

# Finding Neighbors in a Forest: A *b-tree* for Smoothed Particle Hydrodynamics Simulations

Aurélien Cavelan

*University of Basel, Switzerland*  
*aurelien.cavelan@unibas.ch*

Rubén M. Cabezón

*University of Basel, Switzerland*  
*ruben.cabazon@unibas.ch*

Jonas H. M. Korndorfer

*University of Basel, Switzerland*  
*jonas.korndorfer@unibas.ch*

Florina M. Ciorba

*University of Basel, Switzerland*  
*florina.ciorba@unibas.ch*

October 8, 2019

## Abstract

Finding the exact close neighbors of each fluid element in mesh-free computational hydrodynamical methods, such as the Smoothed Particle Hydrodynamics (SPH), often becomes a main bottleneck for scaling their performance beyond a few million fluid elements per computing node. Tree structures are particularly suitable for SPH simulation codes, which rely on finding the exact close neighbors of each fluid element (or SPH particle). In this work we present a novel tree structure, named *b-tree*, which features an adaptive branching factor to reduce the depth of the neighbor search. Depending on the particle spatial distribution, finding neighbors using *b-tree* has an asymptotic best case complexity of  $O(n)$ , as opposed to  $O(n \log n)$  for other classical tree structures such as octrees and quadtrees. We also present the proposed tree structure as well as the algorithms to build it and to find the exact close neighbors of all particles. We assess the scalability of the proposed tree-based algorithms through an extensive set of performance experiments in a shared-memory system. Results show that *b-tree* is up to  $12\times$  faster for building the tree and up to  $1.6\times$  faster for finding the exact neighbors of all particles when compared to its octree form. Moreover, we apply *b-tree* to a SPH code and show its usefulness over the existing octree implementation, where *b-tree* is up to  $5\times$  faster for finding the exact close neighbors compared to the legacy code.

# 1 Introduction

Hydrodynamical simulations rely, in their vast majority, on efficient algorithms for finding the exact close neighbors among the fluid elements. In the case of static meshes, the neighborhood is inherently fixed by the mesh geometry, and can be naively explored for finding neighbors using stencils. But, how can neighbors be found in unstructured meshes? This is precisely the case of the Smoothed Particle Hydrodynamics (SPH) technique [11, 14, 19, 21]. This method discretizes the fluid in a series of interpolating points (named SPH particles or particles, hereafter) that are distributed following the actual density profile of the simulated fluid. Then the physical properties of each particle are obtained with a weighted radial interpolation over closely neighboring particles. The weight of this interpolation has compact support, and its radius is named the *smoothing length*. This means that in non-homogeneous, highly dynamic systems –such as those found in Computational Fluid Dynamics (CFD) and Astrophysics– the particle distribution can be geometrically very distorted. In this case, finding close neighbors in a computationally-efficient way is a non-trivial problem.

The most common approach for finding neighbors in SPH is to use a tree structure that recursively divide the spatial computational domain in sub-cells<sup>1</sup>, until leaves are empty, or contain a small bucket of particles. Once the tree is built, it can be walked to find the exact close neighbors of each particle, discarding whole branches when their parent cells are too far from the current particle, thereby decreasing the *search time*. This is particularly relevant in Astrophysical simulations, where the tree also stores information about the multipolar expansion of the gravitational field and is used to efficiently evaluate the gravitational force that the particles experience [12].

In this work, we introduce a novel tree algorithm for the *exact* close neighbor search, named *b-tree*. The proposed tree aims at drastically decreasing the *search time* by building a very shallow tree. This is achieved by choosing a very high branching factor, i.e. allowing nodes to have many children, effectively building a *broad-tree* instead of a deep-tree.

*b-tree* uses an adaptive branching factor to prevent the number of sub-cells from increasing excessively within a single cell of the tree. This is done by enforcing a limit both, on the maximum number of particles per cell (i.e., the bucket size) and on the number of empty sub-cells allowed.

More specifically, *b-tree* recursively divides the spatial computational domain into smaller, *equal-size sub-cells*. The resulting sub-cells are then mapped onto a regular grid structure for easy access: grid structures with equal-size cells render an  $O(1)$  access time, i.e., it is possible to find the sub-cell that contains a given particle with known coordinates in constant time<sup>2</sup> (see Section 3).

---

<sup>1</sup>In the rest of this paper, we refer indifferently to a tree node as a cell and to its children as a sub-cells.

<sup>2</sup>In theory, it is possible to map any simulation domain onto a large enough grid. In practice, the number of grid cells required would quickly render this design impractical. Specifically, this approach relies on grids with equal-size cells: the cell size (and, therefore, the number of cells in the grid) needs to accommodate the most dense part of the computational domain. To guarantee that any cell in the grid contains at most one particle, the cell size must be smaller than the shortest distance between any two particles. For very heterogeneous particle distributions, this approach would require a prohibitive amount memory to be a viable solution, hence the adaptive branching factor proposed herein.

The resulting tree has interesting properties. In particular when the set of  $n$  particles has a relatively uniform distribution across the computational domain, the depth of  $b$ -tree is 1 and requires  $O(n)$  steps to be built and  $O(n)$  steps to find the neighbors of  $n$  particles (i.e., a constant number of steps per particle), as opposed to  $O(n \log n)$  in the best case with a standard octree implementation. A classical octree requires at most  $O(nd \log n)$  steps to recursively build the tree, where  $d$  is the depth of the tree, and at most  $O(nd)$  steps to find the neighbors (i.e.,  $O(d)$  per particle), where  $d$  is typically close to  $\log n$ .

In this work, we experiment with both, a uniformly distributed 3D particle dataset as well as a non-uniformly 3D particle dataset, to demonstrate the  $b$ -tree properties and performance benefits. The code that builds the tree and finds exact neighbors is provided as a standalone code, with both C++ and Fortran interfaces, while the experiments are provided as a single reproducible package. Furthermore,  $b$ -tree has been integrated into an astrophysical SPH code, SPHYNX [4]. However, the applicability of the  $b$ -tree code is neither limited to astrophysical applications, nor SPH codes.

