Scalable Parallel Tracking of Time-Varying 3D Flow Features

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

Authors Name/s per 2nd Affiliation (Author)
line 1 (of Affiliation): dept. name of organization
line 2: name of organization, acronyms acceptable
line 3: City, Country
line 4: Email: name@xyz.com

***Abstract*—Large-scale time-varying volumetric data set can take terabytes to petabytes of storage space to store and process. One promising approach is to process the data in parallel in situ or batch, extract features of interest and analyze only those features, to reduce required memory space by several orders of magnitude for the following visualization tasks. However, extracting volumetric features in parallel is a non-trivial task as features might span over multiple processors, and local partial features can be only visible within their own processor. In this paper, we discuss how to generate and maintain connectivity information of features across different processors. Based on this connectivity information, partial features can be integrated, which makes possible for the large-scale feature extraction and tracking in parallel. We demonstrate the effectiveness and scalability of different approaches on two data sets, and discuss the pros and cons of each approach.**

***Keywords*-feature tracking; parallel graph;**

I. Introduction

The accessibility to supercomputers with increasing computing power has enabled scientists to simulate physical phenomena of unprecedented complexity and resolution. However, these large-scale time-varying data sets can take tera- or even peta-bytes of space to preserve. One promising solution to the problem is to reduce the data by storing only features of interest in the data. Storing the extracted features instead usually requires storage space that is several orders of magnitude smaller than the raw data. However, large volumetric data sets are typically presented and processed in a distributed fashion, simply because of the shear size, which makes extracting and tracking features embedded in such distributed volumetric data set a non-trivial task.

Existing researches on feature-based data visualization have been done mostly focusing on decomposing features using quantitative measures such as size, location, shape or topology information, and so on. These measures cannot be applied to distributed volumetric data directly since volumetric features are consist of certain amount of voxels, and are very likely to span over distributed data blocks as they evolve over time. Therefore existing quantitative measures of partial data scattered across different data blocks cannot be used to describe an integrated feature, unless the distribution of partial features can be obtained beforehand.

To obtain the distribution information of features, a connectivity graph of each feature should be generated and maintained. In this paper, we present an approach to creating and maintaining feature residual information as well as connectivity information using parallel graphs. Comparing to the existing approaches, which generate and maintain the global feature information in a single host node, our approach can be done locally that only involves residual data blocks of target features. This requires least communication overhead and avoids the potential link contention. We demonstrate the effectiveness of this method with two vortical flow data sets and the scalability of our system in a distributed environment.

II. Background and Related Work

A. Feature Extraction and Tracking

Feature extraction and tracking are two closely related problems in feature-based visualization. Although many feature tracking algorithms have been introduced, most of them extract features from individual time steps and then try to associate them between consecutive time steps. Silver and Wang [1] defined threshold connected components as their features, and tracked overlapped features between successive time steps by calculating their differences. Octree was employed in their method to speed up the performance and the criteria they used were domain dependent. Reinders et al.

[2] introduced a prediction verification tracking technique that calculates a prediction by linear extrapolation based on the previous feature path, and a candidate will be added to the path if it corresponds to that prediction. Ji and Shen [3] introduced a method to track local features from time-varying data by using higher-dimensional iso-surfacing. They also used a global optimization correspondence algorithm to improve the robustness of feature tracking [4]. Caban et al. [5] estimated a tracking window and then tried to find the best match between that window and different sub-volumes of subsequence frames by comparing their distance of textural properties. Also, Bremer et al. [6] described two topological feature tracking methods, one employs Jacobi sets to track critical points while the other uses distance measures on graphs to track channel structures.

Most of the aforementioned methods extract features from each time step independently and then apply correspondence calculations. This could become very slow when the data size becomes large. Muelder and Ma [7] proposed a prediction-correction approach that first predicts feature region based on the centroid location of that in previous time steps, and then corrects the predicted region by adjusting the surface boundaries via region growing and shrinking. This approach is appealing because of its computing efficiency and the reliability in an interactive system.

B. Parallel Feature Extraction and Tracking

To boost the speed for feature tracking in data-distributed applications, Chen and Silver [8] introduced a two stage partial-merge strategy using master-slave paradigm. The slaves first exchange local connectivity information using Binary-tree merge, and then a visualization host collects and correlates the local information to generate the global connectivity. This approach is not scalable since half of the processors will become idle after each merge. It is also unclear how the visualization host can efficiently collect local connectivity information from non-server processors, since gathering operation is typically very expensive given a large number of processors.

