It is known from previous literature that this strategy might be improved by applying a geometry-level parallelization on the upper nodes. Instead, only a node-level parallelization is applied so it can be implemented through a simple decorator pattern, as shown in Figure 6

The SAH KDT extends the Simple KDT and replaces the median criteria with the SAH criteria for determining split position. The FSAH KDT implementation proposed in this work extends the SAH KDT and replaces the SAH computation with a faster histogram-based approximation. Axis SAH is an experimental implementation to analyze the impact of combining the SAH strategy with a better partition axis selection method. Each implementation that extends from a KDT factory also overwrites some methods invoked by the build recursive function. They are not detailed in Figure 6 for simplicity. The multi-thread KDT factory wraps the recursive building function of the decorated KDT factory to support parallelism. First, the main thread builds the first node. Then, the building of both splits is posted to a thread pool in a non-blocking way. If there are available threads, they will handle posted tasks. Otherwise, the caller thread continues the non-blocking recursively building of the KDT. Each thread replicates this behavior until the building process has concluded. If a thread finishes its job before the KDT has been built, it becomes available again so other threads can delegate their work to it. This building process is illustrated in Figure 7, where the blue color represents the main thread, and green and red colors the two secondary threads. The execution order is not deterministic, so only one possible execution is represented in the figure. To conclude, a hard constraint of 32 primitives is imposed such that in those cases where the current node has a smaller number of primitives, it will assume the whole workload. This approach is due to the overhead of thread context operations that might easily exceed the benefits of distributing workload at such a fine grain level.

B. PARALLEL ILOT CACHE

Using a multi-thread SAH KDT factory implies using a multi-thread KDT factory but extending the parallelization mechanism to handle concurrent access to the ILOT cache. The pseudo-code of concurrent access logic is shown in Algorithm 1, where N denotes the node being built, and P is the set of primitives. Besides, τ is the minimum number of primitives required to split, and M is the mutex to rule concurrent access to ILOT, the latter being used through init and speculate functions. Also, N_L denotes the left partition of node N, while N_R denotes its right partition. Analogously, P_L is the subset of primitives for the left partition, while P_R is the subset for the right partition. Moreover, C_T and C_S are the total tree and speculative costs as defined in (1) and (12), respectively.

The ILOT is initialized at the first KDT level. At this step, there is no need to handle concurrent accesses to neither ILOT nor costs because only the main thread is accessing them. After this, all threads share the same behavior, whether they are a secondary thread or the main one. If there are not enough primitives to split, then the node is considered a leaf, and no further branching is required. Otherwise, it is necessary to lock access to ILOT and costs to decide whether the split reduces the KDT cost or not. In the first case, C_T is updated with the speculated cost, and the mutex is released. Later, the recursive building of left and right nodes is posted to the thread pool, if necessary. When there are enough available threads, the entire workload is delegated. Otherwise, either only the left split will be delegated or none at all. In the

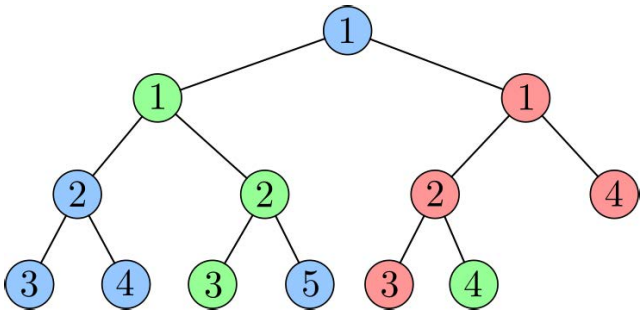


FIGURE 7. Example of parallel recursive building of a KDT.

Algorithm 1 KDT Building With Concurrent ILOT Access

```
function buildRecursive(N, P)
    if N is parent then
        C_T ← initILOT(N)
    end if
    if |P| ≥ τ then
        lock(M)
        C_S ← speculateILOT(N)
        if C_S < C_T then
            C_T ← C_S
            unlock(M)
            N_L ←buildRecursive(N_L, P_L)
            N_R ←buildRecursive(N_R, P_R)
        else
            unlock(M)
            makeLeaf(N)
        end if
    else
        makeLeaf(N)
    end if
end function
```

last case, the current thread will handle the building of both partitions.

IV. PARALLELIZATION OF THE SIMULATION

Three strategies are proposed based on distributing chunks of tasks among the thread pool. They lead to a positive speedup using new KDT heuristics and improved performance using the Simple KDT. All the proposed workload distribution strategies can be run in shared memory and distributed shared memory contexts. Currently, HELIOS++ is not supporting multiprocessor architectures with pure distributed memory. In consequence, all the strategies described in this section have been tested and implemented in shared memory contexts.

First, the one task per thread baseline strategy is described. Second, a static chunk size strategy is proposed. It is based on dividing the workload into chunks with a fixed number of tasks each. Then, a dynamic chunk size strategy is presented. It divides the workload into chunks whose size varies during simulation depending on measured idle times. Last, a warehouse of chunks that allows secondary threads to get their work from a warehouse-like data structure is explained. In consequence, the main thread is more decoupled to auxiliary threads. It can focus on fulfilling the warehouse, and once it is full, it can compute some workload on its own. The improvement of each strategy was measured considering the simulation times of SCALED_COG and FIXED_250 scenes described in Table 1 and Table 2 at Section V.

A. ONE TASK PER THREAD

The baseline parallelization is a naive one task per thread algorithm. Initially, it builds a pool of threads either with a user-given number of threads or with as many threads as available in the system by default. Then, once the simulation starts, the main thread posts each pulse computation task to the thread pool, where the task is assigned to a single thread. The main thread itself does not compute any task.

The first drawback of this strategy is the significant overhead of posting tasks to the thread pool one by one. While the number of tasks per simulation stage can vary, even in orders of magnitude, depending on the scanner configuration and trajectory length, they can easily reach 10^6 for a single simulation stage. This fact implies spending a significant amount of time in communications between the main thread and the worker threads. Parallelization strategies that reduce these communications can increase the time each thread spends computing. Another major inconvenience is that one thread is always either managing the posting of tasks to the thread pool or waiting for them to be completed.

This parallelization strategy is adequate for problems with only a few tasks, ideally with as many tasks as the number of threads. Besides, the workload for each task should be as uniform as possible. So the one task per thread proposal should only be considered for either testing purposes or very concrete use cases.

For the sake of understanding, assume the example problem of computing an affine transformation over a set P of m points, such that $\forall \mathbf{p} \in P, \mathbf{p} \in \mathbb{R}^4$. Suppose that there are 4 available threads. As the affine transformation is known to be applied to points in $\mathbb{R}^4$, it can be defined as $\mathbf{p}' = \mathbf{c} + A\mathbf{p}$ where A is a 4×4 matrix and $\mathbf{c}$ is a column vector as a 4×1 matrix. Thus, the entire work can be divided into m tasks having $m/4$ different points each. The one task per thread paradigm fits this example because it satisfies the two constraints. First, it can be divided into as many tasks as threads. Second, all tasks have the same workload since they all require the execution of one matrix multiplication and one vector addition for $m/4$ input points.

B. A STATIC CHUNK SIZE STRATEGY

Using a static chunk size to define workload distribution is one of the most simple yet effective techniques, with low overhead. It divides the set of tasks into subsets of approximately the same given size. This approach is easy to understand, debug, and manipulate. The main cons are that it is necessary to provide a good chunk size to maximize its efficiency and also that, when the task itself presents a

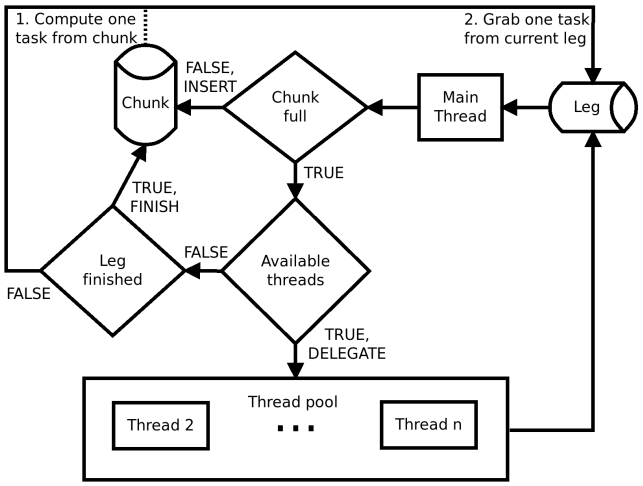


FIGURE 8. Static chunk size logic.

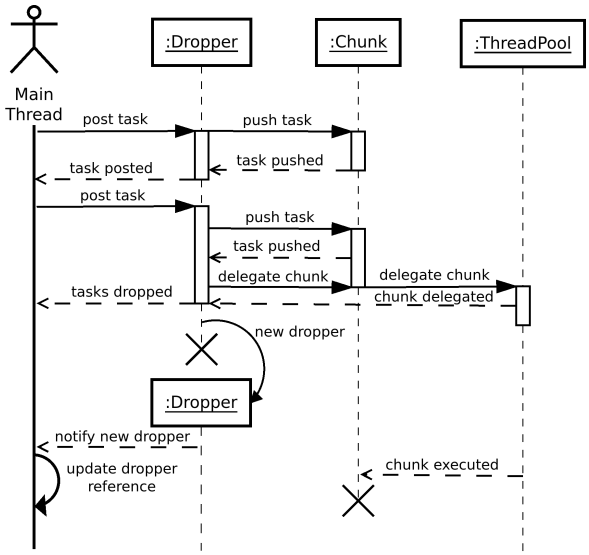


FIGURE 9. Static chunk size sequence diagram.

varying workload, it cannot adapt. In consequence, for the typical HELIOS++ use case, it might be efficient at certain time intervals during the execution but inefficient at others.

The static chunk size strategy is implemented in a non-blocking scheme. The main thread starts by posting chunks into the thread pool until all threads have work to do. When the main thread tries to post a chunk, but all other threads are already working, it computes one task from the chunk. Afterward, the chunk is replenished if there are more tasks to be computed. The aforementioned process is repeated up to the end when no chunks remain in the pool. If there are no more tasks for the current simulation stage, the main thread computes the chunk and waits for other threads to finish. Thread synchronization occurs at the end of each leg before proceeding to the next. The workflow of this proposal is depicted in Figure 8.

