Adaptive Collision Culling for Massive Simulations by a Parallel and Context-Aware Sweep and Prune Algorithm

Gabriele Capannini and Thomas Larsson

Abstract—We present an improved parallel Sweep and Prune algorithm that solves the dynamic box intersection problem in three dimensions. It scales up to very large datasets, which makes it suitable for broad phase collision detection in complex moving body simulations. Our algorithm gracefully handles high-density scenarios, including challenging clustering behavior, by using a double-axis sweeping approach and a cache-friendly succinct data structure. The algorithm is realized by three parallel stages for sorting, candidate generation, and object pairing. By the use of temporal coherence, our sorting stage runs with close to optimal load balancing. Furthermore, our approach is characterized by a work-division strategy that relies on adaptive partitioning, which leads to almost ideal scalability. In addition, for scenarios that involves intense clustering along several axes simultaneously, we propose an enhancement that increases the context-awareness of the algorithm. By exploiting information gathered along three orthogonal axes, an efficient choice of what range query to perform can be made per object during run-time. Experimental results show high performance for up to millions of objects on modern multi-core CPUs.

Index Terms—Collision detection, Simulation, Parallel algorithms, Multicore processing, Multithreading, Tree data structures, Sorting

1 INTRODUCTION

Many applications in computer graphics, animation, and visualization include different forms of multi-body or n-body simulations. Such simulations are also used to study various types of phenomena in computational branches of physics, chemistry, and biology [1]. In large-scale scenarios, a particularly challenging and crucial feature is the ability to detect collisions in real-time. In this paper, a scalable solution is developed which can be used as a general building block in simulations of millions of bodies.

The Sweep and Prune (SaP) algorithm has proven to be particularly useful in practice as a coarse grained collision culling method [2], [3], [4]. By sorting the bounding boxes of the objects along one or several axes, and examining their overlap status by sweeping, it outputs the set of colliding box pairs. The algorithm aims to quickly eliminate a vast majority of all the \( n(n - 1)/2 \) possible box pairs. As such, the SaP method is an example of a top-level or broad phase collision search [5].

Other competing algorithms that can be used to determine a similar spatial ordering of the objects often rely on spatial subdivision using data structures such as BSP trees, \( k \)-d trees, octrees, and bounding volume hierarchies [6], [7], [8]. Bucketing approaches based on uniform grids, non-uniform grids, and spatial hashing have proven to be useful as well [9], [10]. These techniques aim to localize collision searches to gain performance. Uniform subdivision is the simplest, but its effectiveness is severely reduced when objects of widely varying sizes are used [11]. Recently, several parallelization strategies for broad phase collision detection on both multi-core CPU and many-core GPU architectures have been considered. These methods mainly focus on SaP, uniform grid partitioning, and brute force testing [12], [13], [14], [15], [16]. To handle large collections of ellipsoids, a parallel GPU-based solution has been proposed, which is based on binning at the outer level [17].

The purpose of the broad phase is to find the set of colliding box pairs. Usually, since the boxes are used to approximate more complex and arbitrarily shaped objects contained inside them, this output represents a potentially colliding set, where each pair of objects has been found to be sufficiently close to each other to warrant a more detailed investigation of their overlap status. In such cases, a narrow phase follows which produces the exact colliding set by more fine-grained checks [5]. In fact, more or less the same data structures can be used also in this case. It is the division into a two-phased approach that has proven to be very practical. Each phase can then be implemented and optimized for more specific operational circumstances. A commonly recommended combination is to use SaP for the approximate broad phase testing, and bounding volume hierarchies for exact narrow phase testing [4], [18], [19].

What combination of algorithms and data structures that gives the best overall performance in multi-body simulations is an open and challenging research question, which is complicated by the fact that the best choices are often application and scenario dependent. For a broader picture of the collision detection problem, interested readers can consult published surveys (e.g., [20], [21], chap. 2 in [22], and [23]).

In our EGPGV 2016 paper [24], we present a parallel SaP algorithm based on a double-axis sweeping approach and a succinct cache-friendly tree structure for fast range queries. This article is an extended version with the follow-
ing additional contributions: (i) A more efficient version of our data structure (aka SuccTree) which reduces the complexity of the range query operation. (ii) An improved parallel sorting algorithm which exploits temporal-coherence to quickly distribute the workload to the threads. (iii) An enhanced approach which involves all three coordinate axes in the pruning process to further reduce the number of false positives. (iv) A thorough experimental evaluation of the proposed solution, including scalability tests, more complex simulation scenarios, and comparisons to other competing methods using both synthetic and real-world benchmarks.

Altogether, the proposed strategies lead to a fast parallel collision-culling algorithm. In particular, our method stands out as an adaptive CPU-based algorithm that is able to handle complex scenarios with millions of moving objects at interactive rates without introducing an external subdivision step.

2 Prior work

David Baraff introduced the “Sort and Sweep” algorithm [25], [26]. It works by projecting the bounding boxes onto one of the main axes to obtain an array of lower and upper interval bounds. Intersecting interval pairs can then be determined by sorting the array and tracking the interval overlaps by a scanning procedure. When an overlap is found, the full box-box overlap status is determined by considering also the other two axes. It is claimed that insertion sort can be used to re-sort the array for each discrete frame of the simulation, since by temporal coherence, the array is kept almost sorted. However, when the temporal coherence is lost, the sorting turns into a major bottleneck. Second, the sorting turns into a major bottleneck. Second, the number of swaps used in the sorting.

Interestingly, there are theoretical results in computational geometry showing that the box intersection problem can be solved in worst-case $O(n \log n + k)$ time [27].

Another early example of sweep and prune is presented for the I-Collide system [2]. In this realization, three lists, one for each principal coordinate axis, are kept sorted using insertion sort, and the swaps trigger changes in a table of $O(n^2)$ bit flags that represents the overlap status of all possible pairs of intervals. While this may be efficient in some cases, the storage cost prohibits large-scale simulations.

Unfortunately, these classical SaP methods suffer from two drawbacks. First, the sorting is performed using insertion sort in $O(n)\tau$ time under the assumption that the arrays are almost sorted. However, when the temporal coherence is lost, the sorting turns into a major bottleneck. Second, in large scale simulations, the number of false positives along sweep axes tends to be superlinear in $n$ leading to unacceptable performance breakdowns [16], [28].

To illustrate, consider equally-sized cubes regularly spaced side by side in a dense simulation environment. Clearly, there are zero overlapping box pairs. Projection onto a single axis (either $x$, $y$, or $z$) will give clusters of overlapping intervals, with $n^{1/3} \times n^{1/3} = n^{2/3}$ intervals in each cluster. Thus, if we for each one of the $n$ objects examine $n^{2/3}$ projected intervals during sweeping, we end up with $O(n^{5/3})$ performance. For uniformly distributed moving objects in a dense environment, the situation more or less stays the same. The density of the endpoints along the axis means the number of false positives is of the same order, i.e., in expectation, a projected end point will belong to $O(n^{2/3})$ intervals. Traditional sweep and prune is not able to overcome this unfortunate growth of false positives. In such challenging simulations, excessive swapping behavior is observed that leads to $O(n^{5/3})$ time complexity in the three-dimensional case [29]. This clearly points to the need for extended SaP approaches. Many efforts have been made to find new solutions. Terdiman gives an overview of different SaP methods, including some improvements that addresses these issues, such as Multi-SaP, which combines SaP with spatial subdivision [30].