The remainder of the work is structured as follows. In Section 2 we review the different tree algorithms that can be found in the SPH literature. We introduce the proposed  $b$ -tree algorithm in Section 3. We evaluate its performance against the classical octree algorithm in Section 4. We conclude the work in Section 5 and outline future work directions.

## 2 Related Work

A vast amount of literature exists on tree-based algorithms. A detailed review can be found in the work of Curtin et al. [8] (and references therein). In this section, we review three methods in the field of CFD and Computational Astrophysics, with a particular focus on SPH simulation codes.

**Octrees** [13] have widely been used in synergy with SPH codes. With octrees, the computational domain is recursively halved in each dimension, until there is only a single particle per leaf or none at all. Octrees are a 3D generalization of quadrees that are typically applied to 2D computational domains [10]. Hernquist & Katz [12] first proposed the usage of a hierarchical tree structure as an efficient, dynamical, and fully-Lagrangian method to evaluate gravitational forces. This method was based on the Barnes-Hut algorithm [2] that uses an octree to evaluate gravitational forces within a multipolar approximation. Octrees are used in GADGET2 [20], ChaNGa [15], SPH-flow [16], and SPHYNX [4], among many other SPH codes.

**$kd$ -trees** [3] have been proposed to avoid the exponential dependence of quadrees and octrees to the spatial dimension. In  $kd$ -trees, each node has only two children, but every division is always aligned to one of the dimension axes.  $kd$ -trees are employed by GASOLINE2 [23] and PHANTOM [18].

**Ball trees** [17] are binary trees in which each node has an associated hypersphere that it is the smallest volume that contains the hyper-spheres of its children. Unlike  $kd$ -trees, the node regions can intersect and do not require the partitioning of the entire computational domain. Ball trees are seldomly used by SPH codes.

**$b$ -tree vs. octree:** While octrees have branching factors of  $2^3$  on average, i.e. halving the spatial domain along each dimension in 3D distributions,  $b$ -tree

Table 1: Notation used in this work

Input Parameters	
Name	Description
$k$	Number of spatial dimensions
$n$	Total number of particles
Tree Parameters	
$s$	Bucket size, i.e. maximum number of particles in a leaf
$\alpha$	Fraction of the bucket size $s$
$\beta$	Max. ratio of cells with less than $\alpha S$ particles
Other Variables	
$d$	Tree depth
$b$	Branching factor per spatial dimension (at a given node)
$r$	Current ratio of cells with less than $\alpha S$ particles
$n_i$	Number of particles in cell $i$

may have branching factors up to  $n$ , depending on the particle spatial distribution. The number of children at each node in  $b$ -tree depends on the branching factor ( $b$ ) and on the bucket size ( $s$ ) that determines the maximum number of particles in each leaf.  $s$  is a user-defined parameter, while  $b$  is computed while building the tree in order to adapt to the particle distribution. In the best case, for perfectly uniform particle distributions,  $b$ -tree has  $O(1)$  complexity for finding the exact nearest neighbors of a particle, while octrees have  $(\log n)$  complexity at best. In the worst case, when the particle distribution is found to be highly non-uniform,  $b$ -tree collapses to an octree and  $b$  is automatically set to 2 and the complexity for finding the exact nearest neighbors of a particle is  $O(d)$  for both. Table 1 shows the different parameters used in this paper.

**$b$ -tree vs. stratified trees:**  $b$ -tree has a high branching factor, which is a characteristic shared with stratified trees, such as the van Emde Boas trees [22]. Stratified trees have been used for finding the approximate nearest neighbor [1, 5] with lower complexity, namely  $O(\log \log n)$ , than that of octrees, namely  $O(\log d)$  for imbalanced trees and  $O(\log n)$  for balanced trees. However, hydrodynamical simulations require finding the exact neighbors of each particle, and  $b$ -tree provide exactly this, at the cost of sacrificing part of the asymptotic complexity of stratified trees, yet still being more efficient than octrees.

### 3 $b$ -tree

In this section, we introduce the proposed  $b$ -tree structure and present the algorithms for building the tree in Section 3.1 and for finding the exact close neighbors of SPH particles in Section 3.2.

#### 3.1 Tree Building

Algorithm 1 describes the BUILD TREE process. The algorithm recursively distributes a given set of  $n$  particles into smaller, equal-size  $b^k$  sub-cells, where  $k$  is the number of spatial dimensions and  $b$  is the branching factor, i.e. the number

of children in each dimension for the current node (see Table 1).

---

**Algorithm 1** BuildTree

---

```

procedure BUILDTREE( $n, i$ )
   $redistribution \leftarrow true$ 
  while  $redistribution$  is true do
    Compute branching factor  $b$  (Equation 1)
    Create  $b^k$  cells and distribute the particles:
    for each particle  $p$  in cell  $i$  do
      Compute its sub-cell coordinates (Equation 2)
      Compute its sub-cell id  $j$  (Equation 3)
      Add  $p$  to its corresponding sub-cell  $j$ 
      Update  $n_j$ 
    end for
    Compute the distribution ratio  $r$  (Equation 4)
    if  $R < \beta$  then
       $b = \frac{b}{2}$ 
    else
       $redistribution \leftarrow false$ 
    end if
  end while
  for each sub-cell  $j$  do
    if  $n_j > s$  then
      BUILDTREE( $n_j, j$ )
    end if
  end for
end procedure