In order to implement this technique, a task dropper class is introduced. It is based mainly on two attributes, the maximum number of tasks m, and the set of tasks T. The task dropper imitates a typical dropper to handle chunk manipulation. A dropper is submerged into a liquid, and when the bulb is relaxed, the solution is drawn up inside a tube which then can be released in a different place. Analogously, the main thread uses the task dropper to ''submerge'' into the leg and draws up a piece of the remaining work. Afterward, this work is dropped into the thread pool, as explained before. More specifically, releasing the chunk of tasks into the thread pool occurs when the task dropper notifies the main thread that $|T| = m$. This process is illustrated in the sequence diagram of Figure 9. Once a task dropper has dropped its workload on the thread pool, it creates a new instance from itself, and then it is deleted while the main thread's dropper reference is updated. Task dropper class is later extended to exploit its instance renewal mechanism to provide adaptive behavior when using dynamic chunk size.

This parallelization strategy suits well problems that present a uniform workload during the entire execution and where an adequate chunk size can be a priori determined. Nonetheless, the problem must also require a significant number of tasks. Otherwise, the one task per thread strategy should be preferred.

One example of use case would be the computation of a filter (for instance, a typical Gaussian smoothing) over a set of m images $I = \{I_1, \ldots, I_m\}$ for which $\forall i, I_i \in \mathbb{R}^{h \times w}$. In other words, any image is a matrix of $h \times w$ pixels.

When m is much greater than the number of threads, it is justified to prefer a chunk size strategy over one task per thread since a large number of tasks implies a significant communication overhead. The number of communication operations, y, scales linearly with the number of tasks having a unitary slope, so $y = m$. This expression can be seen as a particular case of $y = k^{-1}m$ when $k = 1$, where k represents the chunk size (the number of tasks per chunk). Considering that k must be a positive integer, it is known that $k \geq 1 \iff k^{-1} \leq 1$ and thus, the greater k is, the smaller k^{-1} is, hence the smaller the slope of the linear relationship and the less the number of communication operations.

The workload will be constant during the entire execution because all input images are an array of $h \times w$ real scalars. An adequate chunk size k must be determined to apply the static strategy efficiently. Note that the greater k, the smaller the number of communication operations. Despite this, having $k = m$ would lead to a case with no parallelism because one single thread will compute the entire chunk of tasks. But having k equal to m divided by the number of threads would lead to the minimum number of communication operations, with each thread computing a similar workload. Note that for more practical use cases, having k equal to m divided by a small multiple of the number of threads would be enough.

C. A DYNAMIC CHUNK SIZE STRATEGY

Load balance is one of the main performance issues of the previous approaches. More specifically, two of the main

drawbacks of the static chunk size strategy can be addressed using a dynamic chunk size approach. First, using a reasonable starting size is enough to allow the algorithm to increase or decrease this size during execution to adapt to the workload. Hence, there is no need for the user to successfully speculate the best chunk size beforehand, as in the static case. Almost any initial size will lead to a good performance because of the adaptive nature of the algorithm. Note that the only problem is to specify very low initial values or very high ones ($\geq 10^5$). In such cases, the initial behavior might have stiffness problems and fail or take too long to adapt to the workload. Second, the dynamic chunk size approach can adapt to varying workloads during execution. Note that this situation is likely to happen in many practical cases.

The dynamic chunk size strategy needs an accurate and fast way to measure performance without a significant negative impact on simulation throughput. For this purpose, a timestamp t_0 is registered as soon as the first thread in the thread pool becomes idle. Later, a second timestamp t_1 is registered when the main thread fails to post to the thread pool because it is fully occupied. Thus, an elapsed idle time $t_b = t_1 - t_0$ is defined. Let us introduce the dynamic task dropper that can use idle time measurements to adapt to the current workload. Note that it extends the static task dropper by considering new attributes such as a minimum magnitude ϵ, a significance threshold τ, the initial additive transform magnitude Δ_0, the current additive transform magnitude Δ_1, the update step for additive transform magnitude Δ_2, and the sign s (i.e., either -1 or 1) of the additive transform. Note that both Δ_0 and Δ_2 are the same for any descendant of the first dynamic task dropper, no matter what is currently happening.

From now on, let B_a be the parent dynamic task dropper and B_b be the child dynamic task dropper. Also, assume that the prime variables correspond to the child dropper B_b and the other ones to the parent dropper B_a. Then, let us define the reproduction function r as in (14). Where the definition of the δ function is shown in (14). A task dropper governed by these equations increases its additive transform magnitude Δ_1 at each successive reproduction that preserves the sign, which can be understood as the evolutionary sense of the algorithm. In the case of a change of sign, it starts again from the original additive transform magnitude Δ_0, but with the opposite sign. It is worth mentioning that the implementation has a hard constraint. Thus, if the logic leads to $m < 1$, it will be forced to be $m = 1$. In other words, no sub-unit chunk sizes are allowed. Using high values of ϵ and τ resulted in an undesired loss of information because acceptable idle times were discarded. Moreover, using small values leads to very sensitive criteria which could not filter noisy measurements. Values of 0.1 milliseconds were appropriate for the different studied cases. Notwithstanding, these thresholds are significantly dependent on the application and computer traits. Hence, old and future computers with substantially better components and clocks than nowadays might need to update these thresholds to decrease or increase one order of magnitude to 1 and 0.01 milliseconds, respectively.

$$r(B_a, s') = \begin{bmatrix} m' \\ \Delta_1' \\ s' \end{bmatrix} \tag{13}$$

$$\begin{cases} m' = \max\left\{1, m + s'\delta\left(\Delta_1, s, s'\right)\right\} \\ \Delta_1' = \delta\left(\Delta_1, s, s'\right) \end{cases}$$

$$\delta\left(\Delta_1, s, s'\right) = \begin{cases} \Delta_1 + \Delta_2, & s = s' \\ \Delta_0, & s \neq s' \end{cases} \tag{14}$$

It is possible to define a child dropper B_b from its parent B_a as depicted in (15). This expression can be understood as an evolution-like adaptive algorithm because the child is expected to adapt better than the parent to the work context based on the information transmitted during evolution. In this equation, the "approximately equal to" is used because B_b is not exactly B_a when $t_b < \tau$. All attributes match the parent ones, except the task set T that initially is the empty set. Therefore the children will never inherit tasks from the parent. In other words, any set of tasks will be computed once and only by a unique task dropper. Besides, each consecutive task dropper should adapt better to the current workload through the reproduction function $r(B_a, s')$. Thus, satisfying the condition $t_b < \tau$ means that the idle time is too small to be considered, while satisfying $t_b \geq \tau$ means the idle time brings useful information. Also, satisfying $|t_a - t_b| > \epsilon$ means the difference between parent and child is significant enough to change the adaptive sense. Finally, a situation in which $|t_a - t_b| \leq \epsilon$ means the difference is not big enough to motivate a change of sign.

$$B_b \approx \begin{cases} B_a, & t_b < \tau \\ r\left(B_a, \operatorname{sgn}(t_a - t_b)s\right), & t_b \geq \tau \wedge |t_a - t_b| > \epsilon \\ r\left(B_a, s\right), & t_b \geq \tau \wedge |t_a - t_b| \leq \epsilon \end{cases} \tag{15}$$

An example of the behavior of the dynamic chunk size algorithm in HELIOS++ is shown in Figure 10. In this plot, the evolution of chunk size m is measured for a forestry simulation [9]. On the one hand, the static method maintains the same chunk size during the entire simulation. On the other hand, the dynamic approach varies its chunk size depending on the workload. Adapting the chunk size also updates the workload distribution to fit different simulation stages. Furthermore, the first half of the simulation benefits from a smaller chunk size, while the second half was more efficiently handled with a greater one. Looking at computation times in Table 3, the static chunk size leads to an execution time of 89 seconds, and the dynamic algorithm leads to an execution time of 80 seconds. It means that, for this particular case, there is a speedup of around 12% with respect to the static method when using the dynamic one.

The typical use case for dynamic chunk size also requires that the problem presents a significant number of tasks like the static chunk size. Although it differs from the latter in two points, it is not necessary to determine an adequate chunk

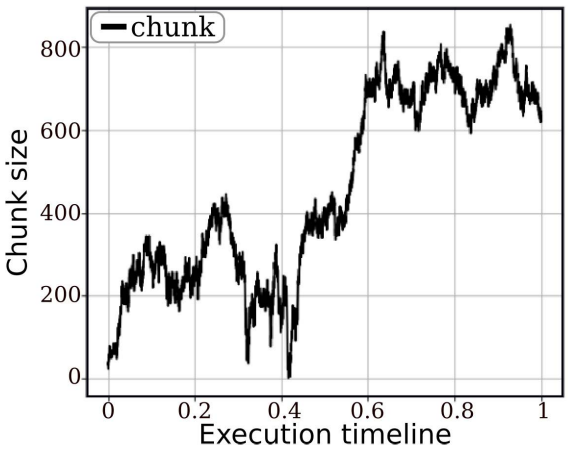


FIGURE 10. Dynamic chunk-size behavior.

size, and the workload does not need to be uniform during the entire execution. The dynamic chunk size proposal is well suited for cases where the workload varies over time, causing the optimal chunk size to change as well.

Considering the use case example for the static chunk size strategy, suppose that the set of images now satisfies $\forall i, I_i \in \mathbb{R}^{h_i \times w_i}$. In other words, every image has its own dimensions, so each one is now composed of a different number of $h_i \times w_i$ scalars. Consider, for example, that the images come from an online data stream that is not entirely known a priori. Alternatively, instead of images, the elements of our set could be more complex data structures leading to more complex tasks whose workload cannot be efficiently estimated beforehand.

In the above case, it is clear that a static chunk strategy is not the best method because the workload is not uniform during the entire execution, and it is not possible to estimate an optimal constant chunk size beforehand. For these cases, the dynamic chunk size constitutes a worthy alternative.