Several parallel solutions have been described in the literature as well. Batista et al. [12] presented a CPU-based parallel algorithm for computing $d$-dimensional axis-aligned box intersection inspired from the sequential version already used in CGAL [31]. The parallel brute-force approach proposed by Avril et al. [16] optimizes the concurrent accesses to shared data to speed up the broad phase computation on multi-core CPUs. Since the pairwise box intersection can be expressed in a data-parallel fashion, many solutions have been also proposed for GPUs. Liu et al. [13] select the preferred sweep direction based on the principal component analysis as well as spatial subdivisions to reduce the number of false positives during the sweep phase. Mainzer and Zachmann use a spatial subdivision based on fuzzy clustering [32]. The GPU-based solution presented by Lo et al. [14] aims to divide the space into optimally sized cells. The solution proposed by Geleri et al. [15] extensively uses the Thrust library [33] to speed up the SaP computation.

3 Sequential Double-axis SaP

This section briefly presents our sequential double-axis SaP approach [34] and gives some details on the used data structures and terminology. In what follows, we explain the idea behind our method in terms of the well-known SaP approach by Baraff [25] referred to as single-axis SaP.

In general, SaP deals with finding overlapping pairs of objects in moving-body simulations. Each simulated object is bounded by an axis-aligned box (AABB). Let $i$ denote the projection of the AABB related to a given object on a coordinate axis. We consider $i$ as a closed interval represented by an ordered pair of values $[i^-, i^+]$ with $i^+ < i^-$, also called low-endpoint and high-endpoint, respectively. Clearly, two AABBs collide iff the corresponding projections overlap on all the three coordinate axes.

The single-axis SaP acts by sweeping the sorted endpoints of one coordinate axis. During this process, when a low-endpoint is picked, the corresponding object turns into the active state and it is added to a data structure called activelist. An object remains active until the corresponding high-endpoint is found, then it is removed from the activelist. Since the objects simultaneously stored in the activelist overlap on the sweeping axis, when one of them is going to be removed, the algorithm checks if such an object collides with the others collected in the data structure. This is done by testing the overlap status on the remaining
coordinate axes and, when the test is positive, the pair is added to the set of collisions.

Differently, the double-axis approach splits the computation in two sweeping phases performed on two different coordinate axes. We firstly define the smallest interval on the first axis in which an object overlaps the projections of the other objects (called candidates). Then, we sort and sweep the second axis to discover the colliding pairs. In contrast to the single-axis SaP, where the full activelist is tested for each object, we check only the part of the activelist belonging to the candidate interval of each object. This reduces the number of tests and speeds up the computation.

Algorithm 1 Sequential double-axis SaP.

```
Algorithm 1 Sequential double-axis SaP.

input: Ω = \{0, …, n − 1\} \hspace{1cm} \triangleright \text{object ids}
input: \{X, Y, Z\} \hspace{1cm} \triangleright \text{endpoints on the coordinate axes}
output: C \hspace{1cm} \triangleright \text{pairs of colliding objects}
1: L[\_], U[\_] \hspace{1cm} \triangleright \text{candidate interval boundaries}
2: R[i], R^{-1}[i] \hspace{1cm} \triangleright \text{rank function and its inverse}
3: S \leftarrow ∅ \hspace{1cm} \triangleright \text{n-sized SuccTree}
4: c \leftarrow 0 \hspace{1cm} \triangleright \text{rank counter}
5: I_x \leftarrow \text{idxSort}(X) \hspace{1cm} \triangleright \text{sweep primary axis}
6: \text{for each } i \in I_x \text{ do}
7: \hspace{1cm} \text{if } i < n \text{ then}
8: \hspace{1cm} \text{R[i] } \leftarrow c \hspace{1cm} \triangleright \text{assign rank}
9: \hspace{1cm} \text{R^{-1}[c] } \leftarrow i \hspace{1cm} \triangleright \text{set inverse of rank}
10: \hspace{1cm} \text{S.ins}(R[i]) \hspace{1cm} \triangleright \text{activate } i \text{ by means of } R[i]
11: \hspace{1cm} \text{L[i] } \leftarrow \text{S.min(\_)}
12: \hspace{1cm} c \leftarrow c + 1 \hspace{1cm} \triangleright \text{next object rank}
13: \hspace{1cm} \text{else}
14: \hspace{1cm} \text{U[i−n] } \leftarrow c \hspace{1cm} \triangleright \text{i−n is the object id}
15: \hspace{1cm} \text{S.del}(R[i−n])
16: \hspace{1cm} C \leftarrow ∅
17: \text{I_y } \leftarrow \text{idxSort}(Y) \hspace{1cm} \triangleright \text{sweep secondary axis}
18: \text{for each } i \in I_y \text{ do}
19: \hspace{1cm} \text{if } i < n \text{ then}
20: \hspace{1cm} \text{for each } r \in S : L[i] \leq r < U[i] \text{ do}
21: \hspace{1cm} \text{j } \leftarrow \text{R^{-1}[r]} \hspace{1cm} \triangleright \text{retrieve candidate id}
22: \hspace{1cm} \text{if Test}_2(i,j) \text{ then } C \leftarrow C \cup \{(i,j)\}
23: \hspace{1cm} \text{S.ins}(R[i])
24: \hspace{1cm} \text{else}
25: \hspace{1cm} \text{S.del}(R[i−n])
26: \text{return } C
```

Algorithm 1 describes in more details the double-axis approach. At Lines 5 and 17, we sort the endpoints related to the primary and secondary axis, denoted by X and Y. To this end, we use a variant of the stable Least Significant Digit Radixsort (LSDR) which returns the indexes of the sorted items instead of permuting the input array. In particular, the 2n endpoints of each axis are stored in an array such that, for each object \(i \in \Omega = \{0, …, n − 1\}\), the corresponding \(i^+\) and \(i^-\) are placed at position \(i\) and \(i+n\) of each array, respectively. During the sweeping phases, let \(i \in [0, 2n]\) be an index picked from an array \(I\) returned by the sorting function, we can easily retrieve the type and object id of the corresponding endpoint by comparing \(i\) and \(n\), namely, if \(i < n\) then the index denotes \(i^-\) otherwise it denotes \((i−n)^−\).

The first phase is computed in the loop on Line 6. Here, for every object \(i \in \Omega\), we compute the rank \(R[i]\) (defined as the number of low-endpoints preceding \(i^+\) in the array \(I_x\)), the inverse function \(R^{-1}[\_]\), and the boundaries of its candidate interval, i.e., \(L[i]\) and \(U[i]\). In particular, ranks are used to represent the objects in the activelist \(S\) in both the phases, \(L[i]\) denotes the earliest object stored in \(S\) when \(i^+\) is picked (i.e., the minimum rank stored in \(S\)), and \(U[i]\) denotes the least object \(j\) following \(i^-\) such that \(j^+ > i^-\) (see the example in Figure 1).

The second phase begins on Line 18 and detects the pairwise collisions. For each \(i^+\) picked from \(I_y\), the interval \([L[i], U[i]]\) is used by the range query on Line 20 to retrieve the subset of \(S\) to check. On Line 22, for every rank \(r\) returned by the query, a box-to-box test is performed between \(i\) and the object \(j = R^{-1}[r]\) by checking the axes \(X\) and \(Z\). Note that \(U[i]\) is excluded by the query since the lower-endpoint of the corresponding object is greater than \(i^-\) on the primary axis. Furthermore, if an object \(j\) belongs to the candidate interval of \(i\), then \(j^+ < i^-\) on the X axis so that, on Line 22, we also need to test if \(i^+ < j^-\) (see Figure 1).

The double-axis approach is supported by a succinct data structure called SuccTree for storing the active objects. The required operations are: insertion and deletion, to find the minimum, and range query. Assuming the w-bit word-RAM model, let \(S \subseteq \{0, …, n − 1\}\) be the set of values to represent, the corresponding SuccTree consists of a w-ary tree with \(n\) leaves. Each level is implemented by an array of bits so that each bit is associated to a node and denotes the presence of the node in the tree. In particular, the leaves represent the values stored in \(S\), therefore \(i\) belongs to \(S\) iff the bit at position \(i\) of the bottom array is set. A node in the upper levels, instead, is used to query the SuccTree and it is set iff it has at least one child. To this end, the parent \(p(i)\) of the node \(i\), is the bit at position \([i/w]\) one level up.