The approach proposed in this paper follows a different strategy. Instead of being sent back to a host, the local connective information is computed and preserved only in the nodes where correspondent features reside. Hence there is no global connectivity information preserved in the host. The host only serves as an interface to broadcast the criterion of feature of interest to the other nodes. In this way, the computation of merging local connectivity information is distributed to the slaves and thus effectively reduces the potential communication bottleneck on the host.

Moreover, our approach does not need a global synchronization for gathering all local connectivity information at the host, hence avoids potentially long computation time caused by features spanning over a large number of nodes. In addition, there are no needs to set a barrier to wait for all connectivity information being sent back to the host and thus if there exists features that span over a large number of nodes but was not selected by the user, the potentially long computation time for these features will not block the whole process. This makes it ideal for an interactive system, where users can select the features of interest and instantly receive the visual feedback as the features evolves.

C. Parallel Graph Algorithm and Applications

Graph-based algorithms have long been studied and used for a wide range of applications, typically along the line of divide-and-conquer approaches. Grundmann et al. [9] recently proposed a hierarchical graph-based approach for video segmentation, a closely related research topic to 3D flow feature extraction as video can be treated as a space-time volume of image data [10]. In their work, a connected sequence of time-axis-aligned subsets of cubic image volumes are assigned to a set of corresponding processors, and incident regions are merged if they are inside the volumes window. Incident regions on window boundary, however, are first marked as ambiguous and later connected by merging neighboring window to a larger window, which consists of the unresolved regions form both window on their common boundary. This approach is not applicable for memory intensive situation since the allocated volume size before merging might already reaches the capacity. Same restriction applies to the parallelized BK algorithm [11] proposed by Liu and Sun [12], in which data are uniformly partitioned and then adaptively merged to achieve fast graph-cuts. These approaches are suitable for shared-memory but not message-passing parallelization as for their frequent shifting on data ranges.

III. System Overview

Our implementation uses a similar system structurer that forms the basis of previous work [7]. Figure 1 depicts a high-level overview of the work flow for a single time-step.

First, the source volume data is read either from a pre-generated data set or from a simulation program in-situ. Then, user can identify features of interest via transfer function manipulation as such candidate features will be extracted. However, since the descriptive power of a transfer function is often insufficient to precisely highlight the desired subset of features, user may have to select significant features either by simple point-and-select or iteratively filter out unwanted ones using different feature descriptors. Once the user has identified the interesting features, the system tracks only these significant features in subsequent time steps.

What our approach distinct from that of the previous one lies in that the global connectivity information of each feature need to be obtained before the Significant Feature Selection process could be carried out. As shown in blue color in Figure 1, first the raw data are distributed in logically adjacent data blocks.. After Feature Extraction is done in each data block, the set of local features as well as the local connectivity information will be generated. Then by applying our merging algorithms, global connectivity information can be obtained efficiently such that each data block will possess the global connectivity information for each of its residing features. This enables Signification Feature Selection in a distributed environment and henceforth Feature Tracking and Visualization.

IV. Methodology

The biggest challenge for tracking large time-varying volumetric data set lies in that, though features can be extracted within individual processors using the conventional

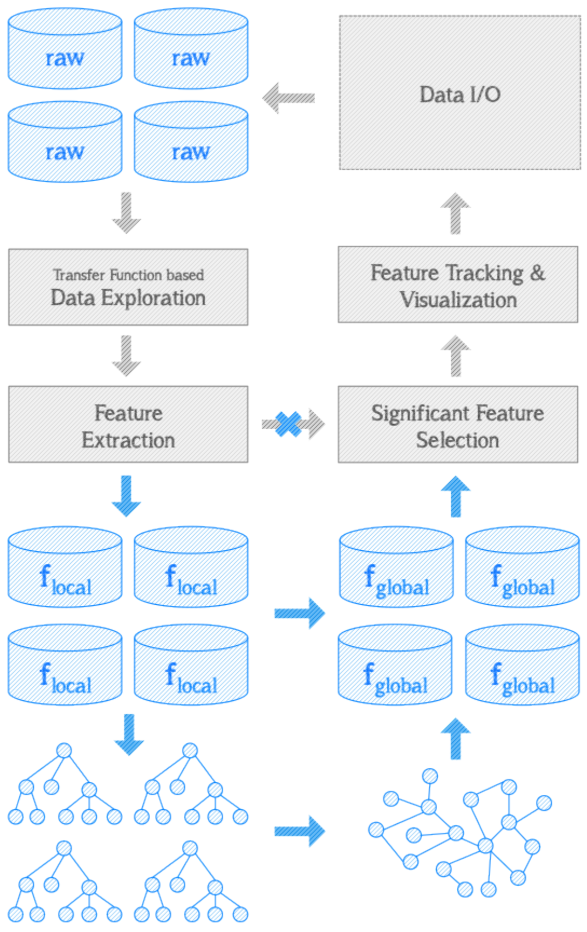

Figure 1. Work flow of a single time-step

methods, they might also span over multiple data blocks, which is unavoidable as the number of processors increases. To extract and trace a feature over a distributed volume data set, we need to build and maintain the connectivity information of the features across multiple nodes. As feature descriptors, such as size, curvature, and velocity, are distributed among processors, connectivity information can facilitate us to obtain the description of a feature from neighboring processors, and also enable more advanced operations such as similarity evaluation.