```

---

**Step 1:** Given a set of  $n$  particles in the current cell  $i$ , the BUILDTREE algorithm computes the branching factor  $b$  and create  $b^k$  empty equal-size sub-cells. To compute the branching factor  $b$ , we initially assume that particles are uniformly distributed in the spatial computational domain. Therefore, given an upper limit on the number of particles per bucket  $s$ , we want to find  $b$  such that  $\frac{n}{b^k} \leq s$ . Solving for  $b$ , and rounding to the nearest higher integer value, we obtain:

$$b = \left\lceil \sqrt[k]{\frac{n}{s}} \right\rceil. \quad (1)$$

**Step 2:** The particles are distributed into the newly created cells, based on the particles coordinates. The sub-cells are mapped onto a  $k$ -dimensional grid, i.e. each sub-cell has a cut of the current computational domain. Each sub-cell therefore has its own set of  $k$ D-coordinates within the grid. For each particle with coordinates  $x_1, \dots, x_k$ , we compute its corresponding sub-cell coordinates  $x'_1, \dots, x'_k$  by: (1) normalizing the coordinates of the particle with respect to the current cell's sub-domain; (2) subsequently multiplying the normalized coordinates by the number of sub-cells in each dimension, i.e.,  $b$ ; and (3) rounding the resulting coordinates to the nearest lower integer value as follows:

$$x'_l = \left\lfloor \frac{x_l - x_{l,min}}{x_{l,max} - x_{l,min}} \cdot b \right\rfloor, \quad (2)$$

where  $x_{l,min}$  and  $x_{l,max}$  are the bounds of the domain in the  $l$  dimension (otherwise known as bounding box).

For example, consider a particle with the following 3D coordinates  $x_1 = 3.6$ ,  $x_2 = 4.2$ , and  $x_3 = 0.6$ . You cannot change the nomenclature to  $x,y,z$  when we are using a generalized coordinate system that is valid to all coordinate sets. In this example,  $b$  has been set to 10. Suppose that this particle is assigned to the sub-domain box defined by  $x_{1,min} = x_{2,min} = x_{3,min} = 0$  and  $x_{1,max} = x_{2,max} = x_{3,max} = 5$ . According to Eq. 2, we obtain the coordinates of the cell that contains the particular particle as follows:

$$\begin{aligned} x'_1 &= \left\lfloor \frac{3.6}{5} \cdot 10 \right\rfloor = \lfloor 7.2 \rfloor = 7, \\ x'_2 &= \left\lfloor \frac{4.2}{5} \cdot 10 \right\rfloor = \lfloor 8.4 \rfloor = 8, \\ x'_3 &= \left\lfloor \frac{0.6}{5} \cdot 10 \right\rfloor = \lfloor 1.2 \rfloor = 1. \end{aligned}$$

For each particle in the domain, assuming that cells are stored in a 1D array and numbered from 0 to  $W^d - 1$ , we can compute the corresponding sub-cell id  $j$  as follows:

$$j = x'_1 + x'_2 W + x'_3 W^2. \quad (3)$$

In our example, the index of the corresponding bucket within the current node would be  $j = 7 + 8 \cdot 10 + 1 \cdot 10^2 = 187$  (out of  $b^3 = 1000$  sub-cells).

**Step 3:** Finally, to prevent the tree breadth from exploding if the initial value of  $b$  was too high, we introduce two parameters, namely  $\alpha$  and  $\beta$  (see Table 1), to control the number of sub-cells per node. Specifically, we first compute the distribution ratio  $r$ , which measures the ratio of sub-cells that contain less than  $\alpha s$  particles (with  $\alpha \sim 0.5$ ) over the total number of sub-cells  $b^d$  within the current node:

$$r = \frac{\sum_{j=0}^{W^d-1} n_j}{W^d}, \quad (4)$$

where  $n_j$  is the number of particles that have been assigned to sub-cell  $j$ . If the resulting distribution ratio  $r$  is larger than  $\beta$  (with  $\beta \sim 0.5$ ), it means that too many cells with too few particles have been created within the current node. Therefore, we divide the branching factor  $b$  by 2, hence reducing the number of cells by 2 in each dimension, or by a factor of  $2^k$  in total.

We recompute steps 1, 2 and 3 until this criterion is met, i.e., when  $r < \beta$ . In the worst case, the algorithm stops at  $b = 2$ , which corresponds to an octree structure.

The particle redistribution takes at most  $O(\log b)$  steps, which is proportional to  $O(\log n)$  due to  $\log b = \log \left( \frac{n}{s} \right)^{\frac{1}{d}} = \frac{1}{d} \log \frac{n}{s}$ .

## 3.2 Finding Neighbors

Algorithm 2 describes the FINDNEIGHBORS algorithm. In SPH simulations, each particle is characterized by a *smoothing length*, denoted  $h$ . Finding the

exact close neighbors of a given particle reduces to finding all particles that are within the  $2h$  radius<sup>3</sup>.

To find the exact close neighbors of a given particle, the algorithm needs to walk the tree by discarding cells that are not within the  $2h$  radius, and by visiting cells that are within the  $2h$  radius.

More specifically, given a particle  $p$ , its radius  $2h_p$  and a starting cell  $i$ , the algorithm needs to identify the sub-cells that are within the  $2h_p$  radius of  $p$ . Rather than checking if every sub-cell is within the  $2h_p$  radius, the algorithm directly computes the range of sub-cells coordinates to visit in each dimension  $l$ , denoted by  $r_l = [r_{l,min}, r_{l,max}]$ .

All close neighbors of  $p$  are within the range  $[x_l - 2h_p, x_l + 2h_p]$ , with  $l \in [1, k]$ , and where  $x_1, \dots, x_k$  are the coordinates of the current particle  $p$ . Based on Equation 2, we write:

$$r_{l,min} = \left\lfloor \frac{(x_l - 2h_p) - x_{l,min}}{x_{l,max} - x_{l,min}} \cdot b \right\rfloor \quad (5)$$

$$r_{l,max} = \left\lceil \frac{(x_l + 2h_p) - x_{l,min}}{x_{l,max} - x_{l,min}} \cdot b \right\rceil. \quad (6)$$

Figure 1 illustrates the process with a 2D example, showing which cells need to be visited in order to find the neighbors of the particle highlighted in red.