D. A STRATEGY BASED ON A WAREHOUSE OF CHUNKS

Storing tasks in a warehouse is proposed as the third strategy. In this approach, the main thread pushes chunks of tasks into a warehouse, so secondary threads can take them and proceed with their computation. Once the warehouse is full, the main thread takes a chunk and solves it. With this method, the main thread takes care of an entire chunk before inserting a new one, reducing the overhead from handling chunks between the computation of single tasks (as in the previous algorithms). When there are no more tasks for the current leg, the main thread behaves as a secondary thread and takes chunks from the warehouse until it gets empty. This strategy is illustrated in Figure 11. It is necessary to prevent secondary threads from going idle while the main thread is computing its chunk of tasks. To do so, a warehouse factor input argument f is used such that the warehouse size (i.e., the number of chunks it can store before being full) is $f \times n$, being n the number of threads. In consequence, secondary threads will have enough pending chunks to handle when using a high enough warehouse factor while the main thread is computing. Of course, selecting an optimum f is not straightforward because it depends on both the scene and the computer. Thus, a chunk of tasks can be computed in a smaller order of magnitude time than another. Besides, it is important to avoid a chunk size that is either too small, leading to excessive fragmentation of tasks, or too high, delaying the start of computations for each leg and increasing required memory. Despite this, a priori finding a sub-optimal but good enough warehouse factor is not too complicated for the general case. Empirical evidence from Section V-A and Section V-C shows that $f \in [4, 8]$ and $m \in [32, 64]$ are good choices for different scenes.

It is worth noticing in Figure 11 that the thread pool and the main thread have each their execution flow. Therefore, the thread pool is constantly taking tasks from the warehouse. When it cannot because it is empty, then it waits for a notify signal that must be triggered by the main thread when inserting a new chunk into the warehouse. In consequence, the starting point of this diagram is the main thread grabbing tasks from the leg. Once the threads at the thread pool are running, they will continue to execute their logic until entering the wait stage. Anytime that secondary threads are awakened by a notify signal, they stay executing on their own. Afterward, they wait either until the insertion of a new chunk or until the leg computation finishes. In essence, the proposal is similar to the typical consumer-producer problem introduced together with semaphores [22]. However, it uses wait and notify signals.

One paradigmatic type of use case that can benefit from our proposal is the family of algorithms based on multiple ray tracing operations, especially those involving a complex ray tracing model. The most typical ray tracing algorithms consist of a set of primitives S defining a scene such that for any primitive $p \in S$ its vertices, normal vector, and any relevant physical property are known. Note that any ray can be defined as $r = \{\mathbf{o}, \mathbf{u}\}$ where $\mathbf{o}$ is the origin point and $\mathbf{u}$ is the normalized vector specifying the ray's direction. Now, the basis of each ray tracing operation is to find the first intersection point $\mathbf{q} \in S \cap r$ in time $t \in \mathbb{R}$ between the ray starting at $\mathbf{o}$ and the closest primitive in the scene $p \in S$ through direction $\mathbf{u}$, if any. This case is illustrated through the particular minimization problem shown in (16). Then, a simple model based on a function $f(p, r)$ of the normal vector of the primitive, some of its physical properties, and the ray itself is computed. One well-known example is Phong's lighting model that considers the ambient, diffuse, and specular light coefficients defining the scene and the material of the object where the primitive belongs to [23].

$$\begin{aligned} \min \quad & t \\ \text{s.t.} \quad & \mathbf{o} + t\mathbf{u} = \mathbf{q} \\ & t > 0 \end{aligned} \tag{16}$$

Traditional lighting models that use typical ray tracing algorithms might be better suited for static or dynamic chunk

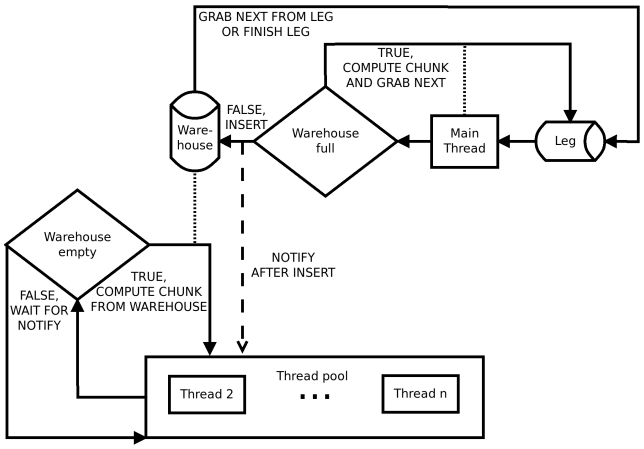


FIGURE 11. Warehouse logic.

strategies. On the one hand, if the scene is composed of similar primitives of the same type and the lighting model leads to a uniform computational cost for each task, then a static chunk size strategy is enough. For instance, a simple lighting model based on ray tracing applied to a regular triangular mesh is well suited to be divided into chunks with similar computational costs. Thus, it can be efficiently computed in a GPU because the workload can be uniformly distributed among warps of threads. On the other hand, a more irregular problem that computes a complex model on ray intersections and whose workload might vary depending on intermediary results, or a problem that deals with irregular meshes, might not be so well suited to be decomposed into uniform chunks. Suppose that the workload can be divided into different stages, with each stage having a similar burden per task. A dynamic strategy that automatically fits the chunk size could be enough for these cases. However, it might be that the problem does not even have such a coarse grain regularity to exploit. Then, the warehouse of chunks strategy must be preferred because it can improve the performance with respect to the dynamic chunk size strategy due to its uncoupled nature.

Moving aside from typical lighting models to more complex simulation models, the function $f(p, r)$ can be a complicated piecewise function that behaves differently depending on the primitive type, some stochastic process, or even different types of rays. One particular case is HELIOS++, for which the function $f(p, r)$ might lead to tasks requiring a different number of ray tracing operations. It is the case of transmittive detailed voxels for which it is possible to calculate an extinction coefficient needed to solve a simple stochastic computation based on a random uniform distribution, inspired by previous work from North [24]. Depending on this computation, either the intersection will be the last one or the ray will be allowed to pass through the intersection point, leading to at least one more ray tracing computation. These transmittive detailed voxels can be combined with other types of detailed voxels and different triangular meshes in the same scene, which will cause the workload to be very irregular. In consequence, this case is more adequate for the warehouse of chunks strategy than any previous ones.

V. PERFORMANCE ANALYSIS ON HELIOS++

All implementations described in previous sections have been thoroughly measured and compared through different operating systems and computers. For this purpose, an Intel Core i7-4930K at 3.40 GHz with 6 physical cores and 32GiB of DDR3 1600 MHz RAM computer with a GNU g++ 7.5.0 compiler has been used for Linux PC measurements, while an AMD Ryzen Threadripper 3970X at 3.70 GHz with 32 physical cores and 256GiB of DDR4 3000 MHz RAM computer with an MSVC v142 compiler has been used for Windows PC measurements. Finally, multiple scenes were tested at the CESGA FinisTerrae-II supercomputer using a node from its thinnodes partition with a GNU g++ 6.4.0 compiler. This node consists of 2 Haswell 2680v3 processors at 2.50 GHz with 12 physical cores each and 128 GiB of DDR4 2133 MHz RAM.

The data set of 5 different surveys summarized in Table 1 has been used for the performance analysis. For each scene, the number of primitives and the number of legs or simulation stages are specified. The volume of the bounding box containing the scene is also given, together with the serialized scene size in mebibytes. Note that the scan frequency (the number of scan lines per second) and the pulse repetition frequency (the number of laser pulses per second) are also specified. A summary of the qualitative features of different scenes is available in Table 2. The HAMMERLE survey comes from a work comparing different TLS and ULS (unmanned laser scanning) configurations. More concretely, the selected survey comes from a set of ULS configurations analyzed in terms of how accurately understory vegetation is represented in the point clouds [25]. The data is publicly available [26]. Furthermore, the DIWANG survey comes from a work that simulates TLS point clouds to build a high-quality synthetic data set to assess single tree isolation and leaf-wood classification through super point graphs [27]. The FIXED_250 and SCALED_COG surveys were used to investigate tree modeling for forestry purposes [9]. Finally, the ARBORETUM scene also comes from forestry-related projects. An example of what a simulated LiDAR point cloud looks like can be seen in Figure 12, which corresponds to the SCALED_COG scene.

A. PERFORMANCE ON LINUX PC

The execution time for the baseline HELIOS++ implementation is shown in Table 3 under the name of BL_Simple (baseline simple). In the same table, the behavior for the baseline implementation extended with FSAH is included under the name BL_FSAH. Both BLI_Simple (baseline improved simple) and BLI_FSAH (baseline improved fast surface area heuristic) entries represent a new version of the one task per thread paradigm with improved concurrency handling. This improvement was based on updating the software design to minimize the time spent in critical regions. The modification

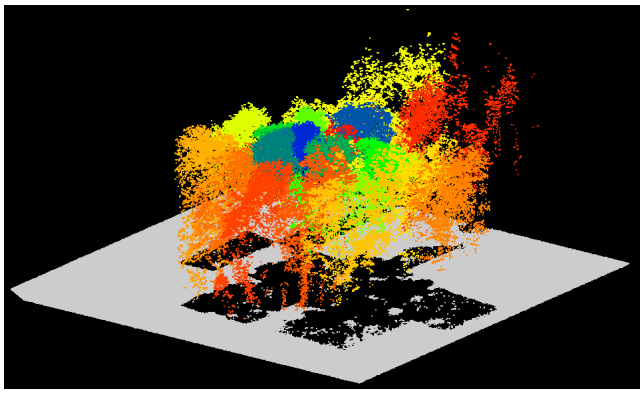


FIGURE 12. Virtual point cloud from SCALED_COG scene. Each point is colored depending on the object it belongs to.

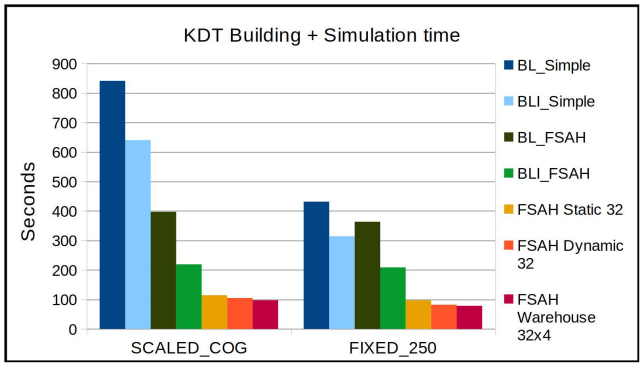


FIGURE 13. HELIOS++ performance comparison on Linux PC.

did not even half the sum of building and simulation times, which justified the three new strategies for parallel simulation introduced in this paper. They appear in the table as FSAH static 32, FSAH dynamic 32, and FSAH warehouse 32 × 4 (FSAH warehouse with chunk size 32 and warehouse factor 4). The three correspond with the FSAH heuristic because it has been the one that leads to the best execution times on Linux PC for both SCALED_COG and FIXED_250 surveys. A graphical comparison of the aforementioned measurements is shown in Figure 13. All executions were made using the 6 physical cores and the mean of 5 different runs.

