Efficient tree-based NNS for CFD applications

Christian Lagares
Dr. J. Fernando Vega
ICOM 5015
University of Puerto Rico at Mayaguez
Electrical and Computer Engineering Department
28 April 2021

Efficient tree-based NNS for CFD applications

CHRISTIAN LAGARES, University of Puerto Rico at Mayaguez, USA

Computational fluid dynamics (CFD) has risen to become one of the most essential tools in engineering design. However, significant progress is still to be made in high-performance, low-latency post-processing techniques. For this work, we concentrated on one specific aspect of particle tracing techniques. Particle tracing techniques are important for both processing and post-processing steps as both certain physics models and visualization techniques require it. For the present work, we focus on accelerating nearest neighbor searches which are required for efficient, on-demand interpolation at arbitrary locations. We present two validation domains and begin by assessing the performance of three approaches. The fastest pair is selected for a more computationally intensive benchmark simulating the order of magnitude of searches required for a single particle tracking experiment. The three methods include our naive approach which assumes a flat array (i.e. a tree with depth 1) and two octree search approaches, depth-first and best-first search. Best first search provides the best overall results improving our existing algorithm by at least 3 orders of magnitude. Given the fuzzy nature of the problem (we deal with floating point comparisons), we require an algorithm capable of exploiting spatial locality and provide a "good enough" solution in a minimal amount of time. The best-first search algorithm provides this compromise on the types of problems we deal with.

Additional Key Words and Phrases: CFD, NNS, particles

ACM Reference Format:
Christian Lagares. 2021. Efficient tree-based NNS for CFD applications. 1, 1 (April 2021), 6 pages.

1 INTRODUCTION

Computational fluid dynamics (CFD) has become a ubiquitous tool for engineering analysis in many fields [1–4]. Although an in-depth discussion of methods for CFD is outside of the scope of the proposed work, it suffice to say that there are three stages: pre-processing, simulation and post-processing [5]. In the pre-processing stage, a continuous problem is partitioned into discrete nodes (or cells) where the governing equations are solved using numerical techniques. In the simulation stage, the governing equations are solved at these discrete locations. This is the most computationally expensive stage and often requires thousands (or even hundreds of thousands) of CPU core hours to generate sufficient information for unsteady simulations (i.e. simulations evolving in time as well as in space). The present work deals with issues pertaining to post-processing where the results of the simulation are used to generate useful information. Multiple post-processing routines rely on being able to interpolate solution variables at arbitrary locations within a volume [6]. The end goal is to enable offline and online post-processing for large-scale CFD and enable both near-real-time particle effects and high-resolution rendering under a single framework. We isolate the scope of the work to a specific issue within particle tracing, the issue of efficient nearest-neighbor search for low-latency interpolation.

Author's address: Christian Lagares, christian.lagares@upr.edu, University of Puerto Rico at Mayaguez, Mayaguez, Puerto Rico, USA, 00681.

2 THEORY

The category of problem families we aim to solve require particle tracing. For this, a simulated particle is placed at an arbitrary location and it is moved through the domain based on the fluid flow properties from the simulation such as the velocity field at discrete time steps. However, even though great care could be taken to ensure the initial placement of the particle in a node where the velocity is known, there are no guarantees the particle will fall on any given mesh location. Thus, the objective is to find the nearest node to a given particle. At this node, we typically know or have previously approximated the gradient of the variable of interest. Knowing the coordinate of the node and particle, one can perform a linear approximation with a second order convergence about the step size. "Closest" is a relative term; nonetheless, we define closeness by the Euclidean norm of the vector formed between a given node and the particle.

Figure 1 (a) depicts graphically the problem of advecting a particle through a staggered mesh. Figure 1 (b) highlights a single quadrilateral with a single particle inside and provides the notation on the step size on the closest node. We typically approximate derivatives using a central differences approximation which already takes into account neighboring nodes, thus there is no need to re-account for these nodes. We will use equation 1 as our approximation for the flow variables.