According to these rules, to add or remove a value, we set the corresponding bit at the bottom level of the tree, then we update its ancestors if required. The minimum, assuming that \(S\) is not empty, corresponds to the leaf reached from the root by passing through the least child at every level. Range queries are implemented by means of the succ operation which returns the position of the least leaf greater than \(i\). This is found by searching the lowest ancestor of \(i\) having a nonempty set \(G\) of greater siblings. Then, we follow the path to the least leaf descending from min(\(G\)). All these operations act by traversing at most two paths in the tree so that their complexity is \(O(\log_w n)\). This is possible since SuccTrees use the CPU bitwise instruction set to exploit bit-level parallelism and perform operations on \(w\) bits at a time that, otherwise, have \(O(w)\) complexity. Finally, the range query complexity is \(O(\ell \log_w n)\) since the \(\ell\) returned values correspond to the same number of succ invocations.
4 Improved Range Queries

During the second phase of Algorithm 1, the complexity of the range queries dominates the computation. In what follows, we introduce a faster way to perform this operation that reduces the complexity of each query by accessing the tree only once.

To this end, each leaf of the SuccTree is linked to the next one by means of the array nexts[] so that nexts[i] corresponds to the value succ(i). When an item is added or removed, the tree and the array are updated. In particular for the insertion case, we defined the dual of the succ(i) operation, namely prec(i), which locates the greatest leaf preceding i in the SuccTree. It traverses the tree as succ() does, but in the opposite direction, so that its complexity is O(log n). Once the position of the item preceding i is known, we can easily update nexts[] as shown in Algorithm 2. Note that the overall complexity of the insertion is the same as in the original SuccTree.

Algorithm 2 LSuccTree: insertion.

1. S.ins ⊿ v
2. p ← S.prec(v)
3. nexts[v] ← nexts[p]
4. nexts[p] ← v

At query time, by means of nexts[], we jump from an active object to the next one bypassing the SuccTree. Nevertheless, we still need to access the tree to locate the first array value to return. As a consequence, the new range query complexity is O(ℓ + log n).

In the rest of the paper, the new data structure, combining the SuccTree and the nexts[] array, is referred as LSuccTree, which stands for Linked SuccTree. LSuccTrees replace SuccTrees in the pairing phase of both the sequential and parallel versions, while SuccTrees are still used during the first phase since range queries are not required. The main drawback of a LSuccTree is the amount of memory required to represent the array. Since the maximum representable value is n−1, which needs only O(log n) bits to be stored, O(n log n) bits are enough. However, in our implementation, we allocate an array of O(wn) bits.

5 Parallel Double-axis SAP

Here we describe the parallel version of the approach shown in Sections 3 and 4. We divide Algorithm 1 in three parts which are treated separately: sorting on Lines 5 and 17, generation of the candidate intervals in the loop on Line 6, and the pairing phase in the loop on Line 18. In the preamble of the algorithms presented for each part, only the variables which do not appear in Algorithm 1 are shown. Finally, we assume that the number of objects, n, is multiple of the number of processors, m.

5.1 Sorting

Our parallel sorting approach makes use of temporal coherence to keep on splitting the input arrays in (almost) equally sized distinct partitions that can be sorted simultaneously and independently. To this end, for each axis to sort, a set B of boundaries is used for mapping endpoints to partitions during the initial setup. Once the endpoints have been partitioned, we sort every partition by means of the sequential LSDR used in Algorithm 1. At the end of each sorting call, the set of boundaries B is updated as an attempt to keep the partition sizes balanced for the next iteration.

In [24], the setup is done by calculating the formula on Line 5 of Algorithm 3 that costs of O(m) operations for each one of the 2n endpoints. Considering that, in two consecutive iterations, the set of items in a partition usually does not change drastically, we find it suitable to update the partitions incrementally throughout the entire simulation. In practice, we store the endpoints of each partition through the SaP iterations and, during the setup, we test if each endpoint still belongs to the same partition as in the previous iteration (according to the current B values and the new endpoint value). If it does not, we move the endpoint into the correct partition by recalculating the previous formula.

In this way we can save most of the time spent in the setup in [24] since checking the partition can be done in constant time. Nevertheless, the complexity is unvaried since, in the worst case, all the endpoints needs repositioning.

Algorithm 3 Parallel sorting.

1: for each i ∈ [1, m] in parallel do
2: for each x ∈ P_i do
3: if not b_i−1 ≤ x < b_i then
4: P_i ← P_i \ {x}
5: i' ← 1 + ∑_{j=1}^{m-1} int(x > b_j)
6: P_i ← P_i ∪ {x}
7: for each i ∈ [1, m] in parallel do
8: S_i ← idxSort(P_i)
9: for each i ∈ [1, m−1] do
10: b_i ← X[1 \cdot (2n/m)]
11: return B

Algorithm 3 shows in more details the procedure for sorting the X axis. The computation for the other axis is identical except for the boundaries B and the partitions P which are independently defined for each set of endpoints to sort. Let B = {b_0, ..., b_m} with b_0 = −∞ and b_m = +∞, X is kept divided into m partitions P_i such that P_i = {x ∈ X : b_i−1 ≤ x < b_i}.

The setup is performed in parallel by distributing P_i among the m processors in the loop on Line 1. Here, to avoid costly synchronizations among the processors, each P_i is arranged as a list of m distinct arrays. Hence, the set P_i is implemented by a matrix of m × m buckets where each P_i is associated to a column, as depicted in Figure 2. The setup is then performed by assigning a row of such a matrix to each processor. In this way a processor has exclusive access to m buckets (one for each P_i) and can freely move endpoints among the partitions. Once the setup is ended, each processor sorts the endpoints related to a column of
the matrix (Line 7). The output \( I \) is stored in \( m \) contiguous arrays \( I_i \) (Line 8) such that \( |I_i| = |P_i| \).

![Fig. 2. Sorting partitions are arranged as matrix of \( m \times m \) buckets.](image)

The final part of the algorithm updates \( B \) for the next iteration. It is done by using the endpoints referred by the indexes in \( I \) stored at distance \( \lfloor 2n/m \rfloor \) to each other (Line 10). On the first iteration, \( B \) is initialized by computing the span of \( X \) as \( \text{span}(X) = \max(X) - \min(X) \), and setting \( b_i = i \cdot \text{span}(X)/m + \min(X) \) for each \( i \in [1, m] \).

### 5.2 Candidates Generation

In the sequential case, the candidate intervals and the object ranks are computed in the loop on Line 6 of Algorithm 1. These values are function of the set of active objects \( S \) which is modified, in each iteration of the loop, by adding or removing a value. Thus, the set of values stored in \( S \) in any iteration depends on the previous ones. In what follows, we show how to compute the object ranks \( R[i] \), its inverse \( R^{-1}[i] \), and the candidate intervals (i.e., \( L[i] \) and \( U[i] \)) in parallel.

To parallelize such a loop, we split the \( 2n \) iterations of the sequential loop into \( m \) contiguous parts to be computed by \( m \) processors. To this end, our preliminary goal is to identify the values stored in \( S \) every \( 2n/m \) iterations starting from the first one. Once this information is ready, the processors sweep the assigned part of endpoints and independently compute the related candidate boundaries.