Then, for every tuple of sub-cell coordinates within the computed range, we retrieve the corresponding sub-cell id,  $j$ , using Equation 3. When the algorithm reaches a cell  $j$  that contains  $n_j \leq s$  particles, then the cell is a leaf and the algorithm checks all the particles that are stored inside the cell individually. For every particle  $q$  in cell  $j$ , we compute the Cartesian distance between  $p$  and  $q$ , and if the distance is less than  $2h_p$  we add particle  $q$  to the list of neighbors of particle  $p$ , denoted by  $\text{NeighborsOf}(p)$ .

---

**Algorithm 2** FindNeighbors

---

```

procedure FINDNEIGHBORS( $p, h_p, i$ )
  Compute range  $r_l = [r_{l,min}, r_{l,max}]$ ,  $l \in [1, k]$  (Eq. 6)
  for each cell  $(x_1, x_2, \dots, x_k) \in r_1 \times r_2 \dots \times r_k$  do
    Compute sub-cell id  $j$  (Eq. 3)
    if  $n_j \leq s$  then
      for each particle  $q$  in cell  $j$  do
        if  $\text{Distance}(p, q) \leq 2h_p$  then
          Add  $q$  to  $\text{NeighborsOf}(p)$ 
        end if
      end for
    else if  $n_j > s$  then
      FINDNEIGHBORS( $p, h_p, j$ )
    end if
  end for
end procedure

```

---

<sup>3</sup> $2h$  is the standard neighborhood radius of many interpolating functions (or kernels) in SPH. Although it is possible to use a kernel that employs a larger radius, the radii are always proportional to the smoothing length  $h$ . The  $b$ -tree approach proposed herein is directly applicable to other SPH kernels with larger such radii.

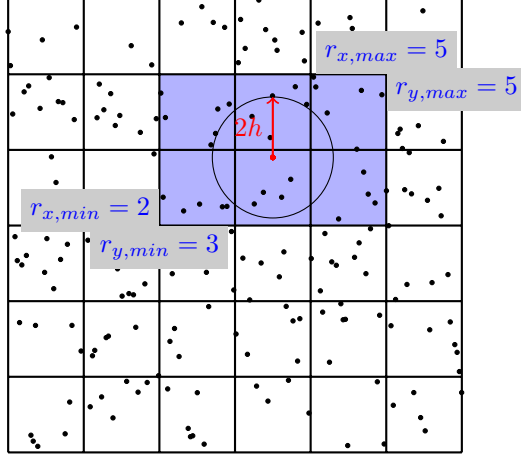


Figure 1: Range of sub-cells to visit (in blue) within the current node in order to find all the close neighbors of the particle highlighted in red, i.e. all particles that are within its  $2h$  radius.

### 3.3 Complexity Analysis

**Time complexity:** Overall, the number of steps required for building the tree and finding the neighbors depends on the depth of tree, denoted  $d$ . The BUILDTREE algorithm requires at most  $O(nd \log n)$  steps for redistributing at most  $n$  particles  $\log n$  times, over  $d$  levels, while the octree only requires  $O(nd)$  steps, without the additional redistribution steps. Both the FINDNEIGHBORS algorithm and classical octree implementations requires at most  $O(nd)$  steps in order to find the neighbors of  $n$  particles, due to having to visit  $d$  levels every time.

The main difference between our  $b$ -tree and an octree is the depth of the tree, which depends on the particle spatial distribution. In the worst case,  $b$ -tree collapses to a quadtree (2D) or an octree (3D) with the same depth. Unlike  $kd$ -trees, the depth of a quadtree or an octree is not guaranteed to be  $\log n$  in the worst case. In fact, in the worst case there is only one particle per level and there are  $d = n$  levels. In the best case, when particles are distributed uniformly across the spatial computational domain, an octree will be balanced and have depth  $d = \log n$ , while  $b$ -tree will be only one level deep and have depth  $d = 1$ , and will require no redistribution.

Table 2 summarizes the time complexity for the proposed  $b$ -tree, compared to a classical octree.  $n$  is the number of particles and  $d$  denotes the maximum depth of the tree.

**Memory complexity:** In the worst case,  $b$ -tree may create up to  $n$  cells per level (with up to  $d = n$  levels in the worst case), therefore the space required is at most  $O(nd)$ . While  $b$ -tree may have many more cells than an octree, only non-empty cells need to be stored in memory, and the impact remains small compared to the space required for storing the neighbors of every particle.



Table 2: Asymptotic complexity for tree building and finding neighbors:  $b$ -tree and octree comparison

	Tree Building		Finding Neighbors	
	Worst Case	Best Case	Worst Case	Best Case
$b$ -tree	$O(nd \log n)$	$O(n)$	$O(nd)$	$O(n)$
octree	$O(nd)$	$O(n \log n)$	$O(nd)$	$O(n \log n)$

## 4 Experiments

In this section, we conducted experiments to assess the scalability of the proposed  $b$ -tree in a shared-memory system as well as to evaluate the impact of the different parameters on its performance. Moreover,  $b$ -tree has been integrated into an astrophysical SPH code, SPHYNX [4], and the performance of the new tree-based algorithms are evaluated against the legacy code in Section 4.7.

### 4.1 Experimental Setup

Building the tree and finding the neighbors are two operations that are typically performed within a single processing node, over a subset of the entire simulation domain, called sub-domain. The topic of domain partitioning across the computation nodes is beyond the scope of this work. In this work, we focus on the performance of the BUILD TREE and FIND NEIGHBORS algorithms for exploiting many-core parallelism. Specifically, we perform experiments using a single Intel Xeon compute node, the details of which are presented in Table 3. Execution times results are averaged out of 100 executions for every configuration to produce representative data.

Table 3: Characteristics of the experimental platform

Parameter	Description
Operating system	CentOS Linux release 7.2.1511
Processor	Intel Xeon E5-2640 v4
Number of cores	20 (+20 with hyperthreading)
Memory	64 GB RAM
Operating frequency	2.4 – 3.4 GHz

We consider two SPH simulation test cases: (a) an Evrard collapse (EC) test [9] with  $10^6$  particles, which studies the gravitational collapse of a gaseous cloud; EC is a common test to evaluate the correctness of the coupling of hydrodynamics and self-gravity and is an example of non-uniform particle distribution; and (b) a Square Patch (SP) test [7] with  $10^7$  particles; SP is a common test case in CFD to simulate highly distorted geometries and resistance to particle cumpling, and has a fairly uniform particle distribution.