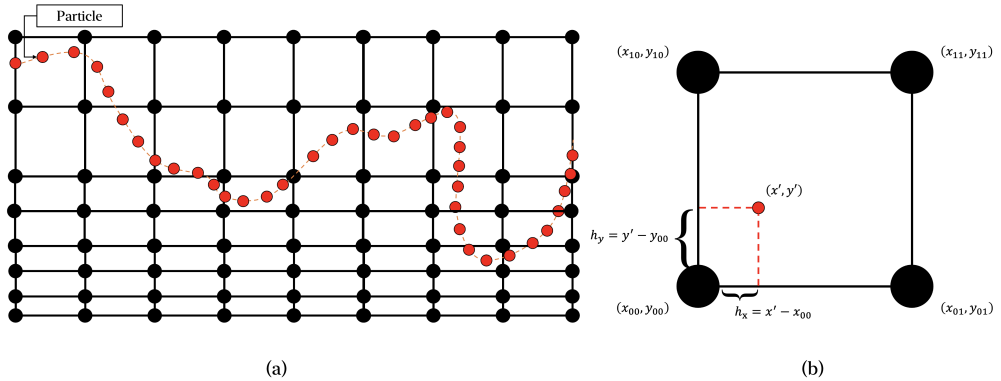

Fig. 1. (a) Particle tracing illustration; (b) illustration of a 2D quadrilateral with an arbitrary location within the quadrilateral; red dot represents a particle along a trajectory within a mesh.

$$u(x_{00} + h_x, y_{00} + h_y, z_{00} + h_z) \approx u(x_{00}, y_{00}, z_{00}) + h_x \frac{\partial u}{\partial x} + h_y \frac{\partial u}{\partial y} + h_z \frac{\partial u}{\partial z} \tag{1}$$

where u represents the velocity field.

2.1 Proposed solution and validation data

2.1.1 Validation: Mesh Visualization. We will evaluate the performance of the implementation in two domains shown in figures 2 & 3. These meshes, although small, provide two varying levels of complexity while keeping computational requirements for the course relatively low. They contain between 2 and 5 million nodes. Further, the second domain contains curved regions which require special attention.

2.1.2 Proposed Solution. We propose representing the mesh as an octree (or quad tree if a 2D representation of the problem is possible). The octree would have a branching factor of 8 and a fixed maximum depth depending on the largest dimension, $d = \log_2(N)$ where N is the largest

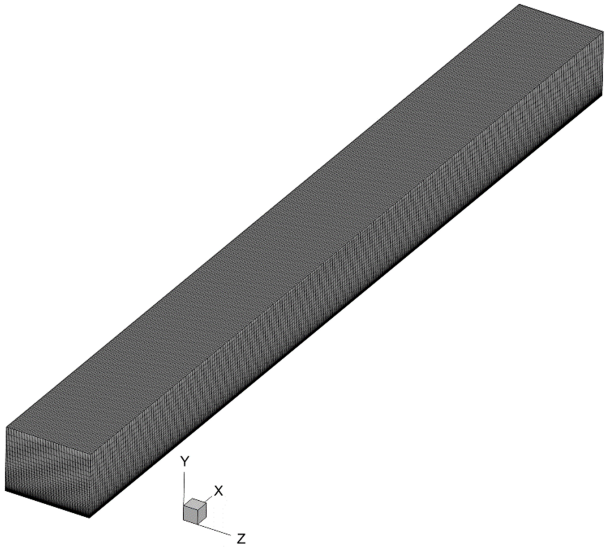

Fig. 2. Domain 1: A Flat Plate domain with ≈2M nodes.

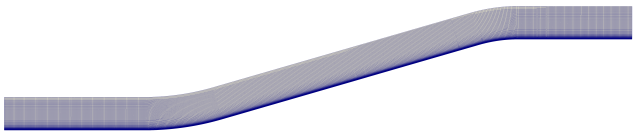

Fig. 3. Domain 2: A complex domain with ≈5M nodes.

dimension of the mesh. Consequently, the overhead of this data structure would be minimal in terms of memory footprint. In terms of algorithms, best-first search and the depth-first algorithms will be explored. However, it is likely that given the tree structure, fixed depth, fixed cost and fixed branch factor that a greedy best-first search would yield optimal and efficient results.

Currently, our best solution is linear search which at worst is an n^3 algorithm.