Algorithm 4 briefly shows the salient parts of the parallel approach. First, we compute the object ranks by means of the following steps: dividing the set \( I \) of sorted indexes into \( m \) equally-sized intervals \( I_{1..m} \) (Line 3); counting the low-endpoints in each partition (loop at Line 5); computing a prefix sum on such values (Line 7); calculating the object ranks of each partition (Line 10). Once the ranks are computed, each processor \( p \) instantiates its private SuccTree, \( S_p \), and flips the values in \( S_p \) according to the objects picked (Line 15). This is done by using the \( \text{flip}(r) \) operation which adds the rank \( r \) to \( S_p \) if \( r \notin S_p \) otherwise \( r \) is removed from \( S_p \) (note that to check if \( r \in S_p \) can be done in constant time by testing the corresponding \( S_p \) leaf). Once it ends, for each \( p \), \( S_p \) contains the ranks of the objects having exactly one endpoint in \( I_p \). Now, we use the binary symmetric difference operator, which is defined as: \( S \oplus S' = (S \cup S') \setminus (S \cap S') \), to merge each SuccTree \( S_p \) with its predecessors \( S_{p'} \). Since the \( \oplus \) operator is associative, the merging step is done on Line 18 by computing the parallel all-prefix-sums of the SuccTrees, inspired by the tree-based implementation in [35]. Each resulting \( S_p \) contains the ranks of the objects \( i \) such that \( i^- \) belongs to a partition \( I_{p'} < p \) and \( i^+ \) belongs to a partition \( I_{p'} > p \). In other words, each \( S_p \) stores the active object ranks at the point of the computation when \( I_p \) begins, which is our goal. Once the all-prefix-sums operation is completed, the processors compute the remaining data structures in parallel similarly as done in Algorithm 1.

#### 5.3 Pairing Phase

The pairing phase dominates the run-time, in particular, in the most challenging scenarios with high density. As a consequence, parallelizing the loop on Line 18 in Algorithm 1 is the most crucial point of this work.

Our solution can be classified as an object partitioning technique. In particular, Algorithm 1 has been parallelized by adaptively dividing the space along the primary axis in non-uniform parts made up of an equal number of objects. This division is calculated directly at the beginning of the pairing phase by splitting the set of candidate intervals. An important side effect of this approach is that the computation of longer intervals (e.g. due to clustered objects) is “spread” on more processors so as to improve the workload balance and the overall throughput.

Algorithm 5 describes the parallel pairing phase. The set of \( n \) objects is divided into \( m \) partitions that are computed in parallel. To this end, we compute the size of a partition, \( \Delta = n/m \), and the partition boundaries, i.e., \( \beta = (p-1)\Delta \) and \( \epsilon = p\Delta \), so that the objects of which rank belongs to \([ \beta \Delta, \epsilon \Delta ] \) are assigned to the processor \( p \). In the loop on Line 7, each processor sweeps the whole array \( I \) which contains
the sorted endpoints related to the secondary axis. Here, each processor keeps track of its active objects in a private LSuccTree $S_p$. Unlike Algorithm 1, where $S$ can store up to $n$ values, the capacity of each $S_p$ is $\Delta$. Thus, for each $p$, the range of values $[\beta, \epsilon)$ to be handled within $S_p$ is mapped into $[0, \Delta)$ by using $\beta$ as an offset, as it is done on Lines 15, 17, and 19. For each low-endpoint $i$ picked from $I$, the processor $p$ discovers the collisions of $i$ with the active objects of which rank belongs to $[L[i], U[i]) \cap [\beta, \epsilon)$. If this intersection is not empty (Line 11), the candidate interval is clipped to fit in the partition (Lines 12 and 13) before the processor can perform the usual range query (loop on Line 14) by means of the local boundaries, $l$ and $u$.

Algorithm 5 SaP – Parallel pairing phase.

```
input: $I$ \hspace{1cm} \triangleright 2n sorted endpoint indexes
input: $m$ \hspace{1cm} \triangleright number of partitions
1: $\Delta \leftarrow n/m$ \hspace{1cm} \triangleright partition size
2: $C \leftarrow \varnothing$ \hspace{1cm} \triangleright collision set
3: for each $p \in [1, m]$ in parallel do
4: \hspace{1cm} $S \leftarrow \varnothing$ \hspace{1cm} \triangleright $\Delta$-sized LSuccTree
5: \hspace{1cm} $\beta \leftarrow (p-1)\Delta$ \hspace{1cm} \triangleright partition begin
6: \hspace{1cm} $\epsilon \leftarrow p\Delta$ \hspace{1cm} \triangleright partition end
7: for each $i \in I$ do
8: \hspace{1cm} if $i < n$ then
9: \hspace{2cm} $l \leftarrow L[i]$
10: \hspace{2cm} $u \leftarrow U[i]$
11: \hspace{2cm} if $l < \epsilon \wedge \beta \leq u$ then
12: \hspace{3cm} if $\beta < l$ then $l \leftarrow l-\beta$ else $l \leftarrow 0$
13: \hspace{3cm} if $u < \epsilon$ then $u \leftarrow u-\beta$ else $u \leftarrow \Delta$
14: \hspace{2cm} for each $r \in S : l \leq r < u$ do
15: \hspace{3cm} $j \leftarrow R^{-1}[r+\beta]$
16: \hspace{3cm} if $\text{Test}_{xz}(i, j)$ then $C \leftarrow C \cup \{(i, j)\}$
17: \hspace{3cm} if $\beta \leq R[i] < \epsilon$ then $S.ins(R[i]-\beta)$
18: else
19: \hspace{2cm} if $\beta \leq R[i-n] < \epsilon$ then $S.del(R[i-n]-\beta)$
```

Note that each object is stored in the LSuccTree of one processor only (see Lines 17). Let $i$ be an object and $R[i]$ belong to the partition of the processor $p$. Each processor $p'$ ($\neq p$) detects the objects in $S_{p'}$ colliding with $i$ when $i^*$ is picked from $I$, but only the processor $p$ can detect the collision between $i$ and any object $j$ ($\neq i$) when $j^*$ is picked. Hence, there are no duplicates in the final set $C$ of collisions, and no reduction phase is needed at the end of the procedure. Finally, the implementation of Algorithm 5 is synchronization free, which means that no inter-thread communication or barriers are required to maintain the data integrity. In the case above, for example, depending of the workflow of each processor, $p'$ is able to discover a collision between $i$ and an object in $S_{p'}$ even if the processor $p$ has not yet reached $i^*$.

6 Dynamic Choice of Sweeping Axes

The double-axis technique is able to speedup the SaP computation by reducing the number of box-box overlap tests substantially [24], [34]. A first axis is used for candidate generation, and a second one is used to perform range queries. However, the remaining third axis is excluded, which means it is only used as necessary in the inner test to determine the overlap status of a pair of boxes.

The key point of the parallel sort discussed in Section 5.1 is to almost fairly divide the workload among the processors due to temporal coherence. However, there are certain cases where this approach is not effective. For example, when endpoints get severely clustered along a sorting axis, the position of many of them can coincide so that many values can gather in one partition, which unbalances the workload by degrading the throughput of the entire computation.

Section 6.1 provides a possible heuristic that is based on switching one of the two currently used axes with the third one during the sorting phase. In this way, an axis that is detected as inefficient becomes the excluded one. As an alternative, Section 6.2 proposes a technique which yields more stable performance by ordering all three coordinate axes and, then, dynamically selects at each iteration the best combination of axes to use.