Comparing BL_Simple and BL_FSAH implementations, there is a speedup of 1.81 in the KDT building. With both of these implementations, there is no parallel KDT building. Thus, the speedup is obtained only because of the fast approximation of SAH. The speedup of 2.18 in the simulation time is due to the improvement in ray intersection checks explained by governing the KDT construction with the FSAH. Updating from baseline (BL) to improved baseline (BLI) results in a total speedup of 1.31 for the simple case and 1.81 for the FSAH cases. The most significant benefit comes from the parallelization of the KDT building. When analyzing the impact of going from the one task per thread approach to the three new strategies, there are speedups of 2.16, 2.42, and 2.67 in simulation time for static, dynamic, and warehouse methods, respectively. With the new strategies, the simulations can run more than twice as fast.

To have a global perspective of the performance improvement, we compare the best strategy (FSAH warehouse 32×4) with the starting baseline (BL_Simple). In terms of KDT building time, there is a speedup of 5.21. For the simulation time, the speedup is 9.80. The total speedup is 8.64 for the SCALED_COG scene, while the FIXED_250 scene brings a total speedup of 5.5. These results imply that HELIOS++ is now between 5 and 8 times faster than the baseline implementation on a Linux PC. The distribution, type (FIXED_250 mixes voxels with triangles while SCALED_COG uses only triangles), serialized size, and the number of primitives in the scene, explain this variability.

B. PERFORMANCE ON WINDOWS PC

Regarding the execution on a Windows platform, we want to compare the performance using the case studies in Section V-C. Several executions took more than 24 hours to complete, so only a single measurement was taken per case study, i.e., one execution per number of cores and parallelization strategy. Each of the KDT building strategies was tested with each parallelization strategy listed in Section IV. The best configurations per scene are listed in Table 4, which contains the best case based on the total execution time and not on the speedup. The analysis of the speedup for every case in each scene is shown in Figure 14.

Regarding the SCALED_COG scene, the Simple KDT, together with a dynamic chunk size of 16 tasks, lead to the best execution time (131.6 s) using 32 cores. In Figure 14a, it can be observed that the speedup still has a positive slope, suggesting that increasing the number of available cores could lead to even lower execution times. Regarding SAH and FSAH, the speedup remains constant when using 7 or more cores. Regarding the execution times, even though the speedup is higher in the Simple Dynamic Case, the global execution times are lower when using 19 or fewer cores, so the FSAH strategy should be chosen in those cases for maximizing performance. When comparing the best case with the sequential execution time, the achieved time reduction is 93.05% in terms of full-time.

For the FIXED_250 scene, the Simple KDT heuristic yields the best execution time, 89.94 s, using a Dynamic chunk size configuration with an initial size of 32 tasks. In this case, the top performance is achieved at 16 cores, yielding a time reduction of 92.60% in full time. Looking to Figure 14b, the behavior of the three shown strategies is similar: there is an increasing speedup with fewer cores, but it becomes constant when more are available. In the case of SAH and FSAH, the speedup stops growing at around 5 cores. Before that point, speedups are similar for both strategies. Beyond that point, the speedup for FSAH is slightly higher because the full simulation time is lower for the sequential execution.

Concerning the HAMMERLE scene, using the FSAH heuristic and the warehouse-based parallelization led to a total simulation time of 71.22 s. It supposes a reduction of 99.15% compared to the sequential execution. The best result is achieved by configuring the warehouse with 32 tasks per

TABLE 1. Quantitative characteristics of the surveys used to measure HELIOS++ performance.

Scene	Primitives	Legs	Volume (m^3)	Size (MiB)	Scan freq. (Hz)	Pulse freq. (Hz)
FIXED_250	1 977 319	41	694 909.80	163	225	1 000 000
HAMMERLE	4 630 456	7	800 000.00	556	200	1 200 000
SCALED_COG	10 290 011	41	636 704.10	577	225	1 000 000
DIWANG	11 738 288	9	158 080.00	1614	50	100 000
ARBORETUM	13 711 899	21	1 555 288.67	866	50	100 000

TABLE 2. Qualitative characteristics of the surveys used to measure HELIOS++ performance.

Scene	Use case	Primitives	Platform	Scanner
FIXED_250	Forestry	Triangles and voxels	ALS	RIEGL VQ-780i
HAMMERLE	Flight comparison	Triangles	ULS	RIELG VUX-1UAV
SCALED_COG	Forestry	Triangles	ALS	RIEGL VQ-780i
DIWANG	Model assessment	Triangles	TLS	RIEGL VZ-400
ARBORETUM	Forestry	Triangles and voxels	Linear path	RIEGL VUX-1UAV

TABLE 3. Execution times for the SCALED_COG scene on the Linux PC.

Implementation	Execution time (s)		
	KDT build	Simulation	Total
BL_Simple	128.27	713.40	841.67
BLI_Simple	43.04	597.20	640.24
BL_FSAH	70.77	326.40	397.17
BLI_FSAH	24.86	194.40	219.26
FSAH static 32	24.70	89.80	114.50
FSAH dynamic 32	24.77	80.20	104.97
FSAH warehouse 32x4	**24.64**	**72.80**	**97.44**

chunk and with as many chunks as 8 times the number of threads. The best performance with this configuration was obtained with 32 cores, suggesting that increasing the available cores would produce even faster simulations. Looking at the HAMMERLE speedups in Figure 14c, we see that there is room for lower execution times, even when the slope of the speedup is close to being flat. In the same figure, note that the speedup for the Simple Static 64 case is remarkably higher than the chosen strategy. Despite this fact, the execution times are higher in that case. The better speedup is explained by the very high simulation time in the Simple Static 64 sequential case.

Regarding the DIWANG scene, the best full simulation time was obtained when combining the SAH heuristic with a static parallelization strategy of 64 chunks. In this case, the total simulation time is 808.71 s. The speedup behavior can be observed in Figure 14d. The SAH and FSAH speedups are parallel along the figure. The lightly higher speedup of the SAH Static 64 strategy is due to the higher simulation time in the sequential case. In any case, the performance of both the SAH and FSAH are pretty similar. For all three strategies, the speedup slope at 32 cores is positive, which suggests that the speedup would continue to grow if more cores were available. Finally, the time reduction achieved with the best parallelization strategy is 98.99%.

Concerning the ARBORETUM scene, the best execution time was obtained when using the Simple KDT heuristic, combined with a warehouse parallelization with 64 tasks chunks and maximum storage of as many chunks as 8 times the number of available threads. The best time (212.31 s) was achieved using 21 cores. In Figure 14e, it can be observed that the speedup remains constant from there. The same behavior is observed in the SAH Dynamic and FSAH Static cases, where the constant speedup starts at 6 cores. With the Simple Warehouse strategy, the speedup is higher than the other cases, and the execution time outperforms the SAH and FSAH heuristics from 10 cores onward. In this case, the achieved reduction is 89.51% in terms of full-time.

For any survey, the average reduction with the best strategy in terms of simulation time, compared to the baseline, is 90.23%, more than enough to justify the implementation of the parallelization strategies.

C. PERFORMANCE ON SUPERCOMPUTER

The data set introduced in Table 1 has been used to measure the performance of HELIOS++ in the CESGA FinisTerrae-II supercomputer. This benchmark has a single measurement for each case instead of a mean of measurements because some executions took more than 24 hours. For the SCALED_COG and FIXED_250 scenes, we executed all combinations between the three different KDT building algorithms (Simple, SAH, and FSAH) and the 3 different parallelization strategies (Static, Dynamic, Warehouse), from 1 core to 24 cores. We consider the Simple and FSAH KDT combined with the three different parallelization strategies for the remaining surveys. The former is justified because previous executions proved that the FSAH heuristic is faster than the SAH heuristic at building time without losing efficiency at simulation time. The best combination of KDT building strategy, parallelization strategy, and the number of cores for each survey is shown in Table 5. Execution times and speedups of the best parallelization strategy for each of the KDT building methods are depicted in Figure 15, with continuous and dashed lines, respectively. Besides, since we combined different heuristic approaches with multiple parallelization strategies, we propose a global speedup metric that considers both improvements in (17). Here, T_{best} stands for the execution time obtained with the best strategy, while

TABLE 4. Best execution times and configuration by scene at Windows PC.

Scene	Best configuration			Best time (s)		
	KDT	Parallelization	Cores	KDT build	Simulation	Total
FIXED_250	Simple	Dynamic 32	16	2.94	87.0	89.94
HAMMERLE	FSAH	Warehouse 32x8	32	25.22	46.0	71.22
SCALED_COG	Simple	Dynamic 16	32	25.60	106.0	131.60
DIWANG	SAH	Static 64	32	196.71	612.0	808.71
ARBORETUM	Simple	Warehouse 64x8	21	29.31	183.0	212.31

T_{simple} stands for the sequential execution time of the survey using a simple KDT. The time reduction percentage can be obtained from global speedup as indicated in (18). All global speedups are introduced in Table 6.

$$S_{\text{global}} = \frac{T_{\text{best}}}{T_{\text{simple}}} \tag{17}$$

$$T_{\text{reduction}} = 1 - \frac{1}{S_{\text{global}}} \tag{18}$$

For the SCALED_COG scene, the best execution time was obtained with the FSAH heuristic using a warehouse-based parallelization with chunks of 64 tasks and maximum storage of as many chunks as 4 times the number of available threads. The best performance was obtained using 9 cores, showing that using more cores did not lead to shorter execution times. Looking at Figure 15a, it can be seen that the SAH strategy presents an execution time that is very close to the FSAH one. Both heuristics lead to a simulation time of 64 seconds. Despite this, building the KDT took 79.28 seconds for the SAH heuristic but only 28.96 seconds for the FSAH heuristic, which explains why the latter is preferred. While the Simple KDT presents a more significant speedup, it is not enough to compensate for its inefficiency as a KDT building strategy concerning SAH-based approaches. Compared with the sequential execution time using a simple KDT, the time reduction brought by the best strategy is around 96.95% in terms of full-time.