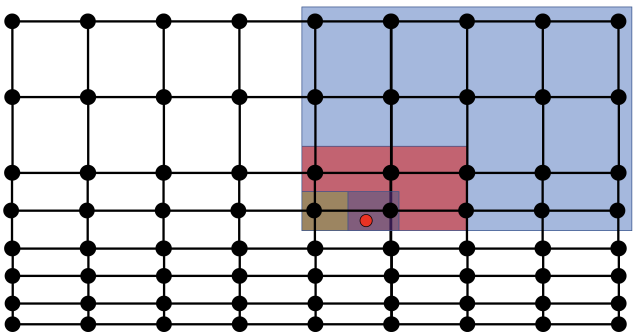

Fig. 4. Spatial structure of a quad-tree; an analogous structure exists for an octree with depth-like divisions.

2.1.3 Platform. We will implement the proposed solution using Python 3.7 [7]. Given the relatively small size of the domains, we will execute the benchmarks on a local machine although the method is easily extensible to a distributed environment [6]. We leverage the Numpy [8] extensively and avoid complex indexing to ensure copy elision and reference semantics within the octree. This should benefit the cache impact on the CPU. All benchmarks are executed on a single CPU thread. The CPU used for this test is an Intel 9980HK which is based on Intel's Coffee Lake Refresh with 16 MB of L3 cache. Lastly, the system has 64 GB's of 2667 MHz DDR4 memory.

3 DISCUSSION

3.1 Comparison of Methods

Initially, we compare three algorithms: naive flat-tree breadth first with neighborhood bounding, depth-first and best-first search. The naive flat-tree breadth first with neighborhood bounding algorithm is our current best implementation and linearly moves through every point and ensures the nearest neighbors have been found by evaluating if a target is contained within a volume formed by a set of points. After this initial test, we will select the two best algorithms based on run time and will execute a benchmark with $O(N_x)$ points to assess a reasonable scenario of a particle traversing a domain and providing a more reasonable estimate of the search time per point and the overhead for a representative number of searches. To avoid falling in the same region of the octree, we randomly vary z and y between the second and second to last nodes in each direction.

Table 1. Initial benchmark for five point search; speedup is measured with respect to flat tree breadth-first search.

	Aggregate Time [s]	**Time per NNS [s]**	**Speedup**
Flat Tree Breadth First Search	54.43210005760193	10.886420011520386	1
Octree Depth First Search	354.58869099617004	70.917738199234008	0.1535
Octree Best First Search	0.002410888671875	0.000482177734375	22,578

Note, the depth-first search is far from optimal because every branch has to be explored given the fuzzy nature of the problem. The algorithm cannot ensure a given point is the closest to a target without exhausting other possibilities.

3.2 Main Benchmark

To establish a more realistic benchmark, we computed NNS across a mesh and avoid a predictable path by randomly selecting vertical (y) and side way (z) points and linearly varying points along the flow, or x, direction. Finding a point should be as close to linear in time as possible to ensure minimal overhead on staggered grids commonly used in CFD applications. The octree structure for this case which has $O(10)$ levels yields an access time of roughly $500\mu s$. If we normalize by the number of levels, the access time per level would be approximately $50\mu s$ per level. A larger mesh, say one with 10 times more nodes along each dimension (i.e. $N_x = 5,650$) would have approximately 13 levels and an access time of about $650\mu s$. This represents a mere 30% increase in computational complexity despite a 1000-fold increase in the number of nodes. Any increase in computational complexity could be estimated by first executing a benchmark analogous to the one made with a maximum number of nodes along any given dimension of N, the increase in runtime should be close to $\log_N(N^*)$ where N^* is the new critical number of nodes.

To asses, the scaling with actual values, we employ the mesh shown in figure 2. For this domain, the octree has a total of 9 levels.

Table 2. Benchmark results comparing the two best algorithms for the domain shown in figure 3.