However, it typically requires data exchanges among processors to build and maintain such connectivity information, and thus incurs extra communication cost. To design a proper communication scheme for better performance and scalability, we carefully consider the following three factors in our design:

- N_{com} : The amount of communication required to build the connectivity graph;
- $N_{proc/com}$: The number of processors involved in each communication;
- $N_{data/com}$: The amount of data that must be changed in each communication.

In the following sections, we give a detailed description on how to create and maintain such connectivity information using tree structures and then merge them into an undirected unweighted graph to minimize the cost over the above three factors.

A. Feature Extraction

In general, a feature can be any interesting object, structure or pattern that is considered relevant for investigation. Here, a feature is defined as the collection of voxices encompassed by a certain iso-surface. Such volumetric feature could be extracted by conventional techniques such as region growing, geometry or topology based clustering, or other domain specific algorithms. In our work we use a standard region-growing algorithm [13] to partition the original volume data into an initial set of features. This can be done by first spreading a set of seeding points inside the volume, and then clustering voxices into separate regions, each regarded as a single feature. After specific features have been identified from a single time step, we can track their evolution over time using a prediction-correction approach [7], as feature locations should be consistent for consecutive time step provided that the sampling interval is sufficiently small. Once the prediction is made, the actual region can be obtained by adjusting the surface of the predicted region: first shrink the edge surface points to obtain the mutual region between consecutive time steps, and then use of a region-growing method to generate the refined region, as depicted in Figure 2. This prediction-correction approach was proved effective and efficient for tracking feature on a single processor [7]. However, when the size of the volume becomes too large to be able to fit into a single processor, a cluster of processors is often needed so as to process the data in parallel.

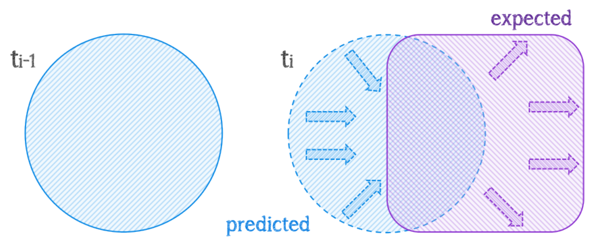

Figure 2. The predict-correct feature tracking approach. First the predicted region is derived using location(s) of that feature in previous time step t_{i-1}. Then we shrink the portion that does not satisfy the criteria in t_i, which obtain us the mutual portion of that in t_{i-1} and t_i. Finally we apply region growing again to obtain the expected feature region depicted.

One challenge to parallelize the tracking approach lies in that it can be difficult to obtain the global feature descriptions unless they can be shared and merged in an efficient way. This is because the features may span over multiple processors and partial features in different data blocks are operated independently. An intuitive way of exchanging such feature information is to first find how features span over multiple processors, and then merge them according to the connectivity information.

B. Creating Local Connectivity Tree

If we partition a volumetric data set into a regular grid of data blocks, it is very likely that some of the features will cross multiple blocks. Leveraging that the cross-section of such feature in both side of adjacent blocks should match, we could connect separate parts of a feature in adjacent blocks by comparing their cross-sections on correspondent boundary surfaces.

Since data is distributed, each processor is not aware of the partial features identified on the other processors. Though exchanging the voxels on the sectional area between two adjacent processors would be sufficient for finding possible matches of partial features, we choose to exchange more abstract data to find matches:

- $P_{centroid}$: The geometric centroid of the cross-sectional area.
- $P_{min-max}$: The minimal and maximal (min-max) coordinate of the cross-sectional area;

The reason why we choose geometric centroid instead of introducing a voxel-width ”ghost surface” is that it requires much less communication cost. A ghost surface that stores boundary surface belonging to neighboring blocks might help to achieve voxel-wise matching for partial features. However, maintaining such ghost surfaces requires frequent inter-process communication and is considerably expensive for data generated in real-time. Consider the fact that it is rarely the case that a feature will have a sharp half-dome-like sectional plane right on the boundary surface, we can loose the matching criterion by allowing a 1-voxel offset between two correspondent geometric centroids. Together with the min-max coordinate of the cross-sectional area, bipartite matching of partial features could be achieved, as shown in Algorithm 1.

