Scalable Parallel Distance Field Construction for Large-Scale Applications

Hongfeng Yu, *Member, IEEE,* Jinrong Xie, *Member, IEEE,* Kwan-Liu Ma, *Fellow, IEEE,* Hemanth Kolla, and Jacqueline H. Chen

Abstract—Computing distance fields is fundamental to many scientific and engineering applications. Distance fields can be used to direct analysis and reduce data. In this paper, we present a highly scalable method for computing 3D distance fields on massively parallel distributed-memory machines. A new distributed spatial data structure, named *parallel distance tree*, is introduced to manage the level sets of data and facilitate surface tracking over time, resulting in significantly reduced computation and communication costs for calculating the distance to the surface of interest from any spatial locations. Our method supports several data types and distance metrics from real-world applications. We demonstrate its efficiency and scalability on state-of-the-art supercomputers using both large-scale volume data sets and surface models. We also demonstrate *in-situ* distance field computation on dynamic turbulent flame surfaces for a petascale combustion simulation. Our work greatly extends the usability of distance fields for demanding applications.

Index Terms—distance field, in-situ processing, parallel algorithms, scalability, spatial data structures, scientific simulations, geometric modeling, large-scale scientific data analytics and visualization

1 Introduction

Computing distance fields is a fundamental requirement for many algorithms of computer graphics and visualization. Distance fields are also referred to as distance transforms or distance maps. Their usage has been widely found in diverse scientific and engineering fields, such as volume graphics, computer vision, image processing, and computational geometry. Beyond their conventional applications, distance fields also receive new attention in the era of big data. Researchers have shown that distance fields can play a critical role in addressing visualization of large and complex data. For example, it is viable to effectively reduce visual clutter while accentuating visual foci via prioritizing data objects according to their distances to regions of interest [1]. Distance fields can also serve for indexing and compression approaches to managing and exploring data at extreme resolutions [2], [3], [4].

There has been extensive prior work in developing algorithms for computing distance fields. We refer readers to the references [5] and [6] for an overview of 2D and 3D distance field construction and applications. However, most of this research has not focused on the requirements imposed by large-scale scientific applications:

First, there are few general distance field construction algorithms that deal effectively with large data generated from tera-to-petascale scientific simulations. It is imperative to improve the scalability of algorithms to achieve high levels of parallelism for large data processing.

- *Hongfeng Yu is with the University of Nebraska-Lincoln. E-mail: yu@cse.unl.edu*
- *J. Xie and K.-L. Ma are with the University of California-Davis. E-mail:jrxie,klma@ucdavis.edu*
- *H. Kolla and J.H. Chen are with Sandia National Laboratories. E-mail:hnkolla,jhchen@sandia.gov*

Second, extrapolating current technology trends towards the exascale computing reveals the increasing disparity between I/O speed and compute speed [7]. Due to the lagged I/O speed, scientists can only store a small fraction of the data during their detailed modeling and simulation processes, thus possibly missing highly intermittent transient phenomena. A promising solution is to process the data *in-situ* on the same machines as the simulation runs, which can minimize the I/O cost and lead to exploration of data with full extent. This requires new in-situ algorithms to scale as well as simulations when executed with thousands to hundreds of thousands of CPU cores. However, such a scale has not been considered in any existing distance field construction algorithms that are almost exclusively done as an offline *post-processing* step.

Third, many real-world data sets are large in size and heterogeneous in type, structure, and semantics. As a result, it is desired to design distance field representations in support of varied data in a uniform and scalable way. Such representations can also facilitate a wide range of subsequent processing operations while minimizing data storage and transformation overhead. This requirement, however, has rarely been addressed in previous work.

It is non-trivial to design scalable solutions for computing distance fields within massive and complex data. Although researchers have exploited parallel and distributed computing in distance field construction, the existing algorithms are typically characterized with high memory access and intensive communication overhead in a distributed environment. Additionally, data elements are at risk of being unevenly partitioned and distributed over space. This produces difficulties in achieving a balanced workload among a large number of processors using the existing algorithms.

In this paper, we present a highly scalable algorithm for computing distance field of large-scale applications. We design a new spatial hierarchical data structure, named *parallel* *distance tree*, that allows us to efficiently capture, track, and manage the essential information of data, and minimize the communication and computation costs across processors to compute the distance field. In addition, our method supports multiple data types including polygonal objects, point clouds, or volumetric data. It is also flexible with different distance metrics, including Euclidean distance, City block distance, and Chessboard distance. Thus, our method can be used in a wide range of real-world large applications with minimal implementation efforts. We have conducted case studies using different 3D data sets, and shown that our achievement is generalizable and beneficial to researchers from different areas.

We have also integrated our method with real-world large simulations. This enables scientists to compute distance fields in-situ and capture highly intermittent and detailed phenomena that were hardly perceived in post-processing. Our method does not depend on any particular architectures, and the experiments have been conducted on state-of-the-art supercomputers. Our results have scaled up to 69,120 CPU cores of parallel distance field constructions, and clearly demonstrated the improvement over the previous state-of-the-art.

2 Related Work

The research work relevant to our problem includes parallel distance field construction, distance field representation, parallel octrees, and in-situ processing. Little previous research has been done to address all aspects of the problem in the context of large scale scientific and engineering applications.

Parallel Distance Field Construction. Researchers have developed a number of parallel algorithms to compute distance fields. Most of them are designed for SIMD architectures [8], [9], parallel random access machines [10], reconfigurable systems [11], and GPUs [12], [13], [14], [15], [16]. Although noticeable performance numbers have been reported, these approaches cannot be optimally scaled with respect to the size of data and the number of processing units due to the inherent memory model of the architectures [17]. Distributed memory MIMD machines are more general purpose and enable algorithms scale to large data. Bruno et al. [18] presented a parallel implementation for the 2D Euclidean distance transform (EDT). They provided the speed-up results with different pixel numbers using 4 processors. Lee et al. [19] presented an optimized algorithm for the 3D EDT; however, the speed-up was only achieved for 3 processors. Torelli et al. [20] used data compression to reduce communication cost, and demonstrated the scalability up to 10 processors. Additionally, these research efforts only examined 2D images or 3D volume data sets, and represented distance fields in pixel or voxel grids. Thus, the stability and applicability of these various approaches to large data sets are limited given the scale of system and the size of data that are considered in our study.

Distance Field Representation. Using spatial hierarchies is a simple but effective way to represent distance fields. Quadtree and octree based approaches are most relevant to our work. Samet [21] used a quadtree representation to compute the chessboard distance transform of 2D images. Strain [22] presented a similar approach that uses quadtree to solve the redistancing problem with the max-norm distance. Frisken et al. [23] proposed the Adaptive Distance Fields method that uses octree to subdivide space and sample a continuous distance field. Bounding volume hierarchy (BVH) is another commonly used hierarchical technique to organize geometric objects. Researchers have proposed different acceleration methods for different geometric models [24], [25], [26]. Although their applications are similar to ours, these methods are mainly designed for collision detection and minimal distance query, rather than distance field construction.

Parallel Octrees. Parallel octree structures have been used in many applications. Zhou et al. [27] presented a data-parallel octree method for surface reconstruction on a single GPU. However, for scalable performance on distributed systems, there is a need to establish and maintain the correlation of octrees across a large number of processors. Parallel octrees are also frequently used in scientific applications, such as N-body simulations [28], [29] and earthquake modeling [30]. However, these applications do not require to organize objects for parallel distance field calculation.

In-situ Processing. The increasing performance gap between compute and I/O capabilities has motivated recent developments in in-situ data processing. Researchers have directly integrated operations, including visualization [30], [31], and statistical compression and queries [32], into simulation routines to operate on in-memory simulation data. In this work, our solution can be as scalable as the simulations, and make it feasible to construct distance fields in-situ.

3 Background

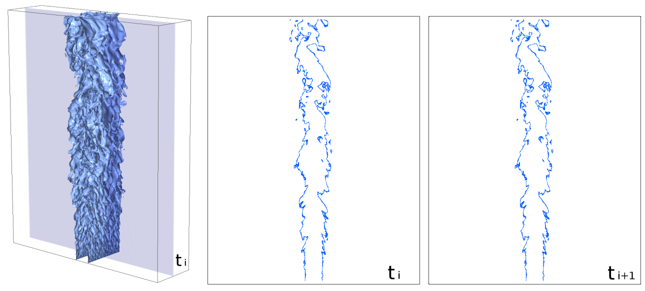
A distance field is defined as the following: Suppose we have a set Γ consisting of *elements* in $\mathbf{R}^3$, where elements can be polygonal objects, point clouds, or volumetric data. The value at each point $\mathbf{p}$ of a distance field domain $\Omega \subset \mathbf{R}^3$ is the distance from $\mathbf{p}$ to its nearest element of Γ:

$$df_\Gamma(\mathbf{p}) = \inf_{\mathbf{x}\in\Gamma} dist(\mathbf{x}-\mathbf{p}), \tag{1}$$

where the distance function *dist* is application specific, and the commonly used ones include the Euclidean distance, the city block distance, and the chessboard distance.

Figure 1 (a) shows an example of application requirement from a large-scale combustion simulation. The bounding box corresponds to the 3D simulation domain, and the blue surface is an isosurface extracted from one time step of data. Using this surface as Γ, a scientist wants to compute the distance from any point inside the domain Ω to the surface.

It is very challenging to achieve optimal parallel efficiency to compute distance field with large and complex data in a distributed environment [18], [19]. This is first because distance field construction is characterized with high memory access. As shown in Equation 1, in the worst case an exhaustive search of elements may be required to find the nearest element of a point. As such, in a distributed environment where each processor may hold a portion of the elements, this can incur intensive message exchanges among processors. In addition, data elements can be unevenly partitioned and distributed over space, and thus it is difficult to achieve balanced workload among processors.