Table 4: Design of target experiments

Parameter Name	Default Value	
Test case	EC	SP
#Threads	40	40
Particle distribution	Non-uniform	Uniform
#Dimensions ( $d$ )	3	3
#Particles ( $n$ )	$10^6$	$10^7$
Target #neighbors/particle	100	500
Bucket size ( $s$ )	8	8
$\alpha$	0.5	0.5
$\beta$	0.5	0.5

## 4.2 Strong Scaling

In this sub-section, we assess the speedup of the  $b$ -tree-based BUILDTREE and FINDNEIGHBORS algorithms, with respect to their standard octree-based counterparts using the particle datasets of the EC and SP test cases.

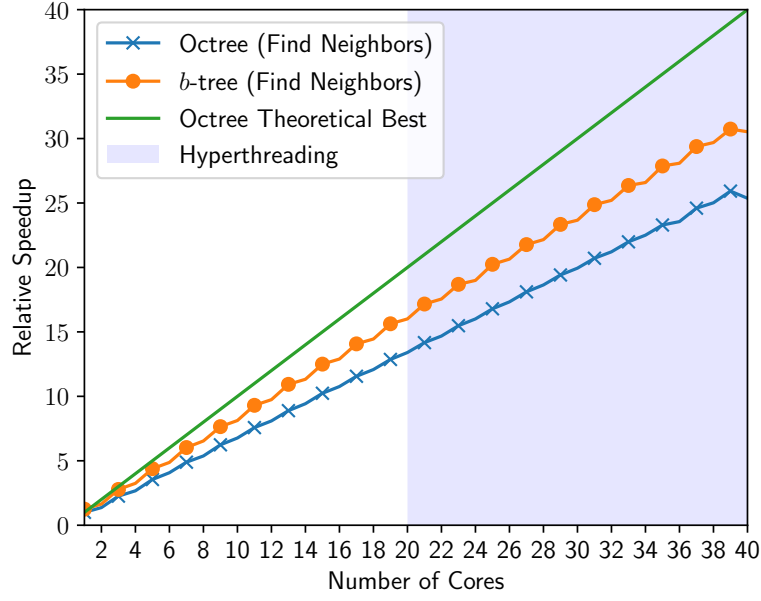
The relative speedup for building the tree and finding the neighbors is computed with respect to the octree build and search time, respectively, with one core. Figure 2(a) and Figure 2(a) show the the relative speedup for EC and SP test cases, respectively. While both search algorithms scale very well, even when exploiting the hyper-threading available on the Intel Xeon, the proposed FINDNEIGHBORS algorithm always yields better performance compared to the classical octree algorithm. In addition, the proposed BUILDTREE algorithm is always faster than its classical octree counterpart.

## 4.3 Impact of the Bucket Size

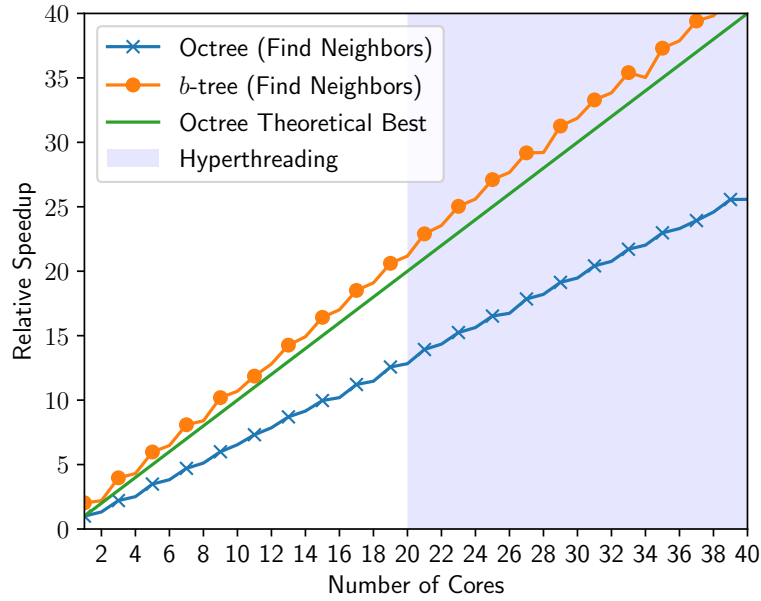
In this sub-section, we evaluate the impact of the bucket size parameter  $s$  on the performance of the  $b$ -tree- and octree-based algorithms for the EC and SP datasets. The optimal value of the bucket size for each test is derived experimentally, as shown in Figure 3. For building the tree, a larger bucket size means more particles per cell, and therefore less cells overall and a more shallow tree, which is faster to build. For finding the neighbors, a small bucket size is preferred to avoid having too many particles per cell, which increases the number of particle to particle distance checks within a cell. The optimal value is around 8 for both test cases.

## 4.4 Impact of $\alpha$ and $\beta$

Finally, we investigate the impact of the  $\beta$  parameter, which control the maximum amount of cells that have too few particles with respect to  $\alpha \sim 0.5$ , and hence we control the final branching factor for every tree node. Figure 4 shows both, the FINDNEIGHBORS and BUILDTREE execution time for different values of  $\beta$ . The non-uniform particle distribution for EC test case means than many empty cells are created with the default branching factor. With small values of  $\beta$ , BUILDTREE ends up selecting very small branching factors and the associated time is close to the classical octree behavior. However with higher values



(a) Evrard Collapse



(b) Square Patch

Figure 2: *b*-tree speedup normalized to the octree speedup for the EC test (a) and the SP test (b). The highlighted region denotes the experiments where hyper-threaded cores were employed in the experiments.

for  $\beta$ , BUILD TREE is allowed to use more cells and there are less redistributions steps, which is faster to build.