Algorithm 1 Match of two partial features

```
if $P_{centroid} = P'_{centroid}$ and $P_{min-max} = P'_{min-max}$ then
    return $f$ matches $f'$
end if
```

Another reason the min-max coordinate values are not optional is because they ensure correct connectivity for some special cases where one cross-sectional area is surrounded by another concave or hollow area whose centroid points happen to be the same, as depicted in Figure 3.

Based on the afore-explained fact that partial features could be connected by finding matches, we can abstract the local connectivity information using a tree structure as depicted in Figure 4. For each data block in the grid, there will be six direct neighbors (the outermost blocks have less), each with a sharing boundary surface with the current block. The connectivity tree is constructed taking the current block as root, its six adjacent blocks as its first level child nodes, and a new leaf is appended to each surface node if a local feature touches the block boundary surface.

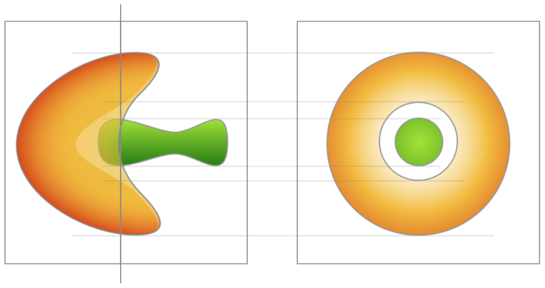
Figure 3. A special case where two features share the same centroid on the section.

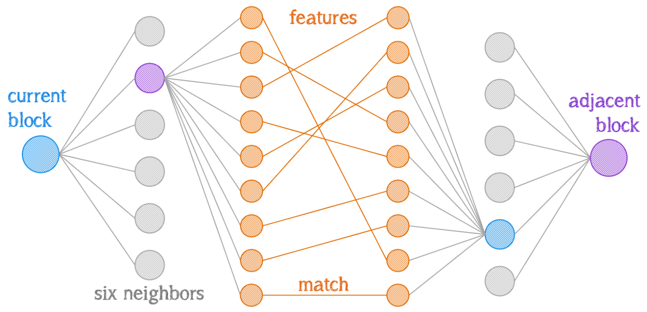

Figure 4. The tree structure used for maintaining local connectivity information, where the root node is encoded with the current block index, its child nodes the index of its neighboring blocks, and for each child node its leaves represent local on-boundary-features. The leaves should match that reside in the corespondent neighboring block, leveraging which a connectivity graph can be constructed.

Note that each voxel in the local data block has a unique global index, each leaf could thus be encoded using 3 integers (global index for $P_{centroid}$, P_{min} and P_{max}), we use the index of $P_{centroid}$ as the feature index, and sort the sibling leaves according to its index in ascending order. On the other hand, the root and the first level child nodes can be encoded with the index of corresponding data blocks, which is irrelevant to the number of feature-on-boundary (henceforth referred as N_{fb}). Therefore the overall spatial complexity of creating local connectivity tree for each data block is $(\theta(3 * N_{fb}))$. Consider an extreme case that there are 10k features touching each side of the boundary surface, the total memory needed for the local connectivity tree is only $((3*10000+1)*6+1)*4bytes \approx 720KB$, neglectable comparing to the volume size itself.

From the temporal complexity perspective of view, the creation of local connectivity tree will not introduce extra computational cost as it can be done along with the region growing process. The value of $P_{centroid}$ and $P_{min-max}$ are updated only if the feature reaches the boundary surface, hence the temporal complexity for creating the local connectivity tree remains $O(\sqrt[2/3]{N_{vexol}})$, a magnitude less than $O(extraction)$.

Algorithm 2 shows the detailed algorithm of creating local connectivity tree in the region growing process.

Algorithm 2 Creating Local Connectivity Tree

```
if $t = t_0$ then
    $seeds \leftarrow randomVortices()$
    for each $seed$ in $seeds$ do
        $feature \leftarrow expendRegion()$
        append $feature$ to $featureList$
    end for
else
    for each $feature$ in $featureList$ do
        $feature \leftarrow predictRegion()$
        $feature, P_{centroid}, P_{min-max} \leftarrow$ adjustRegion()
        $leaf \leftarrow LEAF(P_{centroid}, P_{min-max})$
        append $leaf$ to $connectivityTree$
    end for
end if

adjustRegion:
if Voxel $v$ on boundary surface then
    $P_{centroid} \leftarrow updateBoundaryCentroid()$
    $P_{min-max} \leftarrow updateMinMaxBoundary()$
end if
```

C. Creating Global Connectivity Graph

After local connectivity trees have been created within each data block, their leaves need to be exchanged and merged to obtain the overall description of a partitioned feature. The exchanging and merging process is decisive that their effectiveness will largely affect the overall performance and scalability of the feature tracking algorithm as a whole.