(a) (b) (c)

Fig. 1. (a) shows a 3D rendering of an isosurface extracted from a combustion data set at a time step t_i. We may only need to search a small portion of the surface to compute the distance at any point. (b) and (c) show the slices of the surface evolved at two consecutive time steps, t_i and t_{i+1}. The difference between the two surfaces is around 0.1% of the surface area. Only a small fraction of distance field may need to be updated accordingly.

3.1 Spatial and Temporal Coherence

To address these issues, we are inspired by the spatial and temporal coherence inherent in the data of modelings and simulations. Given the example shown in Figure 1, we have the following observations. First, by leveraging the spatial coherence (Figure 1 (a)), we may only need to search a set of nearby elements to compute the distance at any point. In a distributed environment, this implies that it is possible for one processor (PE) [1] to only communicate with a fewer number of vicinal processors, thus reducing communication costs. Furthermore, Equation 1 shows that the overall computation cost is proportional to the number of elements, and thus we may achieve more balanced workload by evenly partitioning and distributing the workload with respect to the elements and the distance field domain. Second, by leveraging the temporal coherence (Figure 1 (b) and (c)), we may only need to update a small fraction of distance field with respect to the changes of elements, which can further reduce the communication and computation costs over time.

3.2 Octree-based Distance Field Construction

At first glance, it may appear straight-forward to first build a spatial hierarchy, such as an octree, to index the set Γ of elements by subdividing the distance field domain Ω adaptively, and then, for a point $\mathbf{p}$, we can query the tree to find its nearest point among the elements. However, the elements within the same octree node as $\mathbf{p}$ may not contain the nearest point to $\mathbf{p}$. Thus, we need to continue to go through the nearby octree nodes that can be at different tree levels, and this operation will require additional overhead.

To solve this problem, Strain [22] modified the way to build an octree to facilitate the nearest point query. For each octree node C with the center $\mathbf{c}$ and the edge length $2r$, Strain defined the *concentric triple* (or shortened as *triple*) T of C as:

$$T = \{\mathbf{x} \in \mathbf{R}^3 : dist(\mathbf{x} - \mathbf{c}) \leq 3r\}. \tag{2}$$

Based on this definition, an octree is built from a root node that encloses Γ. Each tree node has an element list, and initially the root's element list contains all elements of Γ. A tree node is recursively split if its triple intersects Γ. For each new child node, we check every element in the parent's element list, and add it to the child's element list if the element intersects the child's triple. We continue this recursive splitting until a tree node has an empty element list or the tree depth reaches a maximum criterion.

After building such an octree, for a point $\mathbf{p}$, we can locate the leaf node C containing $\mathbf{p}$, and efficiently compute the distance of $\mathbf{p}$ to Γ through three steps:

1) Find the minimal distance m_1 from $\mathbf{p}$ to the elements of Γ in the element list of C.
2) If the element list of C is empty or m_1 does not satisfy
$$\{\mathbf{x} \in \mathbf{R}^3 : dist(\mathbf{p} - \mathbf{x}) \leq m_1\} \subset T, \tag{3}$$
where T is the triple of C, it means that there are possibly some elements outside T but having the minimal distance to $\mathbf{p}$. Then we find the minimal distance m_2 from $\mathbf{p}$ to the elements of Γ in the element list of the parent node C' of C.
3) If m_2 does not satisfy
$$\{\mathbf{x} \in \mathbf{R}^3 : dist(\mathbf{p} - \mathbf{x}) \leq m_2\} \subset T', \tag{4}$$
where T' is the triple of C', then we find the minimal distance m_3 from $\mathbf{p}$ to the elements of Γ in the element list of the grandparent node C'' of C.

The correctness of this procedure has been proved in [21], [22]. This procedure can provide the *exact* distance value at any $\mathbf{p}$. Alternatively, we can also first compute the distances at the octree vertices and then obtain the distance at $\mathbf{p}$ using interpolation. Such an octree equipped with the vertex distances and the element lists is called a *distance tree* that can provide both exact and interpolated distance values [22].

4 Parallel Distance Tree

Our intuition is to extend the octree-based method to index data distribution across processors and exploit the spatial and temporal coherence. A straight-forward solution is to let each processor first compute a global distance tree, and then use the tree to query the vicinal processors to construct the distance field collaboratively. However, this requires each processor to collect the information of global data distribution before global tree construction. Such a collective operation often incurs a significant amount of communication. In addition, a full-grown global distance tree is typically too large to be handled by a single processor. Hence, it is non-trivial to efficiently construct and maintain distance tree in parallel, and use it to derive workload assignment and communication schedule.

We introduce a novel distributed spatial data structure, named *parallel distance tree*, to index the set Γ of elements and the distance field domain Ω for parallel distance field

1. A processor refers to a single-core processor or a core in a multiple-core processor.

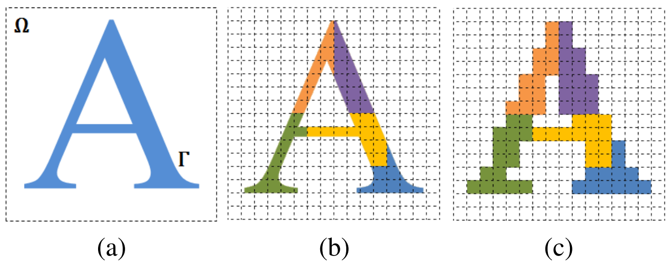


Fig. 2. (a) shows a set Γ of elements in the domain Ω. (b) shows that the elements are partitioned and distributed among 5 processors in a block fashion. The assignments of Γ among the processors are represented in different colors. Each processor fills a bitmap according to its local element block distribution. After a collective reduction operated on the bitmap, each processor obtains the information of the global distribution of the element blocks and the empty blocks, as shown in (c).

construction. Based on our assumptions (Section 4.1) and data structures design (Section 4.2), our strategy is to first build a coarse global distance tree at each processor (Section 4.3). This step only needs a coarse-grained description of global data distribution at each processor, and the memory and communication cost is marginal. We use the global tree to derive a balanced assignment of local trees among the processors (Section 4.4). Each processor can use its local tree to independently find the processors that it needs to communicate with, and establish the communication schedule without message exchanges (Section 4.5). Each processor then independently constructs a full-grown local instance of the underlying global tree (Section 4.6). A local instance is also a distance tree that contains the information of local and remote elements, which enables the processors to collectively compute the vertex distances of their local distance trees with the exploitation of data parallelism (Section 4.7). Our method also takes advantage of temporal coherence in time-varying simulation data so that only a small subset of the distance tree can be updated (Section 4.8). The final distance field is organized and stored in a distributed fashion that can facilitate large data analytics. We have integrated our method with a real-world simulation (Section 4.9), and explored performance acceleration (Section 4.10).

4.1 Assumptions

Given a set Γ of elements in $\mathbf{R}^3$, we assume that the distance field domain Ω is rectangular and encloses Γ, as shown in Figure 2 (a). (For clarity, we use 2D elements, domains and quadtrees in the figures and examples.) The scalability of parallel distance field construction depends on both partitioning and distribution of Γ and Ω.

Partition and distribution of Γ is typically application specific. Without loss of generality, we assume that Γ is partitioned in a representative block fashion [33]. Each block is rectilinearly oriented and can contain multiple elements. The bounding boxes of all blocks have the same size and shape. The blocks can be assigned to the processors in different fashions. For post-processing, the block assignment can suit the needs of parallel distance field construction, and the amount of elements at each processor can be roughly the same to achieve balanced workload. For in-situ processing, the elements of Γ are generated during modelings or simulations. The partition and distribution of Γ is dictated by the applications that generate data, which can be highly uneven among processors: some processors may have multiple blocks of elements, while some processors may have none.

Given the block-based data partition scheme, we assume that the rectangular domain Ω consists of a set of blocks, $\{1,...,b_x\} \times \{1,...,b_y\} \times \{1,...,b_z\}$, where b_x, b_y, and b_z are the numbers of blocks along each axis. A block can be an *element block* that contains the elements, or an *empty block* without any elements. Each element block is assigned to a processor. Figure 2 (b) shows an example that the set Γ in Ω are divided and assigned to 5 processors.

4.2 Data Structures

The information of global data distribution is imperative for distance field construction. However, it is infeasible to collect precise distribution of elements at each processor for a large data set. We address this issue by letting each processor gather the coarse-grained information at a block level. To this end, the first data structure that each processor has is a bitmap which records the global distribution of element blocks and empty blocks. The bitmap contains $b_x \times b_y \times b_z$ bits. Each bit corresponds to a block in Ω, and is set to 0 or 1 for an empty block or an element block, respectively. Each processor first initializes its bitmap by filling 0s. Then, if a processor contains the elements, the corresponding element blocks are marked as 1 in the bitmap. Finally, each processor combines the bitmaps from all processors. This can be easily implemented using a *collective reduction* routine with the bitwise OR operation, such as the MPI_ALLREDUCE function. The size of the bitmap is marginal. For example, a bitmap of 128KB can represent more than 1 million blocks which is sufficient for current large-scale supercomputers. Each processor then obtains an identical bitmap that records the global distribution of the element blocks and the empty blocks in Ω. Figure 2 (c) shows the bitmap of the global block distribution obtained by each processor, given the partitioning and distribution of Γ among 5 processors in (b).

Each processor also has two arrays to store the information of elements. The first array is an element array, *elementarray*, which records the local elements. The element array is empty if a processor does not have any local elements. The second array is an element block array, *elementblockarray*, which records the global element blocks. Each element block *blk* contains its index *blk.id* in the bitmap and the processor ID *blk.proc* to which the *blk* is assigned.