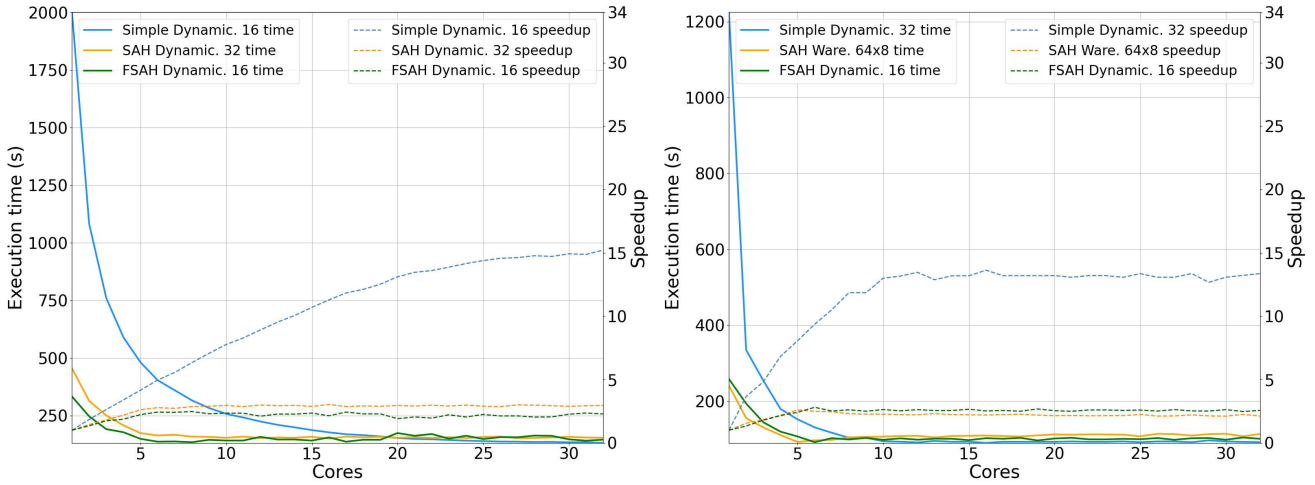
For the FIXED_250 scene, the best execution time was obtained with the Simple heuristic using a warehouse-based parallelization with chunks of 64 tasks and maximum storage of as many chunks as 8 times the number of available threads. We obtained the best performance using 24 cores, suggesting that more available cores would reduce the execution time even more. Nevertheless, looking at Figure 15b, the speedup for the Simple KDT is starting to decelerate, which means adding more cores is not enough to reduce its execution time. The FSAH heuristic is the way to go with fewer cores because it significantly reduces the execution time with just 10 cores. It is worth mentioning that SAH and FSAH heuristics lead to 63 and 62 seconds of simulation time. They also lead to 32.23 and 10.74 seconds of KDT building time, respectively. The Simple KDT overcomes them despite its higher simulation time of 67 seconds because it presents a significantly lower KDT building time of 3.62 seconds. Even so, the difference in total execution time between Simple and FSAH algorithms is bare of 2.12 seconds. If we compare it with the sequential execution time using a Simple KDT, the time reduction brought by the best strategy is around 91.47% in terms of full-time.

Considering the HAMMERLE scene, the best execution time was obtained with the FSAH heuristic using a dynamic chunk size with an initial size of 32 tasks. The best performance was obtained using 24 cores, suggesting that, with more available cores, it could be possible to reduce execution time even more. As in the FIXED_250 scene, the speedup at Figure 15c shows a significant deceleration, which means the scalability is low, and it is not possible to achieve an important reduction of execution time simply by increasing available cores. Compared with the sequential execution time using a simple KDT, the time reduction brought by the best strategy is around 99.10% in terms of full-time.

Concerning the DIWANG scene, the best execution time was obtained with the FSAH heuristic using a warehouse-based parallelization with chunks of 64 tasks and maximum storage of as many chunks as 8 times the number of available threads. The best performance was obtained using 24 cores, suggesting high scalability. Thus, with more available cores, it should be possible to reduce the execution time accordingly. From Figure 15d, it can be seen that the speedup for the FSAH KDT is starting to decelerate, so the reduction of execution time will not hold for many more cores. Compared with the sequential execution time using a simple KDT, the time reduction brought by the best strategy is around 99.40% in terms of full-time.

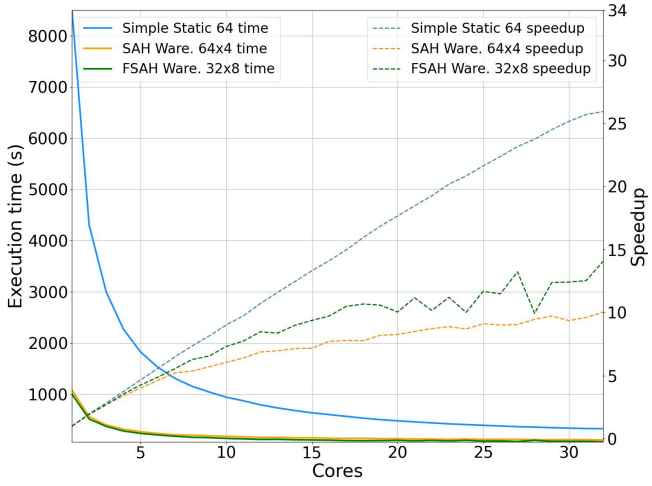
For the ARBORETUM scene, the best execution time was obtained with the FSAH heuristic using a warehouse-based parallelization with chunks of 64 tasks and maximum storage of as many chunks as 8 times the number of available threads. The best performance was obtained using 14 cores, showing that more cores did not lead to shorter execution times. According to Figure 15e, it can be seen that the FSAH-based approach reaches a near-constant speedup at around 10 cores. Compared with the sequential execution time using a simple KDT, the time reduction brought by the best strategy is around 95.25% in terms of full-time.

On the one hand, there are scenes such as SCALED_COG and ARBORETUM, where the FSAH reaches its best performance and then enters a near-constant speedup stage. On the other hand, there are scenes like HAMMERLE and DIWANG where the FSAH presents a decelerated but increasing speedup. However, the FSAH KDT often needs fewer cores than the Simple KDT to reach its highest efficiency. Also, warehouse-based parallelization strategies often achieve the best performance or are close to the best one.

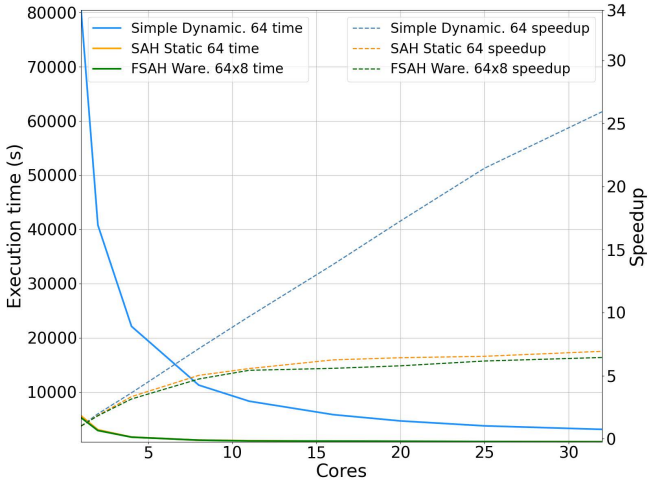


(a) SCALED_COG.

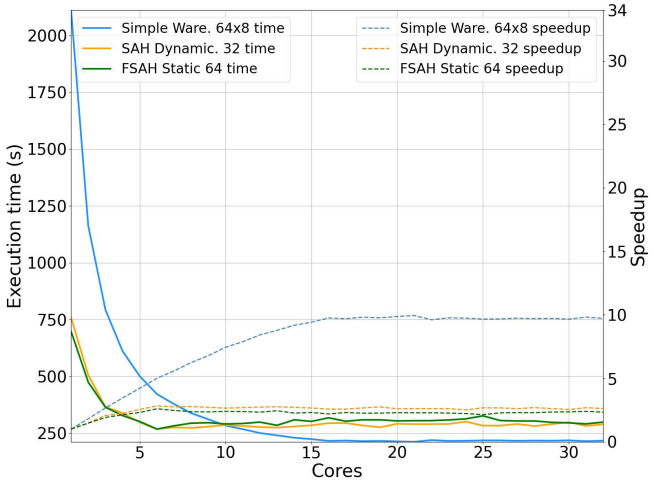
(b) FIXED_250.



(c) HAMMERLE.

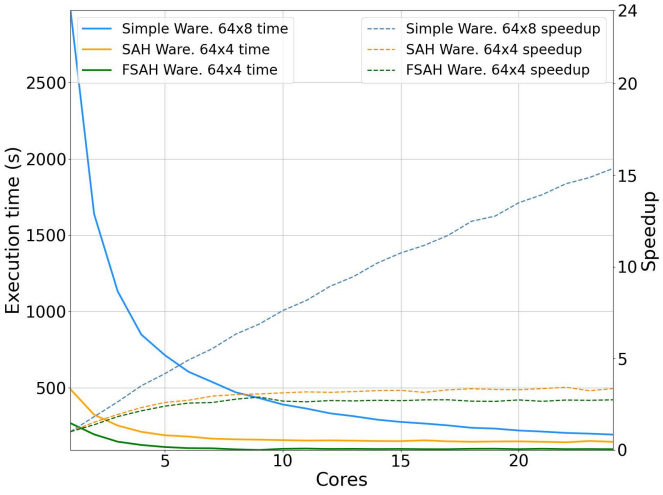


(d) DIWANG.

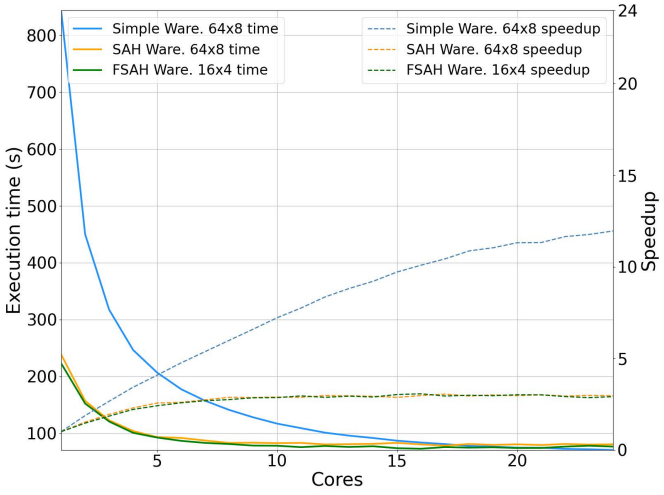


(e) ARBORETUM.