6.1 Dynamic Axes Swapping (DAS)

When a sorting axis gets severely clustered, there is an opportunity to switch the most clustered axis with the unused one. To decide when to swap an axis is advantageous, we calculate a measure of relative data dispersion $D$. In particular, $D$ is computed in the sorting phase and it measures the unbalance of the number of elements per partition with respect to the ideal one. To this end, $D$ is defined as a function of the number of objects $n$, the number of partitions $m$, and the statistical variable $X$ describing the number of elements per partition (namely $x_i$ denotes the number of items in the partition $P_i$ defined in Section 5.1). Let $\delta$ be the mean absolute deviation which is calculated as the expected value $E[|X-\mu|]$ with $\mu = 2n/m$. By definition, $\delta$ is an absolute value while we need a measure which is comparative with the ideal partition size, i.e., $\mu$. Hence, we further divide $\delta$ by $\mu$ and we compute $D$ as follows:

$$D = \frac{\delta}{\mu} = \frac{\sum_{i=1}^{m} |x_i - \mu|/m}{2n/m} = \frac{\sum_{i=1}^{m} |x_i - \mu|}{2n} \quad (1)$$

In this way $D$ gives us a measure of the relative workload unbalance among the partitions. During the computation, when partitions turn out to be unbalanced on one of the two main axes (i.e., $D$ is greater than a given threshold), we switch that axis with the unused one. This operation requires to reinitialized the sorting partition boundaries $B$. To this end, we compute the span of the new axis (of which time complexity is $O(n/m)$ on a PRAM with $m$ processors) and reinitialize $b_1..m-1$, i.e., $b_i = i \cdot \text{span}(X)/m + \min(X)$ for each $i \in [1, m]$.

6.2 Continuous Axes Selection (CAS)

The following technique attempts to speed up the pairing phase by dynamically selecting the best pair of sweeping axes. To this end, the axes chosen for candidate generation and pairing phase should be the least clustered among the three principal axes. Hence, we select those that reduce the average number of active objects the most. The measure $D$ defined in Section 6.1 deals with the sizes of the sorting partitions, but it identifies a clustered scenario only when it gets severally clustered. Compared to $D$, Algorithm 6 gives
a more accurate overview of the clustering degree of an axis by sweeping the ordered list of endpoints and computing the average number of active objects.

Algorithm 6 Clustering degree calculation.
\begin{verbatim}
input: n \quad \triangleright\text{number of objects} \\
input: I \quad \triangleright\text{array of } 2n \text{ sorted indexes} \\
output: \bar{n} \quad \triangleright\text{average number of active objects} \\
1: m \leftarrow 0 \quad \triangleright\text{number of active objects} \\
2: s \leftarrow 0 \quad \triangleright\text{sum of active objects} \\
3: \text{for each } i \in I \text{ do} \\
4: \quad \text{if } i < n \text{ then } m \leftarrow m + 1 \text{ else } m \leftarrow m - 1 \\
5: \quad s \leftarrow s + m \\
6: \text{return } s/n \\
\end{verbatim}

Let \( n \) be the number of objects, Algorithm 6 returns a measure in the range \([1, n]\) of the current clustering, where \( n \) denotes the most clustered scenario for the current axis, while 1 denotes the zero-overlaps scenario. Once Algorithm 6 has been performed for all the three sorted axes, we proceed by computing the candidate intervals on the least clustered axis and sweeping the axis with the second best score in the pairing phase.

In the current implementation, we apply this technique in each frame of the simulation, but we could reduce the related overhead by selecting the axis once every 10th frame or so.

7 Speculative SaP

The double-axis approach with the best sweeping axes can efficiently handle scenarios having the objects clustered along one coordinate axis (see Figure 11) since it is enough to exclude the corresponding dimension to avoid the problem. A different and even more challenging case is when the objects are distributed in such a way that several large clusters along different axes arise within the same time frame. In such a case, the previous strategies can lose their effectiveness. In what follows, we present a further strategy that attempts to overcome this issue.

The proposed technique exploits information from all axes to make a more fine-grained selection of how to proceed. Every axis is swept and a selection is made to pick the best candidate interval for each object before the range query is carried out using the pairing axis.

The whole procedure is described by Algorithm 7 which makes use of the parallel building blocks presented in the previous sections. In particular, for each coordinate axis, we use the parallel sort on the corresponding set of endpoints (Line 2). Then, we compute the average number of active objects using Algorithm 6 (Line 3). Once all the axes have been scored, the one with the lowest score \( s_a \) is selected for the pairing phase (Line 4) and the last two as candidate axes \( a_1, a_2 \) (Line 5). Such an initial overhead (from which comes the name speculative) is justified by the boost of performance we likely achieve during the pairing phase by reducing the number of box-box overlap tests to perform. In particular, once the two sets of candidate intervals have been computed (Lines 6 and 7), we perform the pairing phase twice by alternatively using one candidate axis. This is done by splitting the set of objects \( \Omega \) in two subsets, \( \Omega_1 \) and \( \Omega_2 \), so that each object is assigned to the favorite one on the base of its candidate interval length (Lines 8 and 10). In particular, since a candidate interval is defined by the pair of values \( L[i] \) and \( U[i] \) for each object \( i \), we simply define the length of the interval \( c(i) \) as \( |c(i)| = U[i] - L[i] \) (Line 8). Clearly, the suggested technique is more context-aware, since it exploits knowledge about the specific situation per object, rather than per frame. However, this is a heuristic approach and it does not ensure that the range query performed on the shortest interval is the fastest, since this aspect also depends on the number of active objects intersecting an interval, and this information is known only at query-execution time.

Algorithm 7 Spectulative SaP.
\begin{verbatim}
input: \Omega = \{0, \ldots, n - 1\} \quad \triangleright\text{object ids} \\
input: \{X, Y, Z\} \quad \triangleright\text{coordinate axes} \\
output: C \quad \triangleright\text{pairs of colliding object ids} \\
1: \text{for each } a \in \{X, Y, Z\} \text{ do} \\
2: \quad \text{doSorting}(a) \\
3: \quad s_a \leftarrow \text{noAvgActive}(a) \quad \triangleright\text{score of axis } a \\
4: \quad a_0 \leftarrow \text{arg min}_a \{s_a \} \quad \triangleright\text{pairing axis} \\
5: \quad \text{let } \{X, Y, Z\} \setminus \{a_0\} \mapsto \{a_1, a_2\} \quad \triangleright\text{candidate axes} \\
6: \quad c_1 \leftarrow \text{doCandidate}(a_1) \\
7: \quad c_2 \leftarrow \text{doCandidate}(a_2) \\
8: \quad \Omega_1 \leftarrow \{i \in \Omega : |c_1(i)| < |c_2(i)|\} \\
9: \quad C \leftarrow \text{doPairing}(a_0, a_1, \Omega_1) \\
10: \quad \Omega_2 \leftarrow \Omega \setminus \Omega_1 \\
11: \quad C \leftarrow C \cup \text{doPairing}(a_0, a_2, \Omega_2) \\
12: \text{return } C \\
\end{verbatim}

As a finally remark, even if the pairing phase is performed twice on different candidate axes, only one range query is performed per object so that no duplicates among the detected collisions are generated.

8 Experiments

The experiments presented in this section were run on a dual 2.40 GHz Intel® Xeon® E5-2630v3 CPU having 16 cores with Hyper-Threading Technology (HTT) and 64 GB of RAM, and using Ubuntu 14.04 with gcc 5.4.1.

In the following, we present the results of running different simulations designed to compare performances of our revised parallel double-axis SaP and the version proposed in [24] (denoted as \textit{new} and \textit{old}, respectively). Both algorithms have been implemented by means of the OpenMP API for shared-memory parallel programming. We also show the scalability of our new solution by varying the number of threads \( p \) in a range from 2 to 32 (given that we are using a machine with 16 cores and HTT) and choosing a small (\( 2^{17} \)), medium (\( 2^{19} \)), and large (\( 2^{21} \)) problem size \( n \). Furthermore, the sequential algorithm, used for calculating the scalability, has been upgraded by means of LSuccTree described in Section 4 which yields a faster pairing phase.