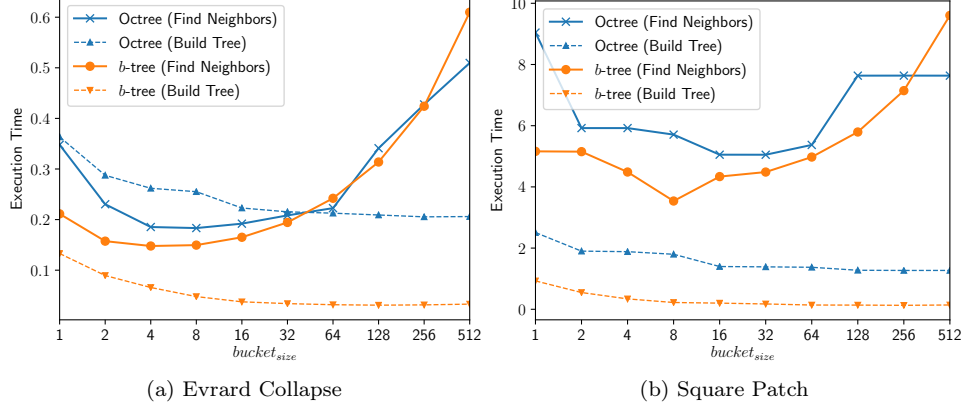


Figure 3: Impact of the bucket size  $S$  on the execution time of the tree-based algorithms for the EC test (a) and the SP test (b).

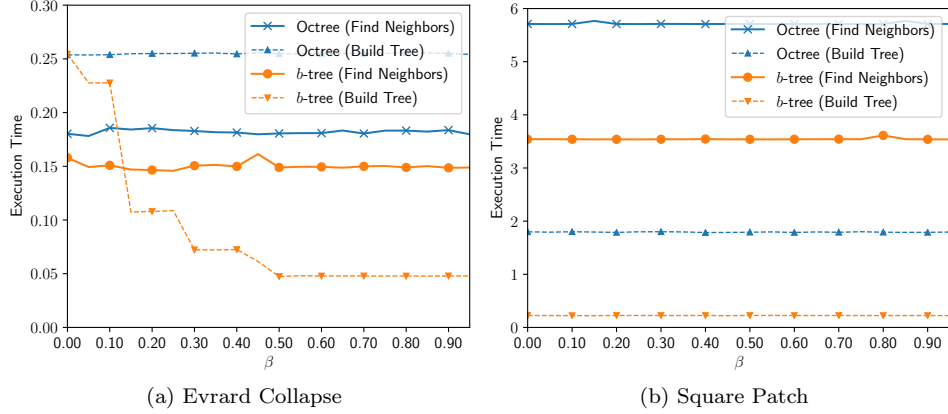


Figure 4: Impact of the target ratio parameter  $R$  on the execution time for Evrard Collapse test (a) and the Square Patch test (b).

## 4.5 Tree Walk Visualization

Finally, we provide a visual comparison of the resulting trees for the non-uniform EC test case. Figure 5 shows a 2D slice of the 3D domain at the center. Black squares represent cells and the colored line shows the order in which the cells are visited when walking the tree recursively.

Two important optimizations have been implemented to accelerate the tree walk: (1) particles are reordered in memory along this curve; and (2) BUILD TREE and FIND NEIGHBORS algorithms have been implemented with a blocked approach, similar to the one employed in blocked matrix-matrix multiplication implementations in order to further improve cache reuse. Different colors correspond to different blocks. Note that while  $b$ -tree has only two levels (note the two different cell sizes), the octree has 5 levels.

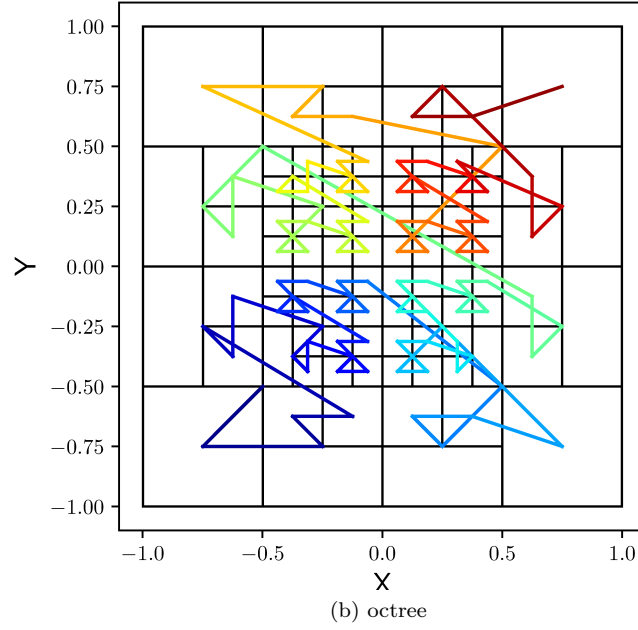
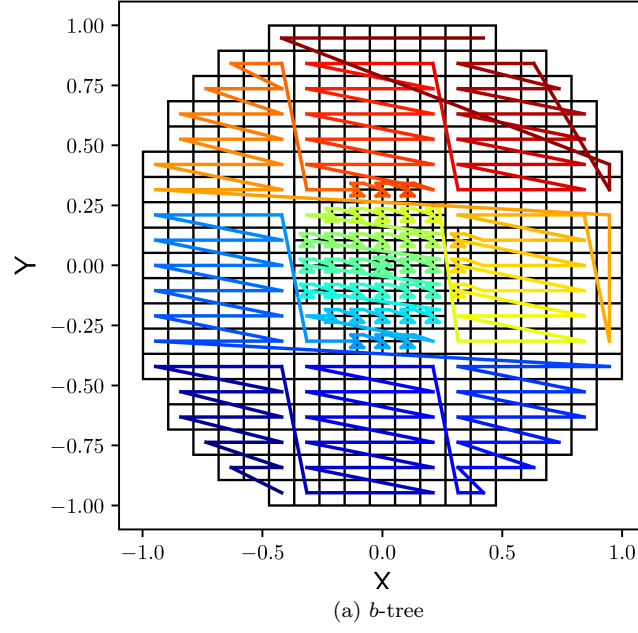


Figure 5: 2D slice at the center of the 3D domain for the EC test case. Black squares represent cells and the colored line shows the order in which the cells are visited when walking the tree. The bucket size is set to  $s = 32$  for more visibility, and particles are not shown.