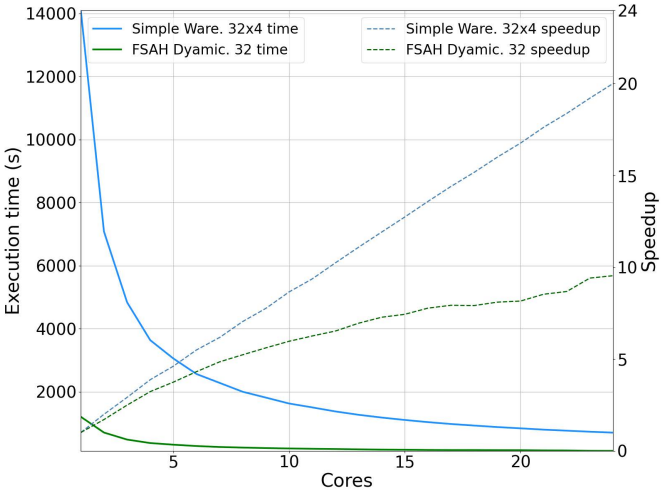
FIGURE 14. Relationship between the execution time and the number of cores at Windows PC. The best simulation strategy is considered for each analyzed KDT building strategy. The solid lines represent the execution times, and the dashed lines the speedups.



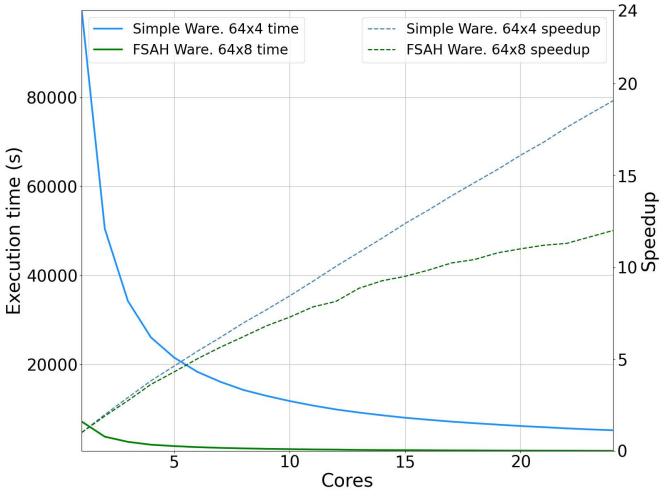
(a) SCALED_COG.



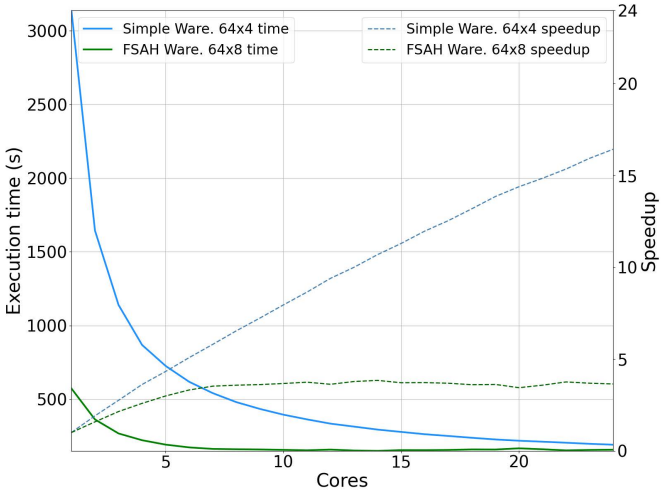
(b) FIXED_250.



(c) HAMMERLE.



(d) DIWANG.



(e) ARBORETUM.

FIGURE 15. Relationship between the execution time and the number of cores at CESGA FinisTerrae-II. The best simulation strategy is considered for each analyzed KDT building strategy. The solid lines represent the execution times, and the dashed lines the speedups.

TABLE 5. Best execution times and configuration by scene at CESGA FinisTerrae-II.

Scene	Best configuration			Best time (s)		
	KDT	Parallelization	Cores	KDT build	Simulation	Total
FIXED_250	Simple	Warehouse 64x8	24	3.62	67	70.62
HAMMERLE	FSAH	Dynamic 32	24	19.56	107	126.56
SCALED_COG	FSAH	Warehouse 64x4	9	28.96	64	92.96
DIWANG	FSAH	Warehouse 64x8	24	77.33	521	598.33
ARBORETUM	FSAH	Warehouse 64x8	14	36.24	113	149.24

TABLE 6. Global speedups between best strategy and simple sequential at CESGA FinisTerrae-II.

Scene	Simple sequential time (s)			Best speedup		
	Build	Sim.	Full	Build	Sim.	Full
FIXED_250	12.16	816	828.16	3.36	12.18	11.73
HAMMERLE	94.32	13990	14084.32	4.82	130.75	111.28
SCALED_COG	163.26	2889	3052.26	**5.64**	45.14	32.83
DIWANG	271.13	99234	99505.13	3.51	**190.47**	**166.31**
ARBORETUM	89.43	3051	3140.43	2.47	27.00	21.04

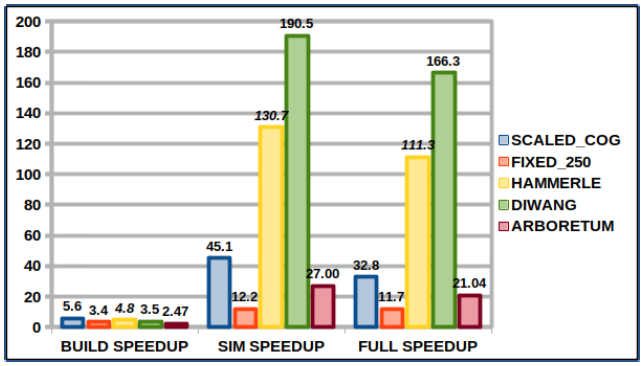


FIGURE 16. Global speedups between best strategy and simple sequential.

As stated in Section IV-D, the warehouse parallelization strategy significantly improves when using 4 and 8 warehouse factors with a chunk size of 64.

Finally, a comparison in terms of global speedup is available in Figure 16. Considering the entire execution, the surveys with higher simulation times and fewer legs show the most significant improvement. There are two reasons for this. The first one is that a higher simulation time implies that the parallelization strategy for the simulation has more influence in the full time. The second one is that the greater the number of legs, the more the synchronization barriers between computational intensive intervals. Thus, HAMMERLE and DIWANG surveys have bigger blocks of pure computing while SCALED_COG, FIXED_250, and ARBORETUM require more synchronization stages where parallel computing stops to end the current leg and prepare the next one.

VI. DISCUSSION AND RELATED WORK

In this section, we discuss our proposals from three different perspectives. First, we investigate how our proposed algorithms can benefit other LiDAR simulators. Second, we analyze the application scope of our algorithms apart from LiDAR simulation. Finally, we highlight the various future lines of work that will be made possible by the performance improvements of HELIOS++ through the application of HPC techniques.

A. RELATED SIMULATORS

There are various approaches to LiDAR simulation in the literature. A direct comparison between these models is not always possible because of their different nature. Some proposals cover only certain types of platforms, such as the work of Hodge on simulating TLS to study the relationship between the scanned surface and the scanner concerning errors in point clouds [28]. Some proposals use stochastic methods, such as the Monte Carlo-based model by North to study the relationship between the waveform and vegetation canopies [29]. Others propose analytical modeling of the waveform, such as Tulldahl's simulation study on the impact of objects on the seabed [30]. It is also important to differentiate general scope simulators, such as the one proposed by Lohani to study ALS with different terrains, sensors, and platforms [31], from specific purpose simulators such as the one of Ranson that aims to study multiple returns in canopies [32]. A more thorough discussion of the different LiDAR simulators is available in the work of Schlager from the perspective of driver assistance systems [33] as well as in our previous paper introducing HELIOS++ [1].

Concerning the computational relationship between the different simulators and the improvements proposed in this paper, we think that the KDT building heuristics could benefit most simulators that use a ray tracing algorithm. For example, Wang proposed a TLS simulation method based on ray tracing over bounding boxes [34]. We think this proposal could benefit from using a KDT and governing its building with SAH-based heuristics. The only cases that would not benefit from the heuristics are simulators with a small computational burden related to ray intersection checks because the cost increase of building the KDT might exceed the cost decrease in the ray tracing method.

Our parallelization strategies are only suitable for specific frameworks. HELIOS++ is a general-purpose LiDAR simulator that supports flexible modeling. For instance, the scenes

can be composed of different primitives, with some leading to a single hit ray casting process and others leading to a more complex ray tracing process supporting recursive hits. This flexibility of HELIOS++ comes at the expense of complicating GPU implementations. Simulators using complex scenes composed of different primitives may therefore benefit from our parallelization strategies. However, we strongly suggest that less flexible models working with one type of primitive, having regular spatial distributions, and solving the same physical model for each primitive, use a classical GPU approach.

B. WIDER APPLICATION SCOPE

The parallelization strategies proposed in this paper should work in any application scope as long as two conditions are satisfied. The first one is that the computational burden must be considerable. Otherwise, the thread handling overhead will exceed the time reduction from parallel computing. This condition means that our algorithms are not expected to be efficient when applied to small problems. The second one is that the computations must be divisible into blocks of tasks with no internal dependencies such that each block has a significant computational burden. Otherwise, those blocks with a reduced computational burden will lead to an overhead that exceeds the benefits. This condition means that our algorithms are not expected to be efficient for cases with frequent synchronization barriers.

To explore the scope of our algorithms, we first consider a simple general use case and then move on to a more complicated one.

For the simple general case, we consider the *Seek_Approval* relation as defined by Rajkumar and Sandhu [35]. This relation is a subset of the cartesian product of the administrative user AU seeking approval, the set of relations $\{RL\}$ that will change after approval, the user U affected by the updated relations, and the set of co-administrators CAU whose approval is requested. Formally speaking, the definition is $Seek\_Approval \subseteq AU \times \{RL\} \times U \times \{CAU\}$. Suppose we have many access control operations divided into blocks $B_1, \ldots, B_n$. Let us assume that the operations for any i-th block do not share dependencies. Let us also assume that executing the operations from block B_{i+1} is only allowed after all operations from block B_i have finished because operations from B_{i+1} depend on the result of operations from B_i. In our simple case, the execution of the blocks will not start until all necessary *Seek_Approval* are confirmed. This particular condition is not limited to the POSTER access control model [35]. It is similar to handling concurrent transactions on a database, where a set of operations must be effective only if previous dependent transactions have been successfully committed [36].