Finally, each processor has a local instance of the underlying global distance tree. A local instance is also an octree with the standard *linear octree* representation [34]. It consists of two arrays, the vertex array and the node array. The vertex array, *vertexarray*, records the octree vertices. Each vertex *v* contains:

- $v.key$: the locational key.
- $v.node$: the index of the node that creates v in the node array.
- $v.coord$: the coordinate of v in Ω.
- $v.distance$: the distance value at v.

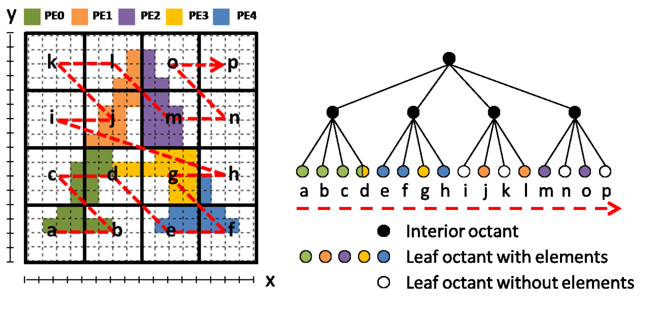
The node array, *nodearray*, records the octree nodes. Each node n contains:

- $n.key$: the locational key.
- $n.boundingbox$: the geometry of n's bounding box.
- $n.vertices$: the indices of n's vertices in the vertex array.
- $n.elements$: the indices of the elements that intersect n's triple in the element array.
- $n.elementblocks$: the indices of the element blocks that intersect n's triple in the element block array.
- $n.processors$: the IDs of the processors whose element blocks intersect n's triple.
- $n.location$: the flag indicating if n is *local* or *remote*.
- $n.parent$, $n.children$: the indices of n's parent and children in the node array.

We note that the *locational key* [34], [35], also known as the *shuffled zyx key* [36], is a commonly used linear octree technique to encode and unambiguously distinguish octants[2]. For an octant at the depth m, the locational key is a bit string, $z_1y_1x_1z_2y_2x_2\cdots z_my_mx_m$, indicating the path from the root to this node. The convention is that if the bit x_i is set to 0, the child at the depth i covers the left side in x of its parent; otherwise, it covers the right side. The interpretation of the y and z bits are similar. The key of each vertex is computed using the same convention. We use a 4-byte integer for a locational key, and the maximum tree level is 10.

4.3 Initial Coarse Global Distance Tree Construction

Each processor independently builds an initial coarse distance tree to subdivide Ω with respect to the information of the element blocks provided by the bitmap. Each processor builds the tree in a similar way as the one in Section 3.2. The main differences are that the tree is built in the breadth-first order and only the element blocks are considered. Each processor starts with a root node, *root*, to cover Ω. The *root.elementblocks* contains all the element blocks, and the *root.processors* contains the IDs of the processors that have the element blocks. Then we recursively split the nodes in the breadth-first order if a node's triple intersects the element blocks. For each new child, we check every element block *blk* in the parent's *elementblocks* list, and add *blk* into the child's *elementblocks* if *blk* intersects the child's triple. We also check every processor *proc* in the parent's *processors* list, and add *proc* into the child's *processors* if *proc*'s element blocks intersect the child's triple. We continue this procedure until the ratio of the leaf node number to the processor number is greater than ε ($\varepsilon = 3$ in our current implementation) or the maximum tree depth has been reached. Because all processors have an identical bitmap, they generate an identical initial global distance tree after this step.


y
PE0
PE1
PE2
PE3
PE4
k
l
o
p
i
j
m
n
c
d
g
h
a
b
e
f
x
a b c d e f g h i j k l m n o p
Interior octant
Leaf octant with elements
Leaf octant without elements

(a) (b)

Fig. 3. (a) shows that the decomposed global distance tree leaf octants (as indicated by the black square) are assigned to the PEs whose element blocks intersect the leaf octants bounding box. It is possible that one leaf octant is assigned to multiple processors, such as the leaf octant d. All the leaf octants are naturally ordered based on their locational keys, so that the traversal of all the leaf octants follows a Z-order space-filling curve that preserves the spatial locality of the leaf octants. (b) shows a quadtree corresponding to the spatial partition of (a). The traversal of the leaf octants from left to right in (b) is equivalent to the zigzag one in (a).

4.4 Leaf Octants Assignment

Each leaf octant in the initial global tree corresponds to a region of Ω (see Figure 3 (a)), and is associated with the workload of computing the distance field. We assign the leaf octants among processors. The leaf octants are not necessarily continuous in *nodearray*. We scan *nodearray* and collect the leaf octants into a temporary array *leafarray* while preserving their orders in *nodearray*. The leaf octants are naturally ordered according to their locational keys because of the linear octree technique. This order is identical to the pre-order traversal of the leaf octants as shown in Figure 3 (b). If we traverse the leaf octants in this order in Ω, we follow the well-known *Peano space-filling curve* which can cluster the spatial nearby octants together, as shown in Figure 3 (a). This property facilitates partitioning and distribution of the workload among processors using a two pass procedure.

In the first pass, we assign the leaf octants whose *processors* lists are nonempty. Each of these leaf octants has the elements within its region. We assign such an leaf octant to the processors in its *processors* list to minimize data movement. As show in Figure 3, some leaf octants may be assigned to one processor, such as e, f and h that are assigned to PE_4, and some leaf octant may be assigned to multiple processors, such as d that is assigned to PE_0 and PE_3.

In the second pass, we assign the leaf octants with the empty *processors* lists. Each of these leaf octants has no elements within its region, and we call it an empty leaf octant, such as the octants i and k in Figure 3. There are two cases for assigning the empty leaf octants:

In the first case, all processors have already been assigned

2. We use octant and octree node interchangeably.

with workload in the first pass. For example, as shown in Figure 3 (b), all five processors have been assigned with the leaf octants. Then, we assign each empty leaf octant *oct* to the processor that is responsible for the *oct*'s neighboring leaf octants in $leafarray$ and these neighboring leaf octants have the closest ancestor with *oct*. For example, in Figure 3 (b), the empty leaf octants *i* and *k* will be assigned to PE_1, because the octants *i*-*l* are neighbors with the same ancestor, and *j* and *l* are assigned to PE_1 in the first pass. Similarly, the empty leaf octants *n* and *p* will be assigned to PE_2.

In the second case, some processors have not been assigned workload in the first pass. This is a typical case for in-situ processing, where the region-of-interest can be extracted or identified on only a small fraction of processors. In this case we adopt a simple rule that we evenly partition and distribute the empty leaf octants according to their order in $leafarray$ among the rest of processors. This rule can generally provide well balanced workload in practice because (1) we build the initial distance tree that has a sufficient large number of octants with respect to the processor number, and (2) the leaf octants are ordered along the spacing-fill curve. For example, in Figure 3, assume there are another two processors, PE_5 and PE_6, who do not have any elements and have not been assigned workload in the first pass. Then we assign the octants *i* and *k* to PE_5, and *n* and *p* to PE_6, and the spatial locality of the assignment is ensured.

Each processor independently performs the same procedure to compute the workload assignment. Thus, each processor obtains an identical assignment of leaf octants of all processors. Then a processor goes through each octant *oct*, sets $oct.location = local$ if *oct* is assigned to itself, and otherwise sets $oct.location = remote$.

4.5 Communication Schedule

After the leaf octant assignment, each processor PE_i takes charge of a sub-region ω_i of Ω which is the union of its assigned octants. In order to compute the vertex distance, PE_i will first compute the minimum distance from each local vertex to the local elements in PE_i's *elements* list. However, such a local minimum vertex distance is not necessarily the global minimum. Each processor needs to further go through the nearby but remote octants, and compute the minimum distances from the local vertices to the elements in the remote octants. To achieve this, PE_i needs to send its local vertices to its neighboring processors, outsource the distance computation to those processors, gather the results and finally find the minimum distance value.

A communication schedule plays an indispensable role that helps each processor to identify the neighboring processors that it will exchange the vertices with. Each processor PE_i builds a *sent_table* and a *receive_table* for the communication schedule. Each entry of the tables contains the IDs of the remote processors that PE_i needs to exchange data with, and the corresponding data buffer for sending or receiving. The communication schedule is built based on the information stored in the initial global distance tree:

For *send_table*, PE_i scans its assigned leaf octants. For each of them, $leaf$, the $leaf.processors$ list contains the remote

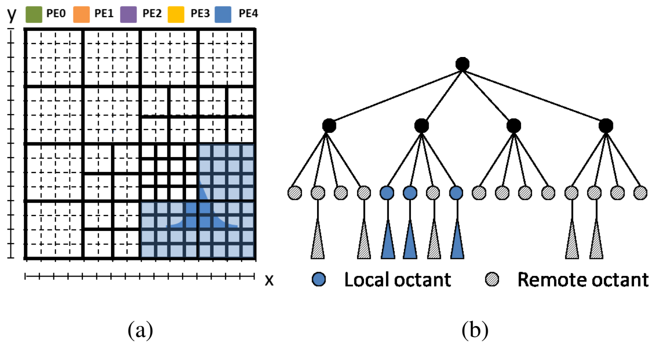


Fig. 4. The light-blue shaded region in (a) is a set of local leaf octants (4x4 square) assigned to PE4. (b) shows the corresponding full-grown local distance tree at PE4.

processors whose element blocks intersect $leaf$'s triple. These processor IDs are then added in *send_table*, because their elements need to be considered and PE_i needs to send local vertices to them. If $leaf.processors$ is empty, PE_i checks each of $leaf$'s ancestors along the path to the root until PE_i finds the first nonempty *processors* list, and adds the processor IDs in *send_table*.