In each simulation, \( n \) objects of varying sizes move freely in a 3D space delimited by a world cube. To challenge our approach, we let colliding boxes pass through each other and bounce on the world boundaries. In this way, we created highly clustered contexts with many deeply overlapping boxes although such scenarios are unrealizable in a simulation of non-penetrating objects. In the beginning, the boxes
were spread uniformly in space with density $d$ computed as the ratio of the sum of the boxes’ volumes to the volume of the world. In each iteration of a simulation, we firstly use SaP to discover the collisions, then the objects move. Each simulation consisted of 100 iterations and we validated the results by checking that the set of collisions discovered at each iteration was the same for all the compared algorithms. In the following, we present the results of each parallelized part: sorting, candidate generation, and pairing. Finally, we show the details of how these parts affect the overall performance.

8.1 Sorting Evaluation

Figure 3 shows the performance of the two different parallel algorithms used for sorting, i.e., new and old. Clearly, the more efficient setup that is used in the new version leads to better performance since it can skip the computation of the object-to-partition mapping for most of the endpoints. Nevertheless, when the size of the input grows, the time spent for sorting each partition becomes increasingly more dominant so that the performances of the two approaches tend to grow closer to each other.

In general, when many endpoints change partition frequently (e.g., if they move very fast w.r.t. the average span of the partitions), the complexity of the new setup grows since we have to recompute the partitions they belong more often. Such an overhead, however, is always bounded by the behavior of the approach in [24] which computes all of them at each iteration.

Figure 4 depicts the scalability of the new sorting phase. Here, the sequential algorithm is based on the LSDR algorithm that has a pseudo-linear complexity while the parallel method shown in Algorithm 3 mainly consists of two steps: setup and sorting (the time spent in updating the set $B$ is negligible and can be left out of this analysis). The time complexity of the setup is linear in the number of objects $n$ since the object partitioning is performed in parallel by $m$ threads and remapping one object in the correct partition requires $O(m)$ operations, as discussed in Section 5.1. However, when temporal coherence can be effectively exploited, the performance of the setup only affects the overall performance with a minor overhead. In the most favorable cases, this leads to a super-linear speedup since the parallel instances of LSDR work independently on each partition by fully exploiting the caches of all processors. Figure 4 shows the results related to only one object density (i.e., $d = 0.20$), since sorting is only slightly affected by this parameter.

Finally, we measured also the workload balance of our parallel method to evaluate the effectiveness of computing the boundary set $B$ by exploiting temporal coherence. To this end, we measured $D$, as defined in Section 6.1, in each frame of the simulation. The results showed that $D$ was always lower than 0.01, which means that the number of endpoints to sort assigned to the buckets missed the perfectly balanced size by less than 1%.

8.2 Evaluation of Candidates Generation

Figure 5 shows that the parallel generation of the object candidates reaches good speedups for the larger input sizes. As a matter of fact, the parallel calculation of the candidate intervals consists of several steps which incrementally calculate the candidate interval boundaries and the other values (i.e., rank $R_i$ and inverted rank $R^{-1}$ arrays). The thread barriers interleaving the different steps (as well as the $O(\log p)$ levels of the all-prefix-sum computation) and the run-time thread management introduce an overhead that cannot be completely recovered due to the fine granularity of each step, especially for small inputs.

As for sorting, even in this case, the figure shows the results related to just one object density (i.e., $d = 0.20$), since candidate generation is only slightly affected by this parameter.
8.3 Pairing Phase Performance

As shown in Figure 6, our adaptive partitioning approach exhibits a good scalability, which increases as the number of simulated objects and the density grow. This is mainly due to the fact that the computation is synchronization-free and no reduction phase is required at the end of the process. Last but not the least, the almost perfect load balancing observed in the experiment led to a better throughput since it minimized the average response time of each thread. In fact, we measured the thread workload in all simulations by collecting the percentages of the number of collisions detected by the various threads. The results showed small values of standard deviation, \( \sigma \), of the percentages, which means that workload was almost perfectly distributed among the threads. We also repeated the same tests by varying the number of threads from 4 to 64 and we obtained similar results, i.e., \( \sigma < 1.5\% \).

Moreover, we observed that, in high-density simulations, we obtained a super-linear scalability. This is a side-effect of the adaptive object-partitioning which acts as an early-exit condition in some cases. When long candidate intervals are spread out on more than one partition, each thread checks that the assigned part of the original interval is overlapping with its own partition (Line 11). When such a test fails, the current iteration ends immediately and no access is done to the L Suc t Tree. In the sequential case, instead, at least one access to the LSuccTree is made for every low-endpoint picked. Furthermore, the LSuccTree instances used in the parallel case are smaller (because only a fraction of the entire set of objects is managed by each of them) which reduces the complexity of the LSuccTree operations.

Finally, we measured the boost of performance related to the faster range query described in Section 4. Figure 7 shows the results of the experiment and the obtained speedup with respect to the previous version. The new data structure is able to remarkably speed up the computation by reducing the time spent for querying the active endpoints up to \( 1.5 \times \). In particular, higher speedup results are obtained when the density of the objects increases so that the number of coexisting active objects grow and their memory representation in the data structure becomes more dense. As a consequence, CPUs can better exploit data locality and the cache hierarchy.

8.4 Overall Parallel Performance

Figure 8 presents how each part of the computation affects the overall performance of both our parallel SaP versions. It shows the time spent for sorting the primary and secondary axes, generating the candidates, and pairing. In this case, we present the results of the experiment by varying the object density and for three different input sizes: \( 2^{17} \), \( 2^{19} \), and \( 2^{21} \) (roughly 131 thousands, half a million, and two millions). As mentioned before, the time spent in the first two phases remained almost constant in each chart since the object density \( d \) affects the performance of sorting and the candidate generation only slightly.

On the other hand, the pairing phase dominated the elapsed time of the entire algorithm. As the scenario became more dense, the time spent in this phase grew due to the increased number of overlapping intervals on the two coordinate axes selected for the candidate generation and pairing phases, which implied a higher number of box-box overlap tests to perform. Furthermore, the improved range query and the enhanced sorting algorithm (see Section 4 and Section 5.1 respectively) led to noticeable speedups, which varies in the range \( 1.3 - 1.8 \times \) as shown in Figure 8.

Figure 9 depicts the scalability of the overall SaP computation. If we consider how the different phases affect the overall performance (see Figure 8), the trends shown in Figure 9 are similar to the ones in Figure 6. The reason for this is that the pairing phase dominates the performance of the solution.

9 Clustering Scenarios

To further challenge our algorithm, two additional experiments were designed to examine the behavior under different kinds of clustering of the objects. In the first case, the objects were randomly positioned using a uniform distribution inside a large ball. The velocity vectors were directed towards the center of this ball with the speeds adjusted so to let each object reach the center simultaneously, should they not hit anything on the way. However, a collision response method was used to prevent objects from passing through each other. In this way, the objects formed a dense ball-shaped cluster before a massive number of collisions forced them to spread out themselves in all directions. The whole simulation was run for 500 frames.

To illustrate the motion, Figure 10 shows five screenshots. The graph under the images gives the collision detection times of our parallel algorithms together with the corresponding sequential run-time for \( n = 2^{19} = 524 \, 288 \). The plot has been cut to better visualize the performance of the parallel versions. The parallel speedup of the new algorithm during the most intense part of this scenario, i.e., frames 75–150, was \( 10.59 \times \) on average. The heaviest sequential iteration was 4.35 seconds long. The average time for all 500 frames was 67.65 ms which, compared to the average sequential run-time, gave a speedup of \( 6.8 \times \). When we compared the new solution to the old one, we observed that the average speedup is \( 1.3 \times \) with a peak equal to \( 1.7 \times \) at the frame corresponding to \( a \) in Figure 10.