Thus, our parallel algorithms can be applied to these use cases as long as each block implies a big computational burden. Besides, if the operations defining each block present a varying workload, then the dynamic chunk size and the warehouse of chunks are the preferable strategies. We think

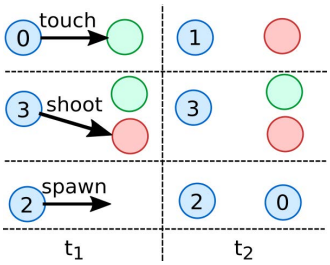


FIGURE 17. Representation of the toy game. The blue balls are the players, the green balls are the untouched balls, and the red balls are the touched balls. The number inside the blue balls represents the number of balls touched by that player.

the algorithms presented in this paper are well suited because we have already tested them with the simulation software HELIOS++. In this software, computations are divided into legs such that the i-th leg must finish before proceeding to the $(i+1)$-th leg. There is a clear equivalence between the blocks of independent access control operations and the legs of HELIOS++ simulations. Both are the main synchronization barriers.

To study a more intricate use case, we consider Pre_UCON_A the pre-authorization sub-model of the $UCON_{ABC}$ usage control model [37]. The $UCON_{ABC}$ is a unified model to handle the security requirements of information systems [38]. According to Rajkumar and Sandhu, it is possible to abstract the usage control scheme to a toy game example involving balls [37]. In this toy game, the object schema contains the *count* attribute with values from $\{0, 1, 2, 3\}$ and the *color* attribute with values from {red, green, blue}. The usage commands are associated to the functions touch(a, b), shoot(a, b), and spawn(a, b), where a and b can be any pair of balls in the game. The set of rights is considered by the usage commands to decide whether to compute one of the previous functions or not.

In the toy game, the blue balls are the players. Consequently, the first condition that any usage command will check is that $a.color$ = blue. Then, depending on the usage command, it is necessary to check some additional conditions. If a wants to touch a ball b, it is necessary that $b.color$ = green. When satisfying these conditions, the color of b will be updated to red. At the same time, the count of touched balls for a will be increased by 1 up to a maximum of 3. If a wants to shoot a ball b, it is necessary that $a.count = 3$ but also that $b.color \neq$ blue because both red and green balls can be shot but blue balls (players) cannot. If a wants to spawn a new blue ball b, it can do it anytime because there are no additional conditions. The spawned ball will have its count of touched balls initialized to 0, such that $b.count = 0$. Figure 17 illustrates the mechanics of this toy game.

The first thing to note is that non-uniform workloads can be a frequent scenario since each block of operations is composed of the player's arbitrary decisions. In consequence, the static chunk size is not well suited for this toy game. Nonetheless, the dynamic chunk size and the warehouse of chunks might be efficient as long as synchronization barriers are not too frequent. If the frequency of synchronization

barriers prevents the grouping of tasks into blocks with a substantial workload, then no performance improvement can be expected from our algorithms. For example, if there is one ball and many players, but only the first player to reach the ball can interact with it at a given time, then there is one task per synchronization barrier. If the frequency of synchronization barriers does not prevent the formation of blocks with a significant workload, our algorithms might improve the performance. For example, if there are multiple players and balls, having multiple parallel ball-player interactions, there are many tasks per synchronization barrier. To summarize, our algorithms can efficiently parallelize multiple simultaneous ball games but not a single ball game.

Finally, we discuss the benefits of using a KDT for a variant of this toy game. Assuming the toy game uses a KDT to account for the position of the balls and to compute a ray tracing algorithm to find the closest intersection in a straight line (for instance, the closest player in terms of euclidean distance), our proposals to improve the KDT should be considered. The first benefit of using the FSAH KDT is the reduction of building time, and the second is a more efficient ray tracing. These improvements are adequate for any software that computes a ray tracing algorithm in Euclidean space.

C. FUTURE WORK

General purpose simulation is quite a broad topic, even when restricted to a specific domain, such as virtual point cloud generation from laser scanning simulation. Hence, there are many different branches of future work. The performance improvements achieved with the techniques explained in this paper make it possible to extend HELIOS++ to even more applications.

First, it is possible to integrate HELIOS++ in a cloud computing environment over a Software as a Service (SaaS) paradigm. Since all the parallel strategies are implemented in a shared memory context, deploying HELIOS++ in the cloud is straightforward because complicated multiprocessing concerns are out of the question. As most standard cloud platforms already handle the life cycle of shared memory nodes with many cores, the only remaining tasks are creating a minimal framework for handling executions and providing a web interface. We think this would make the software more accessible to the entire scientific community, bringing many benefits in terms of transparency, replicability, and accessibility.

Second, we are interested in studying the idea of dynamic simulations. By this, we mean simulations where at least one component in the scene updates its state during the simulation. There are different types of dynamism. The more complicated type is the one that implies frequent spatial modifications because they make it necessary to design efficient algorithms to update the ray tracing related data structures (i.e., the KDT) as well as computing frequent transformations on the dynamic objects. While there is a rich literature on dynamic scenes, the problem is more complicated for HELIOS++ for two reasons. First, we are not simply rendering a scene but computing a laser scanning model. Thus, we must implement an extremely efficient method because the computational cost can easily become prohibitive. The second one is that we want to keep the philosophy of HELIOS++ as a flexible simulator. In consequence, we cannot resort to typical GPU solutions because they will not be efficient for the different scenes supported by the simulator.

Next, we are planning on supporting multichannel scanners. Thanks to the performance improvements, we expect that computing a simulation with a dual channel scanner will take less time than computing a single channel scanner before the improvements. Until now, the simulation of a multichannel scanner on HELIOS++ requires computing multiple simulations changing the scanner configuration for each case, and merging them manually. This manual approach can be a cumbersome procedure, especially for cases where the number of channels is high (e.g., the Velodyne Puck with 16 channels). We will also explore the feasibility of an algorithm to reach a sublinear scale in the computational cost with respect to the number of channels.

Finally, we think HELIOS++ has the potential to be used as a virtual ground-truth generator so that the output point clouds are well suited for training artificial intelligence models. We want to explore this issue in depth to advance research in point clouds because the lack of large amounts of consistent ground truth data is one of the main problems in the field, especially when compared with the available ground truth data in research fields such as image processing.

VII. CONCLUSION

The main conclusion from this work is that the use of HPC techniques and advanced tree-like data structures is adequate to improve the performance of complex and irregular simulations. This conclusion holds even for those cases which are not embarrassingly parallel because of the large number of potential workflow branches the execution might take. However, integrating the different HPC algorithms makes the software design more complex. Consequently, increased development and software maintenance costs must be considered.

A. KDT BUILDING ALGORITHMS

The SAH enhancement leads to a significant improvement of HELIOS++ performance. We showed that our FSAH version of the SAH (or any other accurate yet fast approximation of the heuristic) improves the performance even more because it preserves the efficiency of the KDT with a lower building cost. This conclusion is shared by the multiple works that used SAH-based implementations in the last three decades. However, most of these works are related to regular triangular meshes, one of the most common data structures in simulations, computer-generated graphics, video games, and 3D inventories. In addition, our work shows that the performance improvement of SAH-inspired solu-

tions applies to more complex scenes with different types of primitives and irregular meshes with different granularity. Our simple algorithm to pick the best partition axis at each depth did not lead to an improved simulation performance in our case.

B. PARALLELIZATION STRATEGIES

Among the parallelization strategies proposed in this work, the warehouse of chunks is the most efficient for the general case. Simpler strategies sometimes result in slightly better execution times for the simplest surveys with low execution times. Nonetheless, even here, the warehouse strategy offers significant improvements. Thereby, among the studied parallelization strategies, the warehouse of chunks with a size that is a small multiple of the number of available threads is the best approach to deal with complex simulations that can present irregular and varying workloads during their execution. It is expected that this strategy works better than the static and dynamic ones for any software that deals with big and irregular problems, whether they are simulations or not.

C. BEST CONFIGURATION

For the particular case of HELIOS++, it is possible to define a configuration for the general case that leads to a significant performance improvement. Our experiments show that using an FSAH with 32 bins to govern the KDT building together with a warehouse parallelization strategy with a chunk size of either 32 chunks or 64 chunks and a warehouse factor of either 4 or 8, boosts performance for any case. Even for those cases where other KDT building strategies bring similar improvements, the FSAH strategy is recommended because it is likely to offer the same performance with fewer cores. The benefits of testing different execution configurations to improve this general case configuration are not expected to be big enough to compensate for the extra effort.

ACKNOWLEDGMENT

Performance measurements on the FinisTerrae-II supercomputer were possible thanks to the CESGA (Galician supercomputing center).

REFERENCES

[1] L. Winiwarter, A. M. Esmorís Pena, H. Weiser, K. Anders, J. Martínez Sánchez, M. Searle, and B. Höfle, "Virtual laser scanning with HELIOS++: A novel take on ray tracing-based simulation of topographic full-waveform 3D laser scanning," *Remote Sens. Environ.*, vol. 269, Feb. 2022, Art. no. 112772.

[2] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[3] L. H. Ceze, *Shared-Memory Multiprocessors*. Boston, MA, USA: Springer, 2011, pp. 1810–1812.

[4] P. Heckbert, "Color image quantization for frame buffer display," *ACM SIGGRAPH Comput. Graph.*, vol. 16, no. 3, pp. 297–307, Jul. 1982.

[5] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Comput. Graph. Appl.*, vol. CGA-4, no. 10, pp. 15–24, Oct. 1984.

[6] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, Jun. 1984.

[7] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *Vis. Comput.*, vol. 6, no. 3, pp. 153–166, May 1990.

[8] F. Noichl, A. Braun, and A. Borrmann, "'BIM-to-scan' for scan-to-BIM: Generating realistic synthetic ground truth point clouds based on industrial 3D models," in *Proc. Eur. Conf. Comput. Construct. Comput. Construct.*, vol. 2, 2021, pp. 164–172.