For *receive_table*, PE_i simply returns an empty one if it has no local element, as no neighboring processors will send data to PE_i in this case. Otherwise, PE_i scans the global tree and identifies the remote leaf octants or their closest ancestors whose *processors* lists contain PE_i's ID. This means that PE_i's element blocks intersect the triples of these remote leaf nodes, and their host processors will send message to PE_i to request distance computation. In this case, these processor IDs are added in *receive_table*.

4.6 Full-grown Local Distance Tree Construction

The initial global tree is constructed according to the information of element blocks, and thus is coarse-grained. To compute the distance field, each processor independently grows the initial distance tree with respect to its local elements and the global element blocks in two passes:

In the first pass, each processor PE_i propagates its local elements along the initial global tree, where each tree node's *elements* list records the elements that intersect its triple. After this pass, the *elements* and *elementblocks* lists of each tree node are filled with the elements and the element blocks intersecting its triple, respectively.

In the second pass, each processor PE_i grows its distance tree in a similar way as the one in Section 3.2. For each one of PE_i's local leaf octants, it is recursively split if there are still elements or element blocks within its triple. During the splitting, if the parent's elements or element blocks intersect the child's triple, they will be added into the child's *elements* or *elementblocks*, respectively. For the other remote leaf octants in the initial tree, they are recursively split in the similar way but with only considering PE_i's *elements*. The recursive procedure stops when the maximum tree depth is reached or no more octant can be split.

Figure 4 shows the full-grown local distance tree at PE_4 from the example in Figure 3. We can see that, apart from the refined local octants, some neighboring remote octants are refined according to PE_4's elements as well. These refinement will facilitate the following distance field computing.

4.7 Distance Field Computing

Having constructed the communication schedule and built full-grown local distance tree at each processor, we continue to compute the vertex distances in parallel.

First, each processor independently computes the distance values from the local vertices to the local elements using the method in Section 3.2 that is flexible with different distance functions. Given a local vertex v, it can be shared by multiple octree nodes. We assume that the node n contains the element that has the local minimum distance min_dist to v. min_dist is the global minimum if and only if it satisfies

$$min\_dist \leq 2 \times n.radius \tag{5}$$

Otherwise it implies that there possibly exists an element elm closer to v, and elm can be in the triple of n's parent but not the triple of n. Strain [22] showed that to find the minimum, searching the element list of leaf node's parent is sufficient for the chessboard distance, and searching the element list of leaf node's grandparent is sufficient for the Euclidean distance. Figure 5 shows the case when Equation 5 is violated and searching the element list of the parent is required. In the case of our parallel distance tree, this means that for the chessboard distance, v needs to be sent to the processors in the *processors* list of n's parent; and for the Euclidean distance, v needs to be sent to the processors in the *processors* list of n's grandparent. Thus, for each v whose local min_dist does not satisfy Equation 5, its locational key is added into the entries of its destination processors in $send\_table$ (Section 4.5). Given our communication schedule, these vertex keys are exchanged in a bundle. This significantly reduces the communication cost when the number of exchanged vertices is large. In addition, a locational key is much compact than a coordinates value and can be used to further reduce the communication cost.

After exchanging the vertex keys, each processor translates the received vertex keys into the coordinate values and compute the minimum distance from every remote vertex to its local elements. Once each processor completes calculation, they exchange the results back to their neighboring processors according to the communication schedule but in a reverse direction. Finally, after receiving the remote vertex distances from its neighboring processors, each processor updates the min_dist values if the remote vertex distances are smaller. Up to this point, all min_dist values are the global minimums. We note that the vertex distances are stored in a distributed fashion and the local distance fields of all processors collectively cover the entire domain. We can use the vertex distances to interpolate the distance at any point $\mathbf{p}$ via the distance tree method in Section 3.2. On the other hand, our method to compute vertex distances can be easily applied to compute the exact distance at any point. Therefore, our parallel distance tree can provide both exact and interpolated distance values at users' disposal according to their precision requirement.

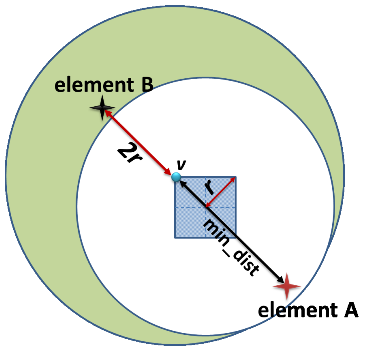


Fig. 5. The blue shaded square corresponds to the current node of interest, n, and its radius is r. The white circle is the *concentric triple* of n. For simplicity, A is the only element within n's *concentric triple* and hence the only entry in $n.elementarray$. We assume that the minimal distance min_dist from vertex v to A within the triple is between $2r$ and $3r$. Thus, there is possibly an element B which is outside the white circle but whose distance to v is shorter than min_dist. In this case, it is necessary to continue searching the elements of n's parent to find B.

4.8 Tree and Distance Field Updating

Detailed modelings and simulations are typically characterized by temporal coherence in their output time steps. This gives us two important implications. First, only a marginal portion of the tree structure needs to be updated with respect to the field evolution between two consecutive time steps. Second, we do not need to update the distance field of a region if there are no field changes within the region's triple.

Based on these implications, we can significantly reduce the cost for updating the tree and the distance field compared with the cost of initial tree construction and full distance field computing. At a time step when distance field construction is evoked, each processor first updates its local elements, fills its local bitmap, and collectively reduces the global bitmap in the same way as we described in Section 4.2. Each processor also records the global bitmap of the previous time step. By comparing the global bitmaps of the current and previous time steps, each processor can detect the blocks that are changed. For a block whose value is changed from 1 to 0 in the bitmap, it means that this block contains no elements at the current time step, and needs to be removed from the distance tree. For a block whose value is changed from 0 to 1, it means that this block contains the elements and needs to be added into the tree.

For a processor that identifies its local changes, we also need to update its element list. For a point or volumetric data set, we can easily track the updated elements according to their coordinates. For the polygonal data, however, we do not track the updated elements, but simply replace the elements with the newly identified polygons for maintaining the accuracy of Euclidean distance computing.

If a processor detects that its lists of element blocks or elements are changed, it then scans the distance tree, and removes/adds the element blocks or the elements from/into the tree nodes. After modification, for a leaf node that contains no element blocks or elements, it needs to be merged with its siblings in a bottom-up fashion, until the new leaf node satisfies the stopping criteria of tree splitting (Section 3.2). For a leaf node that contains new element blocks or elements, it needs to be further split, until the stopping criteria of tree splitting are satisfied.

In practice, the amount of leaf nodes that need to be adjusted is marginal compared with the overall tree size. In our current design, we do not change the assignment of spatial regions among the processors. This means that, for each processor, the assigned regions remain the same, but the distance tree with the regions might become coarser or finer. After tree updating, only the processors having the elements changed in its local and/or triple regions need to update their communication schedules. The rest of steps of communication schedules updating and distance field computing are the same as what we described in Sections 4.5 and 4.7.

4.9 Integration with Simulation

We integrate parallel distance field construction with a petascale combustion simulation based on the APIs developed in our previous work [31]. Through the APIs, the simulation provides the size and coordinates of each processor's global domain and local partition. The simulation also provides the pointer to the buffer of the local field data. The distance field construction module is initialized and invoked by the solver at a given rate. Our integration method can avoid interference between the simulation and the distance field construction.

4.10 Acceleration

The performance of our method can be further improved using GPU or multithreading acceleration. The most intensive computation in our method is to compute the distance values at the tree vertices. For example, the Euclidean distance calculation between the points and the polygons can be accounted for nearly 90% of the computation cost of a processor. However, we can easily parallelize the distance calculation using GPUs or multithreading. This is first because the computation associated with each vertex is independent with each other. Secondly, the linear representations of the distance tree and the elements can facilitate concurrent data access. We use OpenMP in our current implementation: each thread is assigned to be responsible for a set of vertices, and the linear tree and the element arrays are shared among the threads. This approach has achieved noticeable speedup in practice.

5 Results

We use four data sets with different data types in our experimental study. The first two combustion data sets are volumetric data generated from the turbulent combustion simulations performed at the Sandia National Laboratories. The Car data set is a geometric model used in a computational fluid dynamics (CFD) simulation for car design. The Boeing 777 data set is generated from a geometric CAD model constructed for the Boeing Company.

Data Set	Data Type and Scale	Mode
Combustion	volume (1.3B grid points)	in-situ processing
Combustion	volume (1.6B grid points)	post-processing
Car	polygonal (3.4M triangles)	post-processing
Boeing 777	polygonal (350M triangles)	post-processing

TABLE 1
The data sets used in our evaluation.

The distance field domain, Ω, of each data set is evenly partitioned and distributed among the processors. The distance values are stored at the vertices of the parallel distance tree. We use the Euclidean distance that has more communication and computation requirements compared with the other distance functions. For the volumetric data, we first use the features consisting of a set of voxels. The distance values to the features such as an isosurface can be approximated as the Euclidean distances between the points and the voxels covered by the isosurface, which is commonly used in 2D and 3D image processing [5], [6]. We also explicitly extract the polygonal surfaces of the features using the parallel marching cubes algorithm on-the-fly, and compute the exact Euclidean distances between the points and the polygons, which meet the precision requirement for a detailed simulation but increase the computation cost for our evaluation study. Furthermore, to test the interpolation performance, we generate a distance field volume by regularly sampling the resulting parallel distance tree. For the combustion data sets, each distance field volume has the same resolution as the original volumetric data. For the polygonal data set, we use a high resolution value for the distance field volume to sufficiently capture the fine structures. We test both in-situ processing and post-processing to verify the parallelism of our approach. Table 1 lists the data sets, the data types and scales, and the processing modes.