## 4.6 Performance Optimizations

This sub-section presents a performance analysis for the FINDNEIGHBORS algorithm for *b*-tree and octree comparing them in terms of execution time and degree of load imbalance. The experiments employ five OpenMP loop scheduling strategies proposed and described in recent work by [6]: **static**, **dynamic**, **guided**, **fac2** and **rand**.

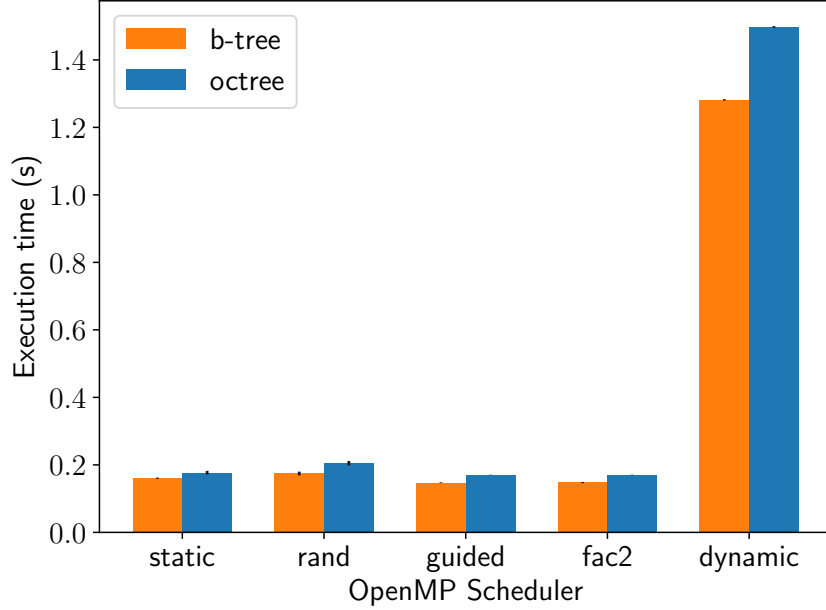


Figure 6: Execution time of employing *b*-tree and octree for the EC test case.

Figure 6 presents the average execution time on the EC input. In terms of execution time, *b*-tree obtained better results for all scheduling strategies. The latter approaches an execution time of 0.15 seconds using most scheduling strategies except for **dynamic**, when it reaches 1.3 seconds. In terms of load balancing, **dynamic** always balances the load better, while causing significant overhead that noticeably degrades the performance.

Figure 7 presents the average execution time the results for the SP test case. In Figure 7, *b*-tree obtained better results for all scheduling strategies. However since this input is much larger, the execution time stays around 4 seconds for most of the scheduling strategies, except for **dynamic** that causes significant overhead during the execution. The small degree of load imbalance observed is well amortized by the benefit of the self-scheduling properties underlying **guided** and **fac2**, resulting in improved performance.

## 4.7 SPHYNX

Finally, *b*-tree has been integrated into an astrophysical SPH code, SPHYNX [4]. Figure 8 shows the execution time of the *b*-tree algorithms with respect to the original, legacy octree implementation for a smaller SP test case with one

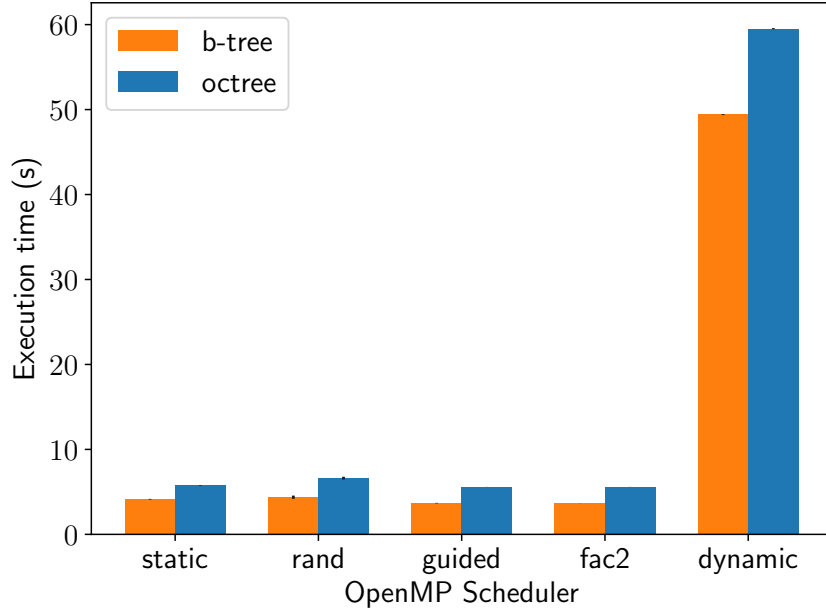


Figure 7: Execution time of employing *b-tree* and *octree* for the SP test case.

million particle (the 10 million particles test could not run with the legacy code). FINDNEIGHBORS is up to  $5\times$  faster.