[9] H. Weiser, L. Winiwarter, K. Anders, F. E. Fassnacht, and B. Höfle, "Opaque voxel-based tree models for virtual laser scanning in forestry applications," *Remote Sens. Environ.*, vol. 265, Nov. 2021, Art. no. 112641.

[10] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The KITTI dataset," *Int. J. Robot. Res.*, vol. 32, no. 11, pp. 1231–1237, Sep. 2013.

[11] I. Armeni, O. Sener, A. R. Zamir, H. Jiang, I. Brilakis, M. Fischer, and S. Savarese, "3D semantic parsing of large-scale indoor spaces," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 1534–1543.

[12] N. M. Singer and V. K. Asari, "DALES objects: A large scale benchmark dataset for instance segmentation in aerial LiDAR," *IEEE Access*, vol. 9, pp. 97495–97504, 2021.

[13] T. Hackel, N. Savinov, L. Ladicky, J. D. Wegner, K. Schindler, and M. Pollefeys, "Semantic3D.Net: A new large-scale point cloud classification benchmark," *ISPRS Ann. Photogramm., Remote Sens. Spatial Inf. Sci.*, vol. IV-1/W1, pp. 91–98, May 2017.

[14] X. Roynard, J.-E. Deschaud, and F. Goulette, "Paris-Lille-3D: A large and high-quality ground-truth urban point cloud dataset for automatic segmentation and classification," *Int. J. Robot Res.*, vol. 37, no. 6, pp. 545–557, 2018.

[15] E. Agapaki, A. Glyn-Davies, S. Mandoki, and I. Brilakis, "CLOI: A shape classification benchmark dataset for industrial facilities," in *Proc. Comput. Civil Eng.*, Jun. 2019, pp. 66–73.

[16] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes," *Comput. Graph. Forum*, vol. 26, no. 3, pp. 395–404, Sep. 2007.

[17] J. Hurley, E. Kapustin, E. Reshetov, and A. Soupikov, "Fast ray tracing for modern general purpose CPU," in *Proc. Graphicon*, Jan. 2002, pp. 1–8.

[18] S. Popov, J. Gunther, H.-P. Seidel, and P. Slusallek, "Experiences with streaming construction of SAH KD-trees," in *Proc. IEEE Symp. Interact. Ray Tracing*, Sep. 2006, pp. 89–94.

[19] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, "Parallel SAH k-D tree construction," in *Proc. Conf. High Perform. Graph. (HPG)*, 2010, pp. 77–86.

[20] D. Wehr and R. Radkowski, "Parallel kd-tree construction on the GPU with an adaptive split and sort strategy," *Int. J. Parallel Program.*, vol. 46, no. 6, pp. 1139–1156, Dec. 2018.

[21] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, W. Bhimji, Prabhat, and P. Dubey, "PANDA: Extreme scale parallel K-Nearest neighbor on distributed architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 494–503.

[22] E. W. Dijkstra, "Cooperating sequential processes," Eindhoven Univ. Technol., Eindhoven, The Netherlands, Tech. Rep. EWD-123, 1965.

[23] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, no. 6, pp. 311–317, Jun. 1975.

[24] P. R. J. North, "Three-dimensional forest light interaction model using a Monte Carlo method," *IEEE Trans. Geosci. Remote Sens.*, vol. 34, no. 4, pp. 946–956, Jul. 1996.

[25] M. Hämmerle, N. Lukač, K.-C. Chen, Z. Koma, C.-K. Wang, K. Anders, and B. Höfle, "Simulating various terrestrial and UAV LiDAR scanning configurations for understory forest structure modelling," *ISPRS Ann. Photogramm., Remote Sens. Spatial Inf. Sci.*, vol. IV-2/W4, pp. 59–65, Sep. 2017.

[26] M. Hämmerle, N. Lukac, K.-C. Chen, Z. Koma, C.-K. Wang, K. Anders, and B. Höfle, "HELIOS full-waveform laser scanning simulation framework. Source code, precompiled version, example files for study of understory tree height scanning and respective output," Version V1, heiDATA, Geographisches Institut Heidelberg, Heidelberg, Germany, 2017, doi: 10.11588/data/10101.

[27] D. Wang, "Unsupervised semantic and instance segmentation of forest point clouds," *ISPRS J. Photogramm. Remote Sens.*, vol. 165, pp. 86–97, Jul. 2020.

[28] R. A. Hodge, "Using simulated terrestrial laser scanning to analyse errors in high-resolution scan data of irregular surfaces," *ISPRS J. Photogramm. Remote Sens.*, vol. 65, no. 2, pp. 227–240, Mar. 2010.

[29] P. R. J. North, J. A. B. Rosette, J. C. Suárez, and S. O. Los, "A Monte Carlo radiative transfer model of satellite waveform LiDAR," *Int. J. Remote Sens.*, vol. 31, no. 5, pp. 1343–1358, Mar. 2010.
[30] H. M. Tulldahl and K. O. Steinvall, "Analytical waveform generation from small objects in LiDAR bathymetry," *Appl. Opt.*, vol. 38, pp. 1021–1039, Feb. 1999.
[31] B. Lohani and R. Mishra, "Generating LiDAR data in laboratory: LiDAR simulator," *Int. Arch. Photogramm. Remote Sens.*, vol. 52, no. 1, pp. 1–6, 2007.
[32] G. Sun and K. J. Ranson, "Modeling LiDAR returns from forest canopies," *IEEE Trans. Geosci. Remote Sens.*, vol. 38, no. 6, pp. 2617–2626, Nov. 2000.
[33] B. Schlager, S. Muckenhuber, S. Schmidt, H. Holzer, R. Rott, F. M. Maier, K. Saad, M. Kirchengast, G. Stettinger, D. Watzenig, and J. Ruebsam, "State-of-the-art sensor models for virtual testing of advanced driver assistance systems/autonomous driving functions," *SAE Int. J. Connected Automated Vehicles*, vol. 3, no. 3, pp. 233–261, Oct. 2020.
[34] Y. Wang, D. Xie, G. Yan, W. Zhang, and X. Mu, "Analysis on the inversion accuracy of LAI based on simulated point clouds of terrestrial LiDAR of tree by ray tracing algorithm," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, Jul. 2013, pp. 532–535.
[35] R. P. V. and R. Sandhu, "POSTER: Security enhanced administrative role based access control models," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1802–1804.
[36] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*. New York, NY, USA: McGraw-Hill, 2006.
[37] P. V. Rajkumar and R. Sandhu, "Safety decidability for pre-authorization usage control with identifier attribute domains," *IEEE Trans. Depend. Secure Comput.*, vol. 17, no. 3, pp. 465–478, May 2018.
[38] J. Park and R. Sandhu, "The uconabc usage control model," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 128–174, Feb. 2004.

ALBERTO M. ESMORÍS was born in Santiago de Compostela, in February 1993. He received the bachelor's degree in computer engineering from the Universitat Oberta de Catalunya, in 2018, and the master's degree in high performance computing from the Universidade de Santiago de Compostela and Universidade da Coruña. He is currently pursuing the Ph.D. degree in high performance computing and artificial intelligence with the Universidade de Santiago de Compostela, Spain.

He is also working as a Researcher with the Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela. His research interests include theoretical computer science, high performance computing, artificial intelligence, and point clouds. The main focus of his work is the development of algorithms to process real and virtual LiDAR point clouds.

MIGUEL YERMO was born in Corcubión, in November 1990. He received the B.S. degree in physics and the M.S. degree in applied mathematics from the Universidade de Santiago de Compostela, Spain, in 2015 and 2017, respectively, where he is currently pursuing the Ph.D. degree in high performance computing.

In 2017, he worked as a Research Assistant with the Faculty of Mathematics, Universidade de Santiago de Compostela, developing magnetic equivalent circuits to speedup FEM simulations of a novel electric generator, where he has been working as a Research Assistant with the Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), since 2018. His research interests include high performance computing, embedded systems, remote sensing, and developing rule based classifiers for 3D point clouds. Most of his work is focused on developing algorithms to process real airborne LiDAR point clouds.

HANNAH WEISER received the B.S. degree (Hons.) in geography from Heidelberg University, Germany, in 2020, where she is currently pursuing the M.S. degree in geography with a focus on geoinformatics.

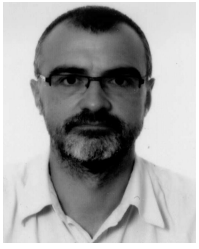
Since October 2018, she has been a Student Research Assistant with the 3DGeo Research Group, Institute of Geography, Heidelberg University. Her research interests include virtual laser scanning and 3D/4D point cloud analysis in the context of forestry and horticulture.

LUKAS WINIWARTER received the B.Sc. and M.Sc. degrees (Hons.) in geodesy and geoinformatics from TU Wien, Vienna, Austria, in 2016 and 2018, respectively, and the Ph.D. degree (Hons.) in geoinformatics at Heidelberg University, Germany, in March 2022.

Until May 2022, he was worked as a Research Assistant with Heidelberg University. Since then, he has been working as a Visiting Postdoctoral Fellow with The University of British Columbia, Vancouver, Canada, funded by an Erwin-Schrödinger-Fellowship. He is also investigating the role of uncertainty in forest remote sensing using LiDAR. His research interests include application of machine learning on 3D point clouds and uncertainty for geospatial analyses.

BERNHARD HÖFLE received the Ph.D. degree (Hons.) in geoinformatics and physical geography from the University of Innsbruck, Austria, in 2007.

He was a Junior Professor of GIScience and 3D spatial data processing with the Institute of Geography, Heidelberg University, Germany, from 2011 to 2017. In 2017, he became a Full Professor of 3D spatial data processing with the Institute of Geography and the Head of the Interdisciplinary Center for Scientific Computing, 3DGeo Research Group, Heidelberg University. His research interests include development of computational methods for 3D geospatial data processing and extraction of geoinformation from 3D/4D point clouds acquired by different sensor systems and platforms.



FRANCISCO F. RIVERA is currently a Full Professor with the University of Santiago de Compostela, Spain. Throughout his career, he has supervised researches and published extensively in the areas of computer-based applications, parallel processing, and computer architecture. His current research interests include compilation of irregular codes for parallel and distributed systems; the analysis and prediction of performance on parallel systems and the design of parallel algorithms; and memory hierarchy optimisations, GIS, image processing, and 3D point clouds computing.