5.1 Application Results

The first combustion data set has a spatial resolution of $2025 \times 1600 \times 400$ and each grid point contains 27 variables. Figure 6 shows an example of distance-based visual analytics for this data set, where two variables, temperature (T) and hydroxyl radical (HO_2), are used. Figure 6 (a) shows an overview of the data. The main flame structure corresponds to the T surface at a particular isovalue. We use the geometry of the isosurface as the elements to construct the distance field. After generating the distance field, a scientist can assign the opacity values to the voxels at the given distance threshold, and control the amount of information displayed around the surface to better observe variable relationships and fine structural information of small turbulent eddies, as shown in Figure 6 (b) and (c). This distance-based visualization can clearly separate previously hidden features that exist at the interior of larger structures. Figure 6 (c) and (d) show the results of two consecutive time steps, where we can perceive the smooth evolution of structures as we construct in-situ distance field construction at a high temporal frequency. This approach not only allows scientists to see previously hidden features but

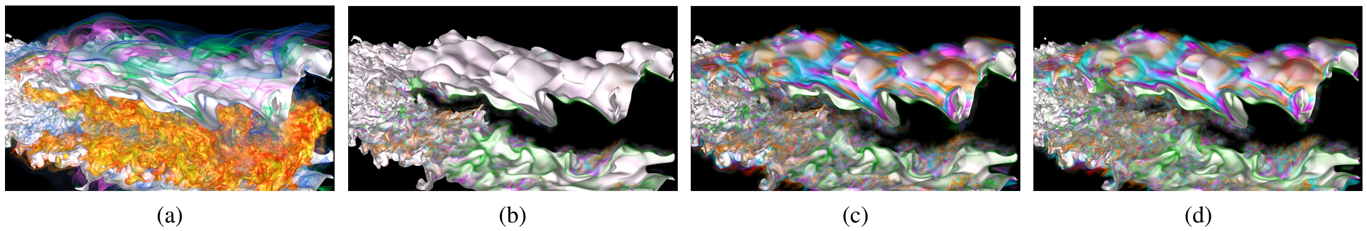


Fig. 6. (a) shows the simultaneous rendering of two variables, temperature (T) and hydroxyl radical (HO_2) of the first combustion simulation data set. The T surface at an isovalue is white; the HO_2 variable is volume rendered. We construct the distance field based on the isosurface. (b) and (c) show the scientists can interactively change the distance threshold and control the amount of information of HO_2 displayed around the T surface to better observe variable relationships. Following this natural coordinate system of a flame, the scientists can see the interaction of small turbulent eddies with the preheat layer of a turbulent flame, a region that was previously obscured by the multi-scale nature of turbulence. (c) and (d) show the results of two consecutive time steps, which convey the smooth evolution of eddies via in-situ distance field construction.

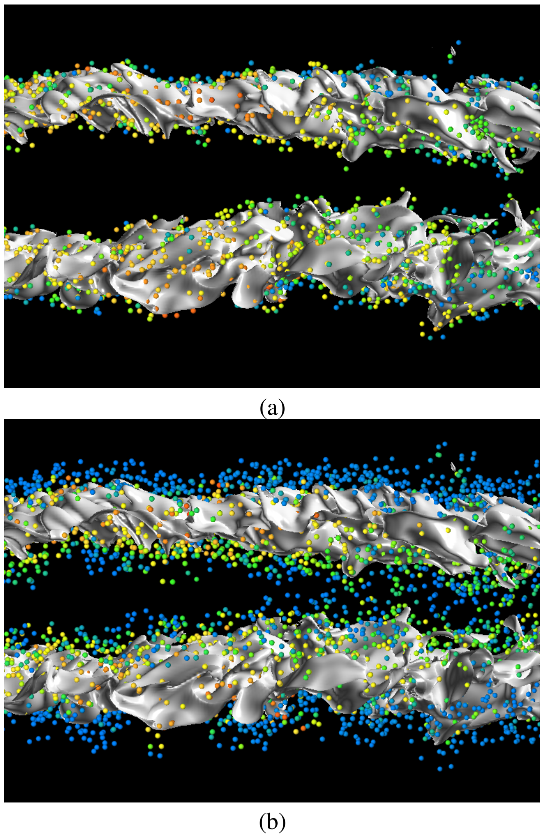


Fig. 7. During a combustion simulation, the distance field is constructed based on an isosurface of the T variable. (a) and (b) show that the scientists can control the initial placement and amount of particles with respect to the different distance thresholds. The particles are colored according to the values of HO_2.

also enables more aggressive encoding of the data according to the distance field, leading to greater storage saving [4].

Recently scientists have also instrumented their simulations with particles to capture and better understand the turbulent dynamics in combustion processes [37]. The initial placement of each particle is desired to be close to the flame surface. Through our integration method, we can construct the distance field with respect to the dynamic flame surface during simulations, and allow scientists to fine tune the placement and amount of particles with respect to the different distance thresholds, as shown in Figure 7.

The second combustion data set has a spatial resolution of $1408 \times 1080 \times 1100$. In this case, we used the variables of HO_2 and stoichiometric mixture fraction ($mixfrac$). The main flame structure corresponds to the $mixfrac$ surface for which the isovalue is 0.2. Figure 8 shows the overview of the original simulation data, the resulting distance field, and the distance-based visualization. Our solution can generate a high-resolution distance field of the data set. By interactively changing the distance threshold, we can reveal the distribution of HO_2 around the flame surface with great clarity. These details are occluded by the larger exterior structures (Figure 8 (a)) and cannot be easily visualized using traditional transfer functions, as shown in the comparison between Figure 8 (c) and (d). Our visualization provides scientists a clear exploration of the relationship between the combustion variables and the flame surface for detailed chemical mechanism.

Apart from scientific simulation applications, our method can also naturally support the traditional geometric models used in engineering applications. The third data set is a geometric model used in a numerical study on air flow effects on a passenger car. We build a parallel distance tree and generate a distance field with a spatial resolution of $512 \times 512 \times 512$. The researchers have used the CFD approach to obtain the aerodynamic data and complex flow structure around the car. Although the researchers are equipped with advanced visualization and analytic tools to capture important flow features, distance fields provide the researchers another dimension for investigation. For example, the researcher are interested in the effects of different material types applied on the front hood of the car to effectively control the distribution of air-temperature. As shown in Figure 9, we can clearly see the distribution of high temperature close to the car front (engine), as well as the attenuation of temperature with respect to the different distances from the car. This distance-based

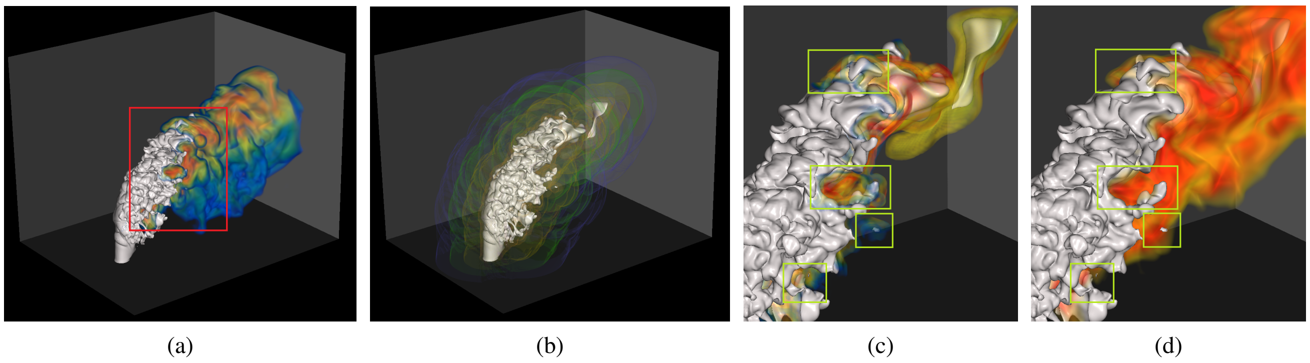


Fig. 8. (a) shows the simultaneous rendering of two variables of the second combustion simulation data set. The $mixfrac$ isosurface is white. The HO_2 variable is volume rendered, where higher values are indicated by orange or red, and lower values by green or blue. (b) shows the volume rendering of the distance field, where the white $mixfrac$ surface is surrounded by a set of isosurfaces corresponding to the different distance values. (c) shows a distance-based rendering of the highlighted region in (a). With distance control, we can clearly see the distribution of HO_2 around the surface of the flame, which was occluded by the larger exterior structures in (a). This result cannot be easily obtained using traditional value-based transfer functions. (d) shows a rendering where we make the region of lower HO_2 (the green and blue region in (a)) translucent to make the interior $mixfrac$ surface visible. However, the result cannot clearly reveal the relationship between higher HO_2 and the surface. It has also lost the distribution information of lower HO_2. The highlighted regions in (c) and (d) provide the examples of comparison.