In the next experiment, \( n = 750^2 = 562 \, 500 \) equally sized cubes were distributed in a large cube. To define the midpoints of the boxes, we used a regular grid spacing for
two of the coordinates, whereas the third coordinate, call it \( u \), was chosen at random to place the objects uniformly distributed above and below the plane \( u = 0 \). The motions of the objects were then chosen so to let all the objects pass through the plane \( u = 0 \) simultaneously in the middle of the simulation. An illustration of this scenario is given in Figure 11. The size of the cubes was set so that the cubes almost touched their neighbors while moving through the plane (see the middle image). This means that the entire scenario was free from collisions, but note that the clustering of the objects within a single plane can be a difficult case causing severe bottlenecks for certain algorithms.

This scenario is used to highlight the different behaviors of our two alternative approaches for the axes selection when applied to the new version of our parallel algorithm. The first one, described in Section 6.1, is called Dynamic Axis Swapping (DAS) and it is characterized by a custom initial configuration which is changed as one of the used axes becomes too clustered. The second method, described in Section 6.2, referred as Continuous Axis Selection (CAS), keeps all three coordinate axes sorted throughout the entire simulation and selects the two most promising axes, at each iteration, on the base of the number of average active objects. In particular the candidate axis corresponds to the axis with the best score and the pairing axis to the axis with the second best score (as described in Section 6.2).

Since our algorithm uses a double-axis approach, it is less sensitive to clustering in general compared to single-axis approaches. Nevertheless, if clustering occurs along either the primary or the secondary axis, a performance bottleneck occurs, given that the chosen sweeping axes remain fixed throughout the simulation. However, when we enable one of our heuristics such problems can be avoided in most cases. The performance depicted in Figure 11 show the obtained results when we ran our parallel algorithm with and without a dynamic axis selection technique. The algorithm used the \( x \)-axis as the primary axis, and the \( y \)-axis as the secondary axis to begin with. Without selecting an appropriate pair of axes, we saw that run-times increased substantially during the most intense part of the plane clustering in frames 225–275, referred as “baseline” in the figure. When our DAS heuristic was used, however, this problem was avoided completely. After the axis swap, which occurred at frame 214, the whole scenario ran at more or less a constant speed (about 30 ms per frame).

Differently, by paying a small overhead for the additional axis to sort, the CAS method is able to run SaP on the best axes combination while the DAS method opportunistically swaps the axis when it is required. In fact, the configuration chosen by the CAS technique turned out to use, from the first frame, the best configuration and the SaP maintains almost the same level of performance throughout the entire simulation except for the central part where it slightly slows down due to some workload-unbalanced sorting sessions. The DAS approach has in the beginning a non-optimal, but still good, configuration which includes the most problematic axis, i.e., \( x \). When such an axis becomes too clustered, it is changed with the third axis and

Fig. 6. Scalability of the pairing phase measured for different values of \( d \) and \( n \) by varying the number of threads \( p \).

Fig. 7. Elapsed time for the pairing phase for different \( d \) values by varying \( n \).
10 Speculative Solution Performance

The solution described in Section 7 can be very useful in the most complex scenarios where objects are clustered along different axes. In this section, we show the results related to such a scenario where half objects move on the plane $y = 0$ and the remaining ones on the orthogonal plane $z = 0$. Unlike the previous section, we computed the density $d$ as the ratio of the sum of the objects’ surface areas over the sum of the areas of the two planes.

Table 1 shows the average time spent for the different algorithmic phases (sorting, candidate generation, and pairing) by varying the number of objects and the density for the two tested algorithms. We compared the speculative solution with the non-speculative parallel algorithm which is configured for working with the best pair of axes during the entire simulation. In contrast to the speculative case, where the two remaining axes are concurrently used for alternatively querying the shortest candidate interval for each object, the competitor is forced to use only one axis. As a consequence, in the solution denoted as non-speculative, almost one half of the objects are computed by means of a long candidate interval which degrades the entire performance. On the other hand, to keep sorted all the axes and compute two sets of candidate intervals implies an overhead which is recovered especially for high-density large scenarios.

### Table 1

<table>
<thead>
<tr>
<th>$n$</th>
<th>$d$</th>
<th>$t_{s+t_c}$</th>
<th>$t_{p}$</th>
<th>$t = t_{s+t_c} + t_p$</th>
<th>$s$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{17}$</td>
<td>.05</td>
<td>7</td>
<td>11</td>
<td>18 (0.87×)</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>.35</td>
<td>7</td>
<td>11</td>
<td>18 (1.08×)</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>.65</td>
<td>7</td>
<td>11</td>
<td>18 (1.24×)</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>.05</td>
<td>15</td>
<td>21</td>
<td>36 (0.90×)</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>.35</td>
<td>15</td>
<td>21</td>
<td>36 (1.14×)</td>
<td>9</td>
<td>32</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>.65</td>
<td>15</td>
<td>22</td>
<td>37 (1.23×)</td>
<td>9</td>
<td>37</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>.05</td>
<td>31</td>
<td>44</td>
<td>75 (0.95×)</td>
<td>20</td>
<td>52</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>.35</td>
<td>32</td>
<td>44</td>
<td>76 (1.23×)</td>
<td>19</td>
<td>74</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>.65</td>
<td>31</td>
<td>44</td>
<td>75 (1.36×)</td>
<td>21</td>
<td>81</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>.05</td>
<td>74</td>
<td>108</td>
<td>182 (0.95×)</td>
<td>40</td>
<td>132</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>.35</td>
<td>78</td>
<td>116</td>
<td>194 (1.31×)</td>
<td>42</td>
<td>212</td>
</tr>
<tr>
<td>$2^{21}$</td>
<td>.65</td>
<td>77</td>
<td>124</td>
<td>201 (1.44×)</td>
<td>42</td>
<td>248</td>
</tr>
<tr>
<td>$2^{23}$</td>
<td>.05</td>
<td>178</td>
<td>262</td>
<td>440 (0.94×)</td>
<td>87</td>
<td>326</td>
</tr>
<tr>
<td>$2^{23}$</td>
<td>.35</td>
<td>190</td>
<td>258</td>
<td>448 (1.50×)</td>
<td>93</td>
<td>579</td>
</tr>
<tr>
<td>$2^{23}$</td>
<td>.65</td>
<td>171</td>
<td>246</td>
<td>417 (1.73×)</td>
<td>89</td>
<td>632</td>
</tr>
</tbody>
</table>

As a consequence, none of the two axis selection heuristics is a clear winner, even if the DAS approach has a weak point that makes it less predictable. In particular, this technique works by means of the threshold indicating when to perform the axis-switch. If such a value is too high we could pay a longer pairing phase while a too small value could turn out in a “ping-pong” effect when all the three axes are clustered. In such cases, since the switch becomes effective in the next frame, we frequently pay some extra time for performing the pairing phase on a problematic axis in the current frame and the initialization of the new axis.

11 Comparison to Other Algorithms

We compared our parallel algorithm with the CPU solutions presented in [12], [16] and the GPU-based one described in [14]. Batista et al. [12] parallelized the algorithm presented by Zomorodian et al. [31]. The method presented in [16] uses a parallel all-pairs testing algorithm. To speed up the approach, the implementation used here divides the computation into blocks that are sized to fit in the cache, which increases data locality and reduces cache misses. The GPU-based solution presented by Lo et al. [14] splits the space in a grid, then performs pairwise checks for each cell. To reduce
First, we performed a set of synthetic tests as in Section 8, namely, a set of \( n \) boxes of varying sizes, initially spread uniformly with density \( d \), moving freely in a cubic world for 100 iterations. Table 2 shows the average elapsed time per iteration for each run.