	Flat Tree Breadth First Search [s]	Octree Best First Search [s]	Speedup
Total Time	1205.869924545288	0.2977714538574219	4,049x
Mean Time	2.141864874858416	0.0005289013390007494	4,049x
Standard Deviation	1.308632942334026	0.00010562296871505838	12,389x
99.999th Percentile	9.993662528862572	0.001162639151291	8,595x

Table 3. Benchmark results comparing the two best algorithms for the domain shown in figure 2.

	Flat Tree Breadth First Search [s]	Octree Best First Search [s]	Speedup
Total Time	724.9867377281189	0.10173702239990234	7126x
Mean Time	1.6552208623929656	0.0002322763068490921	7126x
Standard Deviation	0.1810474584506883	4.417128596121881e-05	4099x
99.999th Percentile	2.741505613097095	0.000497304022616	5512x

Note that the scaling prediction is not fully accurate. This could be attributed, in part, to the "distance" in memory between elements which allow for better caching behavior in the CPU and thus reducing overhead. The first dataset has a size of 54 MB whereas the second has a size of roughly 24 MB. At any given moment, 67% of the second dataset is in cache whereas 29% of the first dataset fits in cache. This drastic difference when coupled to an efficient memory prefetcher and smaller strides between planes (4800 vs 8400) could explain the differences in scaling performance. Nonetheless, the overhead introduce by a best-first search in an octree is minimal (sub millisecond for the two meshes considered).

4 CONCLUSIONS

We have presented and motivated a use case for efficient NNS in arbitrary, semi-structured domains which are ubiquitous in CFD. Floating point NNS is inherently fuzzy and efficient search requires exploiting spatial information to avoid searching every possible branch to guarantee an optimal solution. We presented results for an octree data structure which inherently incorporates spatial locality as part of the data structure. Further, each node of our octree structure incorporates not only child nodes but also stores a reference to the centroid coordinate which facilitates search. We compared three search algorithms: 1) breadth first search on a flat tree, 2) depth first search on an octree and 3) best first search on an octree. We found a notable improvement over our naive search algorithm of roughly 3 orders of magnitude for a representative example and up to 4 orders of magnitude for random searches. We also noted that a minimal overhead (sub millisecond) is introduced per search which is ideal for particle tracing applications to be integrated into real-time visualization platforms. For a 60-75 HZ stereoscopic headset, a budget of 13-16 ms per frame is available. Thus, minimizing latency for each stage allows for a larger budget available for high resolution shading effects including ray tracing. Consequently, our highest mean time per search accounts for 3.3% to 4.07% of the total frame render time budget. For offline rendering and post-processing, the low overhead implies a larger number of particles can be considered, a reduction in processing time can be achieved, larger domains can be considered or a combination of these.

ACKNOWLEDGMENTS

Christian Lagares acknowledges financial support from the Puerto Rico Louis Stokes Alliance for Minority Participation's Bridge to the Doctorate Program from the National Science Foundation under grant no. HRD-1906130. This material is based upon work supported by the Air Force Office

of Scientific Research (AFOSR) under award number FA9550-17-1-0051. This work was supported in part by high-performance computer time and resources from the DoD High Performance Computing Modernization Program.

REFERENCES

[1] J. Slotnick, A. Khodadoust, J. Alonso, D. Damofal, W. Gropp, E. Lurie, and D. Mavriplis, "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences," tech. rep., NASA, 03 2014.

[2] K. Jansen, "https://github.com/phasta," 2015.

[3] C. J. Lagares, K. E. Jansen, J. Patterson, and G. Araya, "The effect of concave surface curvature on supersonic turbulent boundary layers," Presented at the 72^{nd} Annual Meeting of the American Physical Society's Division of Fluid Dynamics, 2019.

[4] C. J. Lagares, K. E. Jansen, and G. Araya, "The re-laminarization of a supersonic boundary layer subject to a strong convex curvature," Presented at the 73^{rd} Annual Meeting of the American Physical Society's Division of Fluid Dynamics, 2020.

[5] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics.* Harlow: Pearson, 2007.

[6] C. J. Lagares, W. Rivera, and G. Araya, "Aquila: A distributed and portable post-processing library for large-scale computational fluid dynamics," *AIAA SciTech*, 1 2021.

[7] P. P. L. Website, "http://www.python.org." Python Software Foundation.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, p. 357–362, 2020.