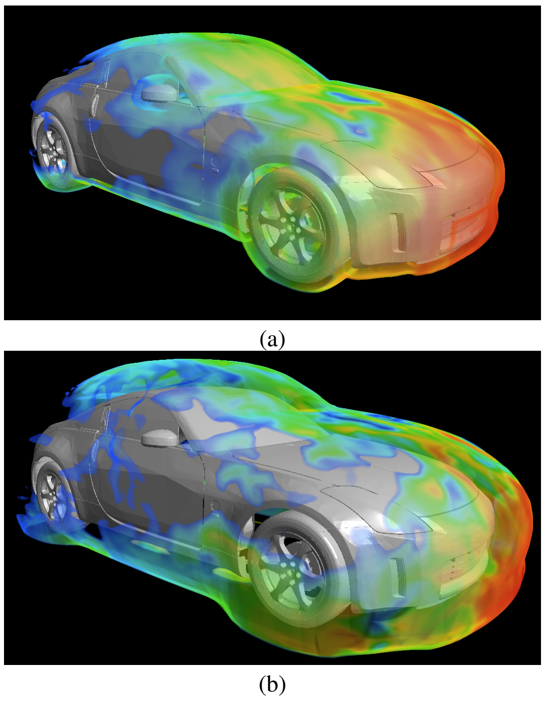


Fig. 9. Volume rendering of the temperature field at different distances from the car (in particular, the car engine).

visualization can facilitate the researchers' investigation on material types with various air flow conditions.

We also compute the exact Euclidean distance between points and polygons for the Boeing 777 model. After building the parallel distance tree, we generate a volume of distance field with a resolution of $2048 \times 1942 \times 624$. The constructed distance field features smooth isocontours and preserves fine details of individual components. We can apply different operations using the distance field, for example, the morphological operation of dilation shown in Figure 10. Note that the left images in the second and third rows of Figure 10 are generated using the distance field rather than the original model. Because the distance tree is refined adaptively around the vicinity of objects, our method can effectively capture fine structural details in such a large geometric model, which can be potentially useful in distance-related applications, such as collision detection in maintenance and assembly task simulations [38].

5.2 Performance Evaluation

We tested our parallel distance field method on two supercomputers. The first one is *Hopper*, a Cray XE6 supercomputer at the Lawrence Berkeley National Laboratory. The system contains 6384 nodes interconnected by the Cray Gemini Network. Each node has two 2.1 GHz twelve-core CPU with 32 GB of RAM. The second one is *Intrepid*, an IBM Blue Gene/P supercomputer at the Argonne National Laboratory. The system contains 40960 nodes interconnected by the InfiniBand Ethernet. Each node has one 850 MHz quad-core CPU with 2 GB of RAM. For clarity, we only present the performance results using the two combustion data sets and the Boeing 777 data set in Table 1. Table 2 lists the major time components contributing to the overall cost.

In the first test, we conduct in-situ distance field construction using the first combustion data set with the following three configurations:

- Config. 1: computing the Euclidean distances between points and isosuface polygons without OpenMP.
- Config. 2: computing the Euclidean distances between points and isosuface polygons with OpenMP.

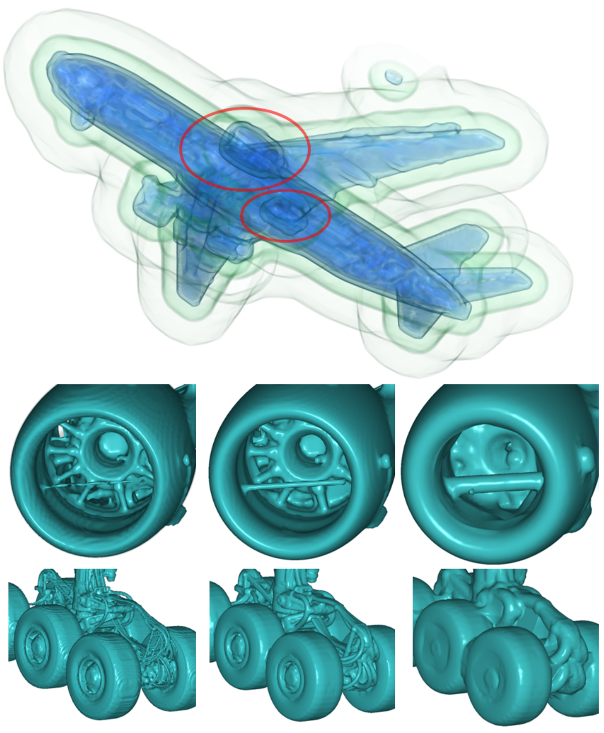


Fig. 10. Isosurface rendering of the distance field generated from the Boeing 777 model. The top image shows the overviews of the distance field. The second and third rows provide the close-up views of two highlighted components of the model, where we use isosurface rendering to depict the distance field, and the isovalues increase from left to right to mimic the effect of dilation. Our method can capture the fine structural details, for example, a long and very thin wire across the engine.

- Config. 3: computing the Euclidean distances between points and isosuface voxels without OpenMP.

We use the T isosurface in Figure 6 to construct the distance fields. Table 3 shows the detailed performance results on Hopper [3]. In Configuration 1 (the top green section of Table 3), we achieve the ideal speedup for the distance tree initialization (T_dist_i) with a parallel efficiency of 103.9% from 4320 to 34560 CPU cores. However, we also note that the parallel efficiency is about 67.6% from 8640 to 34560 CPU cores. This is because the initial global tree construction (T_g_tree) is not scalable in our current design, which becomes the performance bottleneck with a large number of processors. However, we note that this is one-time initialization and the cost can be amortized for a large number of time steps. Without considering T_g_tree, for T_dist_i', the parallel efficiency is 82.2% from 8640 to 34560 CPU cores, and 128.1% from 4320 to 34560 CPU cores. On the other hand, as we pointed out in Section 4.8, the cost to update the tree can be significantly lower than the initial tree construction cost. We can clearly see that T_u_tree is much smaller than the sum of T_g_tree and T_l_tree. Furthermore, we compute the Euclidean distances based on polygons in this configuration. As discussed in Section 4.8, we do not reuse the results from the previous time step for a precision purpose, and thus there are no savings for updating the distance field (T_u_dist). The overall parallel efficiency of tree and distance field updating (T_dist_u) is 132.7% from 4320 to 34560 CPU cores.

In Configuration 2 (the middle orange section of Table 3),

Time	Operation
T_sim	Simulation
T_g_tree	Build initial global distance tree
T_l_tree	Build full-grown local distance tree
T_l_dist	Compute local vertex distances
T_c_dist	Compute remote vertex distances
T_exg	Exchange vertices and vertex distances
T_vol	Construct sampled distance volume
T_u_tree	Update local distance tree
T_u_exg	Exchange vertices and vertex distances for updating
T_u_dist	Update local and remote vertex distances
T_dist_i	Initialize: $T\_g\_tree + T\_l\_tree + T\_l\_dist + T\_c\_dist + T\_exg + T\_vol$
T_dist_i'	Initialize: $T\_l\_tree + T\_l\_dist + T\_c\_dist + T\_exg + T\_vol$
T_dist_u	Update: $T\_u\_tree + T\_u\_exg + T\_u\_dist$

TABLE 2
Major execution time components and their corresponding operations.

PEs	4320	8640	17280	34560	69120
T_sim	123.64	62.72	27.21	14.62	24.39
Config.1: Euclidean distances between points and polygons (w/o OpenMP)					
$\boldsymbol{T\_dist\_i}$	60.62(100%)	19.73(153.5%)	10.30(147.1%)	7.29(103.9%)	
$\boldsymbol{T\_dist\_i'}$	60.46(100%)	19.39(156.0%)	9.28(163.0%)	5.90(128.1%)	
T_g_tree	0.16	0.36	1.03	1.39	
T_l_tree	1.76	0.96	0.73	0.38	
T_l_dist	11.27	4.04	1.93	1.13	
T_c_dist	47.42	14.37	6.61	4.38	
T_exg	0.01	0.01	0.01	0.01	
$\boldsymbol{T\_dist\_u}$	58.83(100%)	18.47(159.2%)	8.59(171.3%)	5.54(132.7%)	
T_u_tree	0.13	0.06	0.04	0.02	
T_u_exg	0.01	0.01	0.01	0.01	
T_u_dist	58.69	18.40	8.54	5.51	
Config.2: Euclidean distances between points and polygons (with OpenMP)					
$\boldsymbol{T\_dist\_i}$	20.12(100%)	7.25(138.7%)	4.41(114.2%)	3.53(71.3%)	
$\boldsymbol{T\_dist\_i'}$	19.96(100%)	6.89(144.8%)	3.38(147.7%)	2.14(116.8%)	
T_g_tree	0.16	0.36	1.03	1.39	
T_l_tree	1.76	0.96	0.73	0.38	
T_l_dist	4.16	1.33	0.56	0.37	
T_c_dist	14.03	4.59	2.08	1.37	
T_exg	0.01	0.01	0.01	0.01	
$\boldsymbol{T\_dist\_u}$	18.33(100%)	5.99(153.0%)	2.69(170.4%)	1.78(128.9%)	
T_u_tree	0.13	0.06	0.04	0.02	
T_u_exg	0.01	0.01	0.01	0.01	
T_u_dist	18.19	5.91	2.64	1.74	
Config.3: Euclidean distances between points and voxels (w/o OpenMP)					
$\boldsymbol{T\_dist\_i}$	0.67(100%)	0.79(42.5%)	0.69(24.2%)	1.30(6.4%)	1.45(2.9%)
$\boldsymbol{T\_dist\_i'}$	0.59(100%)	0.43(68.5%)	0.35(42.7%)	0.11(68.6%)	0.09(43.6%)
T_g_tree	0.08	0.35	0.34	1.19	1.37
T_l_tree	0.12	0.11	0.08	0.03	0.03
T_l_dist	0.14	0.08	0.07	0.02	0.01
T_c_dist	0.33	0.24	0.19	0.05	0.03
T_exg	0.01	0.01	0.01	0.01	0.01
$\boldsymbol{T\_dist\_u}$	0.076(100%)	0.030(126.7%)	0.031(61.7%)	0.022(43.8%)	0.018(26.4%)
T_u_tree	0.065	0.020	0.018	0.010	0.008
T_u_exg	0.006	0.007	0.011	0.011	0.009
T_u_dist	0.005	0.003	0.002	0.001	0.001

TABLE 3
Timing breakdown for in-situ distance field construction with the combustion simulation on Hopper. The time is measured in seconds per time step. The percentage numbers represent the parallel efficiency where the times measured with 4320 CPU cores are used as the references.