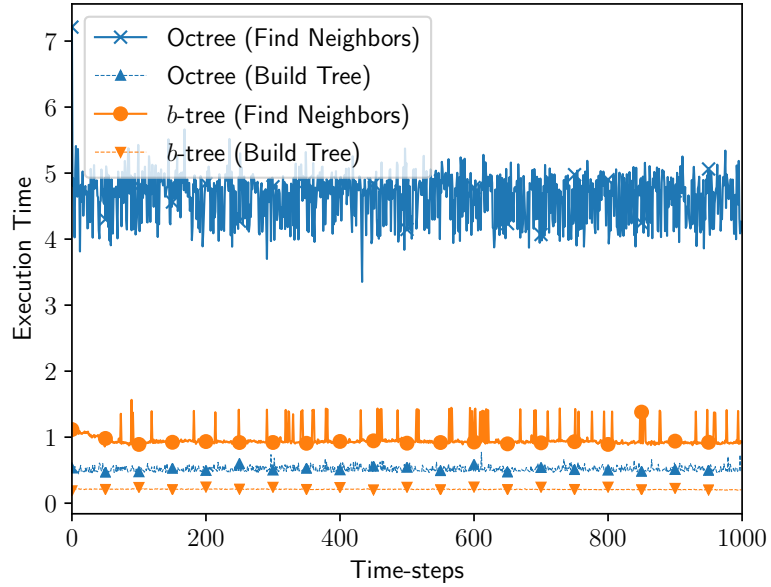


Figure 8: SPHYNX execution times for FINDNEIGHBORS and BUILDTREE algorithms compared to the legacy octree implementation.

## 5 Conclusion

In this paper, a novel tree structure, namely  $b$ -tree, was proposed to improve the exact close neighbors searching time for Smoothed Particle Hydrodynamics simulations. Algorithms to build the tree and to find the neighbors have been presented, and their complexity analyzed. Experimental results show both a good scalability and improved speedup compared to a classical octree implementation. Integration of the algorithm in a production SPH simulation shows promising results, with up to  $5\times$  improvements over the legacy code, delivering relevant speedups in a section of the hydrodynamical codes that is a common performance bottleneck.

Future work will address computational domain decomposition among computing nodes, coupling of the tree-based algorithms with gravity calculations, as well as testing the usefulness of the proposed  $b$ -tree on other SPH test cases.

## Acknowledgement

This work is supported in part by the Swiss Platform for Advanced Scientific Computing (PASC) via the “SPH-EXA: Optimizing Smooth Particle Hydrodynamics for Exascale Computing” grant and by the Swiss National Science Foundation (SNF) via the “Multi-level Scheduling in Large Scale High Performance Computers” grant, number 169123. The authors acknowledge the support of sciCORE (<http://scicore.unibas.ch/>) scientific computing core facility at University of Basel, where part of these calculations were performed.

## References

- [1] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for location and proximity problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 160–170. IEEE, 1999.
- [2] J. Barnes and P. Hut. A hierarchical  $O(N \log N)$  force-calculation algorithm. *Nature*, 324:446–449, Dec. 1986.
- [3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.
- [4] R. M. Cabezón, D. García-Senz, and J. Figueira. SPHYNX: an accurate density-based SPH method for astrophysical applications. *Astronomy & Astrophysics*, 606:A78, Oct. 2017.
- [5] M. Cary. Toward optimal  $\epsilon$ -approximate nearest neighbor algorithms. *Journal of Algorithms*, 41(2):417 – 428, 2001.
- [6] F. M. Ciorba, C. Iwainsky, and P. Buder. OpenMP Loop Scheduling Revisited: Making a Case for More Schedules. In *Proceedings of the 2018 International Workshop on OpenMP (iWomp 2018)*, Barcelona, September 2018.



- [7] A. Colagrossi. *A meshless Lagrangian method for free-surface and interface flows with fragmentation*. PhD thesis, 2005.
- [8] R. R. Curtin. *Improving dual-tree algorithms*. PhD thesis, Georgia Institute of Technology, 2015.
- [9] A. E. Evrard. Beyond N-body - 3D cosmological gas dynamics. *Monthly Notices of the Royal Astronomical Society*, 235:911–934, Dec. 1988.
- [10] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, Mar 1974.
- [11] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, Nov. 1977.
- [12] L. Hernquist and N. Katz. TREESPH - A unification of SPH with the hierarchical tree method. *Astrophysical Journal Supplement Series*, 70:419–446, June 1989.
- [13] C. L. Jackins and S. L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249 – 270, 1980.
- [14] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024, Dec. 1977.
- [15] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato. Adaptive techniques for clustered N-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2:1, Mar. 2015.
- [16] G. Oger, D. L. Touzé, D. Guibert, M. de Lefte, J. Biddiscombe, J. Soumagne, and J.-G. Piccinalli. On distributed memory mpi-based parallelization of sph codes in massive hpc context. *Computer Physics Communications*, 200:1 – 14, 2016.
- [17] S. Omohundro. *Five Balltree Construction Algorithms*. Technical report (International Computer Science Institute). International Computer Science Institute, 1989.
- [18] D. J. Price, J. Wurster, T. S. Tricco, C. Nixon, S. Toupin, A. Pettitt, C. Chan, D. Mentiplay, G. Laibe, S. Glover, C. Dobbs, R. Nealon, D. Liptai, H. Worpel, C. Bonnerot, G. Dipierro, G. Ballabio, E. Ragusa, C. Federrath, R. Iaconi, T. Reichardt, D. Forgan, M. Hutchison, T. Constantino, B. Ayliffe, K. Hirsh, and G. Lodato. Phantom: A smoothed particle hydrodynamics and magnetohydrodynamics code for astrophysics. *ArXiv e-prints*, Feb. 2017.
- [19] S. Rosswog. Boosting the accuracy of SPH techniques: Newtonian and special-relativistic tests. *Monthly Notices of the Royal Astronomical Society*, 448:3628–3664, Apr. 2015.
- [20] V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, Dec. 2005.

- [21] V. Springel. Smoothed Particle Hydrodynamics in Astrophysics. *Annual Reviews of Astronomy and Astrophysics*, 48:391–430, Sept. 2010.
- [22] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, SFCS '75, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.
- [23] J. W. Wadsley, B. W. Keller, and T. R. Quinn. Gasoline2: a modern smoothed particle hydrodynamics code. *Monthly Notices of the Royal Astronomical Society*, 471:2357–2369, Oct. 2017.