The parallel brute-force approach was not able to cope with large scale simulations in a reasonable time, while the GPU solution yielded the best performance on large inputs which is probably due to the higher computing power and memory bandwidth of the graphics device.

We further performed a set of self-intersection tests on real-world cases as shown in Table 3. In this way, we can study the behavior for other types of box distributions. Note, however, that our solution was primarily designed for true \( n \)-body scenarios, or broad phase collision culling. There are other types of algorithms specifically designed for dealing with self-intersections (e.g. [36], [37], [38]). We selected a number of datasets from The Stanford 3D Scanning Repository [39], the KScan3D gallery [40], and the UNC Dynamic Scene Benchmarks set [41]. Those picked from the first two repositories are dense triangle meshes. Since these models are static, our algorithm could not exploit the temporal coherence. The remaining datasets are simulations of moving objects represented by triangles of which the vertices change position at each frame. For every frame, we approximated each triangle with its bounding box, and then

### Table 2

<table>
<thead>
<tr>
<th>( n \times d )</th>
<th>Avril [16]</th>
<th>Batista [12]</th>
<th>Lo [14]</th>
<th>Our</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>3.08</td>
<td>0.37</td>
<td>4.36</td>
<td>0.29</td>
</tr>
<tr>
<td>12</td>
<td>3.25</td>
<td>0.49</td>
<td>4.13</td>
<td>0.34</td>
</tr>
<tr>
<td>12</td>
<td>3.37</td>
<td>0.58</td>
<td>3.79</td>
<td>0.38</td>
</tr>
<tr>
<td>14</td>
<td>41.77</td>
<td>1.42</td>
<td>5.07</td>
<td>0.90</td>
</tr>
<tr>
<td>14</td>
<td>43.91</td>
<td>1.97</td>
<td>5.00</td>
<td>1.06</td>
</tr>
<tr>
<td>14</td>
<td>44.33</td>
<td>2.31</td>
<td>4.70</td>
<td>1.19</td>
</tr>
<tr>
<td>16</td>
<td>646.93</td>
<td>7.06</td>
<td>6.39</td>
<td>3.63</td>
</tr>
<tr>
<td>16</td>
<td>670.39</td>
<td>10.27</td>
<td>5.98</td>
<td>4.63</td>
</tr>
<tr>
<td>16</td>
<td>682.00</td>
<td>11.66</td>
<td>6.11</td>
<td>5.39</td>
</tr>
<tr>
<td>18</td>
<td>10207.67</td>
<td>61.17</td>
<td>7.72</td>
<td>16.59</td>
</tr>
<tr>
<td>18</td>
<td>10482.85</td>
<td>87.00</td>
<td>8.73</td>
<td>23.24</td>
</tr>
<tr>
<td>18</td>
<td>10596.14</td>
<td>85.26</td>
<td>8.44</td>
<td>28.35</td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>307.47</td>
<td>14.24</td>
<td>79.89</td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>369.31</td>
<td>16.94</td>
<td>126.60</td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>398.10</td>
<td>17.73</td>
<td>163.68</td>
</tr>
</tbody>
</table>
we measured the time required for detecting all pairwise box collisions. We also tested our CAS strategy since, here, the endpoints are not uniformly distributed.

Our method outperformed the others in most of the cases and, for some datasets, a remarkable speedup was obtained by means of CAS, e.g., “Exploding Dragon & Bunny”. The GPU framework was not able to complete successfully in some of the tests and its performance dropped significantly in others. The problem might be related to the implementation of the gridding phase or a limitation of this method when applied to scenarios with an uneven distribution of the boxes.

Overall, our solution proved to be significantly faster than the other multi-core approaches and flexible enough to provide good performance in different kinds of scenarios. On the other hand, the GPU algorithm led to better results in large simulations with well-distributed objects. Of course, the relative performance depends on factors such as the computing power of the devices, the input size, and the scenario. In particular, high-end GPUs are capable of higher FLOP rates than CPUs. However, such solutions need to exchange data between the graphics device and the system main memory, which may lead to significant overheads.

12 CONCLUSIONS

We have presented a fully parallelized SaP algorithm for the dynamic box intersection problem. By exploiting the architecture of modern CPUs, we realized an efficient, cache-oriented, multi-core solution that scaled up to large datasets. Our algorithm is able to handle challenging scenarios without severe performance drops. The experimental results confirm its good qualities in practice resulting in a remarkable boost in the collision culling performance.

As several high performance collision culling methods targeting GPUs are already known [13], [14], an interesting next step would be to aim for a heterogeneous parallelization utilizing a combination of CPUs and GPUs to cope with even larger simulations. Finally, the auspicious runtime behavior of our parallel SaP algorithm also motivates us to consider how it can be applied in other contexts. For example, it would be interesting to adapt it to collision avoidance in simulations of large animal swarms.

TABLE 3
Statistics of the self-intersection datasets shown at the bottom. The average number of collisions and the elapsed time (ms) per frame are given.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Stanford Bunny [39]</td>
<td>69,451</td>
<td>1</td>
<td>431,277</td>
<td>450.69</td>
<td>21.48</td>
<td>69.90</td>
<td>6.25</td>
</tr>
<tr>
<td>Armadillo [39]</td>
<td>345,944</td>
<td>1</td>
<td>2,348,419</td>
<td>1573.41</td>
<td>142.06</td>
<td>42.56</td>
<td>22.23</td>
</tr>
<tr>
<td>Happy Buddha [39]</td>
<td>1,087,716</td>
<td>1</td>
<td>7,846,101</td>
<td>–</td>
<td>417.86</td>
<td>–</td>
<td>74.26</td>
</tr>
<tr>
<td>Model [40]</td>
<td>3,129,251</td>
<td>1</td>
<td>22,770,847</td>
<td>–</td>
<td>826.30</td>
<td>317.61</td>
<td>148.53</td>
</tr>
<tr>
<td>Hockey Player [40]</td>
<td>6,892,532</td>
<td>1</td>
<td>50,616,871</td>
<td>–</td>
<td>1,866.85</td>
<td>805.30</td>
<td>527.85</td>
</tr>
<tr>
<td>Funnel [41]</td>
<td>18,484</td>
<td>500</td>
<td>137,122</td>
<td>38.62</td>
<td>5.09</td>
<td>5.32</td>
<td>1.61</td>
</tr>
<tr>
<td>Cloth-ball [41]</td>
<td>92,230</td>
<td>94</td>
<td>692,506</td>
<td>733.29</td>
<td>31.88</td>
<td>9.17</td>
<td>2.97</td>
</tr>
<tr>
<td>N-body [41]</td>
<td>146,480</td>
<td>76</td>
<td>1,061,893</td>
<td>1,467.98</td>
<td>73.67</td>
<td>11.14</td>
<td>14.78</td>
</tr>
<tr>
<td>Exploding Dragon &amp; Bunny [41]</td>
<td>252,572</td>
<td>16</td>
<td>1,372,773</td>
<td>4134.49</td>
<td>104.24</td>
<td>–</td>
<td>35.00</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

This research was supported by the Swedish Foundation for Strategic Research (grant IIS11-0060).

REFERENCES


Gabriele Capannini received the PhD degree in computer science from the University of Pisa in 2012. He joined the Institute of Information Science and Technologies (ISTI) of the Italian National Research Council (CNR) in Pisa before the PhD studies. Currently, he works as a researcher at Mälardalen University, Sweden. His interests include: GPU/GPGPU, information retrieval, automatic summarization, and, more recently, computational geometry.

Thomas Larsson received the PhD degree in Computer Science from Mälardalen University, Sweden, in 2008. Currently, he is a senior researcher and teacher at the same university. His research interests include three-dimensional computer graphics, parallel algorithms, GPU computing, computational geometry, collision detection, simulation, real-time rendering, interactive techniques, and human-computer interaction.