3. Not all data points of the time components have been obtained in our tests due to the limit of our supercomputer time allocation.

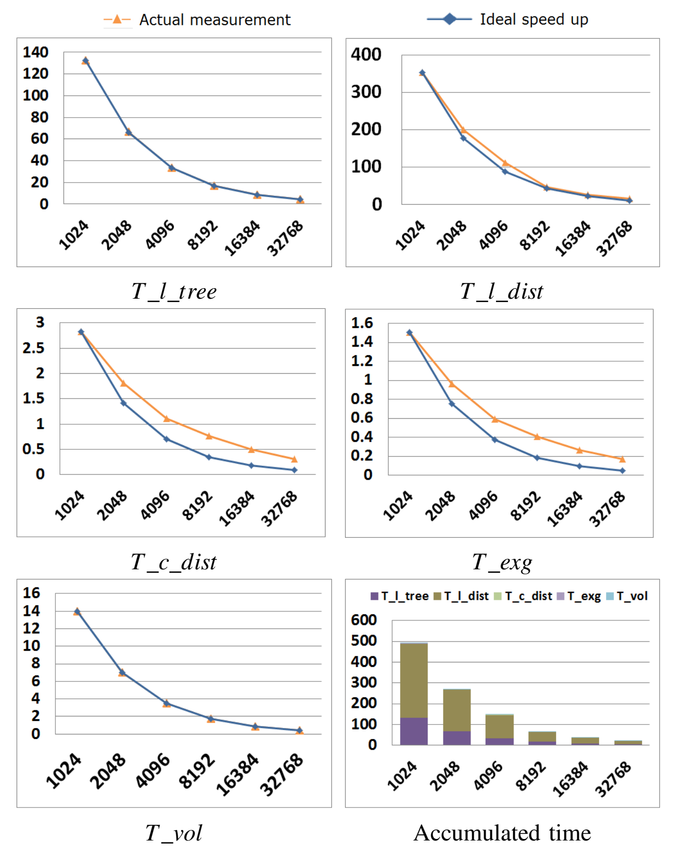


Fig. 11. Scalability study using the second combustion data set for post-processing on Intrepid. In each plot, the horizontal axis represents the number of processors, and the vertical axis represents the running time in seconds.

we use OpenMP with four threads per MPI process to accelerate distance computation. The average speedups of T_l_dist and T_c_dist from OpenMP are about 305% and 322%, respectively, compared with the times in Configuration 1. Averagely, T_dist_i is 17% of T_sim, which is reduced from 42% in Configuration 1. T_dist_u is around 11% of T_sim, showing that we can compute the exact Euclidean distances at a low cost during a simulation run.

In Configuration 3 (the bottom pink section of Table 3), we compute the Euclidean distances based on isosurface voxels. The corresponding computation cost is much lower than the one based on polygons. The sum of T_l_dist and T_c_dist is averagely 1% of its counterpart in Configuration 1. Because of the lower computation cost, the overhead of T_g_tree clearly dominates T_dist_i with a large number of processors. However, in this configuration we can easily track the updated voxels overtime, and only update the portion of distance fields related to these voxels. Therefore, we can significantly lower both T_u_tree and T_u_dist. As shown in Table 3, we can see that T_dist_u is only about 1.2% of T_dist_i with 69120 CPU cores, and is negligible to the simulation time T_sim.

Figure 11 shows the detailed performance results using the second combustion data to construct a distance field on Intrepid. Figure 8 (a) and (b) show the overview of the original data and the resulting distance field. Our method achieves

PEs	1024	2048	4096	8192
T_dist_i	4083.79(100%)	2327.11(87.74%)	1259.33(81.07%)	678.26(75.26%)
T_dist_i'	4083.74(100%)	2327.02(87.75%)	1259.06(81.09%)	677.78(75.31%)
T_g_tree	0.05	0.09	0.27	0.48
T_l_tree	426.93	293.62	168.08	101.84
T_l_dist	2812.12	1548.52	803.37	405.44
T_c_dist	836.55	477.48	281.10	165.72
T_exg	8.14	7.39	6.51	4.78

TABLE 4
Timing breakdown for computing the distance field of the Boeing 777 model on Hopper. The time is measured in seconds. The percentage numbers represent the parallel efficiency where the times measured with 1024 CPU cores are used as the references.

86.9% average parallel efficiency from 1024 to 32768 CPU cores, without considering T_g_tree. We can see that the most expensive operations, the local tree construction and the local vertex distances computing, are still close to the ideal speedup. Compared with the first test, an interesting observation is that the computation times are significantly higher in the second test. This might be because Intrepid has a slower CPU speed compared with Hopper. Nevertheless, our method achieves desired performance on both architectures.

Table 4 shows the detailed performance of distance field construction around the Boeing 777 model on Hopper. The overall parallel efficiency is 75.26% from 1024 to 8192 CPU cores. To the best of our knowledge, we present for the first time the scalable distance field construction of a geometric data at this large scale.

5.3 Discussion

The performance study shows several advantages of our method. First, we can clearly see that the overall parallel distance field construction scales very well with the increasing number of processors. The effectiveness of our distance tree is clearly illustrated in the two rather challenging combustion cases where the set Γ of elements is only distributed among a small set of the processors. Moreover, although T_g_tree increases with the number of processors, it is only a one-time cost. The subsequent cost for updating tree is much smaller by leveraging the temporal coherence of simulation data. Such a low cost makes it feasible to compute distance field during the simulation run, and thus significantly reduce time to solution by avoiding expensive data movement.

Second, our method has a low communication cost even with a large number of processors. This is first because our communication schedule is constructed by leveraging the spatial coherence of simulation data, and each processor only needs to exchange data with a small set of neighboring processors that are identified efficiently with our parallel distance tree. In addition, we only send the vertex key that is a 4-byte integer, rather than three floating point coordinates. Thus, our method only incurs a small communication footprint relative to the size of the entire distance field volume. For example, in the second test, the exchanged data is under 0.01% of the total data on 32768 CPU cores.

Third, the experimental results show that our method does not depend on any particular architectures. The linear data representation can facilitate the acceleration of distance computation using different techniques, such as GPU and OpenMP. Furthermore, our examples highlight the flexibility of our method with respect to different data types and applications in real-world. Given our novel design of parallel distance tree, we obtain clear improvement over the previous state-of-the-art, and believe that the design principle is applicable to a wider class of scientific and engineering applications.

However, a native octree may not provide effective I/O once we store a large tree on persist storage. This is because an octree is characterized as a narrow/deep tree such that visiting a leaf node may require a long traversal path from the root and incur intensive I/O operations. In the future, we will study approaches to store out-of-core octree for large distance fields. In addition, we have not implemented a direct visualization of parallel distance tree yet. Thus, although exact distance values at any point can be computed from a tree, we have to interpolate the tree to generate a volume of distance field for visualization. We plan to leverage our parallel octree-based volume rendering method [39] to enhance visualization.

6 Conclusion

We present a highly scalable parallel distance field construction algorithm that is critical for data analysis and visualization of large-scale applications. Our method employs inexpensive load balancing at the global tree construction stage and efficiently parcels out workloads among the processors to minimize the communication traffic. It features a highly scalable scheme of data partition and representation compatible to large systems. It also has been directly integrated with simulation codes to minimize the data transformation overhead. Moreover, it allows clear and interactive observation of fine details in large scale data. We believe that our algorithm sets a record with regard to highly scalable performance of real-world large data on distributed architectures for distance field construction.

Researchers have exploited massive GPUs to accelerate simulations [40]. However, the disparity between I/O speed and compute speed is further aggravated, and communication and load balancing remain the fundamental challenges for scalability. Our ultimate goal is a fast, scalable distance tree that can handle large scale data with different underlying representations using hundreds of thousands of CPU cores or GPU accelerators. Our approach provides a foundation for big data management and is key to bridge the gap between simulations and data analytics in the exascale computing era.

Acknowledgments

This research has been sponsored in part by the National Science Foundation through grants IIS-1320229, CCF-1025269, and IIS-1423487, the Department of Energy through grants DE-FC02-06ER25777 and DE-FC02-12ER26072 with program managers Lucy Nowell and Ceren Susut-Bennett, and the ExaCT Center for Exascale Simulation of Combustion in Turbulence.

References

[1] A. Gyulassy, M. Duchaineau, V. Natarajan, V. Pascucci, E. Bringa, A. Higginbotham, and B. Hamann, "Topologically clean distance fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1432–1439, Nov. 2007.

[2] D. Laney, M. Bertram, M. Duchaineau, and N. Max, "Multiresolution distance volumes for progressive surface compression," in *3D Data Processing Visualization and Transmission, 2002. Proceedings. First International Symposium on*, 2002, pp. 470–479.

[3] M. B. Nielsen, O. Nilsson, A. Söderström, and K. Museth, "Out-of-core and compressed level set methods," *ACM Trans. Graph.*, vol. 26, no. 4, Oct. 2007.

[4] C. Wang, H. Yu, and K.-L. Ma, "Application-driven compression for visualizing large-scale time-varying data," *IEEE Computer Graphics and Applications*, vol. 30, pp. 59–69, 2010.

[5] R. Fabbri, L. D. F. Costa, J. C. Torelli, and O. M. Bruno, "2D Euclidean distance transform algorithms: A comparative survey," *ACM Computing Surveys*, vol. 40, no. 1, pp. 2:1–2:44, 2008.

[6] M. W. Jones, J. A. Bæentzen, and M. Sramek, "3D distance fields: A survey of techniques and applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, pp. 581–599, 2006.

[7] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proceedings of the 9th international conference on High performance computing for computational science*, ser. VECPAR'10, 2011, pp. 1–25.

[8] H. Yamada, "Complete Euclidean distance transformation by parallel operation," in *Proceedings of Int. Conf. Patt. Recogn.*, 1984, pp. 69–71.

[9] L. Chen and H. Y. Chuang, "An efficient algorithm for complete Euclidean distance transform on mesh-connected SIMD," *Parallel Computing*, vol. 21, no. 5, pp. 841–852, 1995.

[10] A. Datta and S. Soundaralakshmi, "Fast parallel algorithm for distance transform," *IEEE Transactions On Systems, Man, And Cybernetics-Part A: Systems And Humans*, vol. 33, no. 5, pp. 429–434, 2003.

[11] L. Chen, Y. Pan, and X. hua Xu, "Scalable and efficient parallel algorithms for Euclidean distance transform on the LARPBS model," *IEEE Transactions On Parallel And Distributed Systems*, vol. 15, no. 11, pp. 975–981, 2004.

[12] A. Sud, N. Govindaraju, R. Gayle, and D. Manocha, "Interactive 3D distance field computation using linear factorization," in *Proceedings of I3D*, 2006, pp. 117–124.

[13] N. Cuntz and A. Kolb, "Fast hierarchical 3D distance transforms on the GPU," in *Proceedings of EUROGRAPHICS*, 2007, pp. 93–96.

[14] W.-K. Jeong and R. Whitaker, "A fast iterative method for eikonal equations," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2512–2534, 2008.

[15] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel, "Parallel algorithms for approximation of distance maps on parametric surfaces," *ACM Transactions on Graphics*, vol. 27, no. 4, pp. 104:1–104:16, 2008.

[16] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, 2011.

[17] P. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[18] O. Bruno and L. Costa, "A parallel implementation of exact Euclidean distance transform based on exact dilations," *Microprocessors and Microsystems*, vol. 28, no. 3, pp. 107–113, 2004.

[19] Y.-H. Lee, S.-J. Horng, and J. Seitzer, "Parallel computation of the Euclidean distance transform on a three-dimensional image array," *IEEE Transactions On Parallel And Distributed Systems*, vol. 14, no. 3, pp. 203–212, 2003.

[20] J. C. Torelli, R. Fabbri, G. Travieso, and O. M. Bruno, "A high performance 3-D exact Euclidean distance transform algorithm for distributed computing," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 24, no. 6, pp. 1–19, 2010.

[21] H. Samet, "Distance transform for images represented by quadtrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 4, no. 3, pp. 298–303, 1982.

[22] J. Strain, "Fast tree-based redistancing for level set computations," *Journal of Computational Physics*, vol. 152, no. 2, pp. 664–686, 1999.

[23] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones, "Adaptively sampled distance fields: a general representation of shape for computer graphics," in *Proceedings of SIGGRAPH'00*, 2000, pp. 249–254.

[24] C. Lauterbach, Q. Mo, and D. Manocha, "gProximity: Hierarchical gpu-based operations for collision and distance queries," *Computer Graphics Forum*, vol. 29, no. 2, pp. 419–428, 2010.
[25] A. Krishnamurthy, S. McMains, and K. Haller, "GPU-Accelerated minimum distance and clearance queries," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 6, pp. 729–742, 2011.
[26] Y.-J. Kim, Y.-T. Oh, S.-H. Yoon, M.-S. Kim, and G. Elber, "Coons BVH for freeform geometric models," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 169:1–169:8, 2011.
[27] K. Zhou, M. Gong, X. Huang, and B. Guo, "Data-parallel octrees for surface reconstruction," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 5, pp. 669–681, 2011.
[28] S. Aluru, G. M. Prabhu, and J. Gustafson, "Truly distribution-independent algorithms for the n-body problem," in *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, ser. SC '94, 1994, pp. 420–428.
[29] L. Ying, G. Biros, D. Zorin, and H. Langston, "A new parallel kernel-independent fast multipole method," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03, 2003, pp. 14–29.
[30] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron, "From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06, 2006.
[31] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 45–57, 2010.
[32] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "ISABELA-QA: Query-driven analytics with ISABELA-compressed extreme-scale scientific data," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 31:1–31:11.
[33] U. Neumann, "Communication costs for parallel volume-rendering algorithms," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 49–58, 1994.
[34] I. Gargantini, "An effective way to represent quadtrees," *Commun. ACM*, vol. 25, no. 12, pp. 905–910, 1982.
[35] D. J. Abel and J. L. Smith, "A data structure and algorithm based on a linear key for a rectangle retrieval problem," *Computer Vision, Graphics, And Image Processing*, vol. 24, pp. 1–13, 1983.
[36] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation," *ACM Trans. Graph.*, vol. 11, no. 3, pp. 201–227, 1992.
[37] J. Wei, H. Yu, R. W. Grout, J. Chen, and K.-L. Ma, "Visual analysis of particle behaviors to understand combustion simulations," *IEEE Computer Graphics and Applications*, vol. 32, no. 1, pp. 22–33, 2012.
[38] T. C. Johnson and J. M. Vance, "The use of the voxmap pointshell method of collision detection in virtual assembly methods planning," in *Proceedings of ASME 2001 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2001.
[39] H. Yu, K.-L. Ma, and J. Welling, "A parallel visualization pipeline for terascale earthquake simulations," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ser. SC '04, 2004.
[40] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 15:1–15:11.

Hongfeng Yu is an assistant professor in Computer Science and Engineering at the University of Nebraska-Lincoln. His research interests include scientific visualization, high-performance computing, and user interfaces and interaction. He received the BE and ME degrees in computer science from Zhejiang University, China, and the PhD degree in computer science from the University of California, Davis. After his PhD graduation, he spent four years as a postdoctoral researcher at Sandia National Laboratories, California. Contact him at yu@cse.unl.edu.

Jinrong Xie is currently a Ph.D candidate in the Visualization and Interface Design Innovation research group (ViDi) in the Department of Computer Science at the University of California, Davis. His research topics are related to large scale scientific visualization and high performance parallel computing. Before he jointed the ViDi group, Jinrong obtained a Bachelor degree in Computer Science from Shanghai Jiaotong University in 2007 followed by a Master degree in Computer Science from Zhejiang University in 2010. Contact him via email: jrxie@ucdavis.edu.

Kwan-Liu Ma , an IEEE Fellow, is a professor of computer science and the chair of the Graduate Group in Computer Science (GGCS) at the University of California, Davis. He leads the VIDi research group and directs the UC Davis Center for Visualization. Professor Ma received his PhD degree in computer science from the University of Utah in 1993. His research interests include visualization, high-performance computing, and user interface design. Professor Ma was a recipient of the NSF PECASE award in 2000, and the IEEE VGTC 2013 Visualization Technical Achievement Award. He was a paper chair of the IEEE Visualization Conference in 2008 and 2009, and an associate editor of IEEE TVCG (2007-2011). He is a founder of PacificVis, Ultravis, and LDAV. Professor Ma presently serves on the editorial boards of the IEEE CG&A, the Journal of Computational Science and Discoveries, and the Journal of Visualization. Contact him via email: ma@cs.ucdavis.edu.

Hemanth Kolla obtained a B.Tech in Aerospace engineering from IIT Madras in 2003 followed by an M.Engg. in Aerospace from IISc Bangalore in 2005. From 2005 to 2006 he worked as a design engineer for the combustor design team of GE aviation at the John F. Welch Technology Center (JFWTC), Bangalore. During his stint at GE, he was drawn towards the field of turbulent combustion and computational modelling for predictive simulations. He pursued a doctorate at the University of Cambridge, working on the topic of turbulent premixed combustion modelling. He obtained his PhD in 2010 for a thesis titled Scalar dissipation rate based flamelet modelling of turbulent premixed flames. Since 2010 he has been working as a post-doctoral associate at the Combustion Research Facility, Sandia National Laboratories, performing massively parallel direct numerical simulations of turbulent combustion on some of the fastest super-computers in the world (TITAN at OakRidge Leadership Computing Facility and HOPPER at National Energy Research Scientific Computing Center). Contact him at hnkolla@sandia.gov.

Jacqueline H. Chen is a Distinguished Member of Technical Staff at the Combustion Research Facility at Sandia National Laboratories. She has contributed broadly to research in petascale direct numerical simulations (DNS) of turbulent combustion focusing on fundamental turbulence-chemistry interactions. These benchmark simulations provide fundamental insight into combustion processes and are used by the combustion modeling community to develop and test turbulent combustion models for engineering CFD simulations. Working closely with SciDAC VACET, SDM, and Ultrascale Visualization Institute, and NCCS/ORNL, she and her team have also developed methodology for automated combustion workflow, in-situ topological feature tracking and visualization of petascale simulated combustion data, and developing DNS software for hybrid architectures. She received the DOE INCITE Award in 2005, 2007, 2008-2010 and 2011 and the Asian American Engineer of the Year Award in 2009. She is a member of the DOE Advanced Scientific Computing Research Advisory Committee (ASCAC) and Subcommittee on Exascale Computing. She was the co-editor of the Proceedings of the Combustion Institute, volumes 29 and 30 and a member of the Executive Committee of the Board of Directors of the Combustion Institute. Contact her at jhchen@sandia.gov.