In this subsection, we start with a naive solution and discuss progressively through different data exchanging strategy for different scenarios.

1) The Naive Solution: A naive solution to obtaining global connectivity information for each feature-on-boundary is to exchange its corresponding leaf with its targeting block. Recall that the first level child nodes are encoded as the ranks of adjacent blocks, and the leaves the global index of the geometric centroid and the min-max boundary on the shared boundary surface. Therefore if a leaf received from neighboring data block matches a local leaf, the two leaves, that is, the two partial features they represented, must share the same boundary section with the same centroid and section region. In other word, these two partial features are resulted by partitioning a original feature, and should be considered as the same feature.

To prevent a leaf from being resending to the same neighboring block multiple times, it is marked as sent and unless the same feature was found partially reside in another neighboring block, this leaf will be ignored in the next communication to reduce $N_{data/com}$.

Algorithm 3 shows the detailed process to merge matched leaves.

Algorithm 3 Merging Matched Leaves

```
traverse local connectivity tree in preorder
for each $f_{local}$ in $localLeaves$ do
    if $f_{local}$ is sent = false then
        send $f_{local}$ to targeting block
    end if
    mark $f_{local}$ as sent
end for
for each $f_{recv}$ in $recievedLeaves$ do
    for each $f_{local}$ in $localLeaves$ do
        if $f_{recv}$ matches $f_{local}$ then
            $f_{local}.id = min(f_{recv}.id, f_{local}.id)$
            mark $f_{local}$ as not sent
        end if
    end for
end for
```

The naive solution may work for data sets with feature-on-boundary that span only a few number of blocks. However, if there exists long curly features partitioned evenly over the data block grid, each block needs to consecutively communicate with its neighbors to incrementally gather and merge a global feature. This requires O(N_p) communications to connect a single feature, where N_p is the number of processors in the grid. Consequently, the total communication cost is O($N_{fb} \times N_p$) times communication. In addition, since one processor cannot predict how many leaves it will receive from its adjacent processors, it is hard to schedule the communication process.

2) The Centralized Approach: One possible solution to improve the aforementioned serialized approach is to introduce master-slave hierarchy to reduce the number of communication required for merging all leaves. The master-slave hierarchy can be constructed using a separate host processor. When the feature extraction process is done, all local connectivity trees are gathered to the host processor. Then the host nodes starts to merge the leaves from each connectivity tree to construct a single global connectivity graph.

The merit of this centralized approach lies in that it requires inter-processor communication only once, ($N_{com} = 1$). Moreover, the global graph of feature information can be preserved in the host that it can response to feature queries directly without collecting information from the slaves again. However, this approach has an obvious drawback. Since all local connectivity trees are sent to the host, the number of processors involved in each communication is $N_{proc/com} = N_p$, and there exists potential bottlenecks, both in communication and computation, on the host.

3) The Decentralized Approach: A better solution is to decentralize the gathering and merging process from a single host processor to all processors available. After the feature extraction process is done and so does the creation of local connectivity tree, an *all-gather* process starts to exchange all local connectivity trees within each data block to all the others. Each block will first collect a full copy of all local connectivity trees followed by the same process as mentioned in Algorithm 3 to merge the leaves into a single concise connectivity graph.

Though the ”redundant” host processor is no longer required when applying for computation resources, this approach does not actually resolve the bottleneck problem since now every processor is acting like the host for that they still need to gather all local trees and to merge them simultaneously. For real world data set however, it is rarely the case that all features will span over every data block. In other word, it is unnecessary for one processor to gather leaves of features that are not local. To reduce the redundant communications with every other data block in the grid, the decentralized approach could be further improved to communicate with only those data blocks that are directly adjacent. That is, each data block only communicates with its direct neighbors and shares information with them regardless what is happening outside.

For a regularly partitioned volumetric data set, there are at most six direct neighbors for each data block. Instead of gathering connectivity information from all non-current data blocks in the grid, we let each data block gather only new leaves from its neighboring blocks within each communication. This could be considered as a higher level of region growing process, starts from one seeding block and grows to adjacent blocks by exchanging and merging connectivity information in a bread-first fashion, until all cross-boundary features are connected. Figure 5 gives an example of the process of processor-level region growing in a 2D processor grid.

The reason we choose the six-direct-neighbor paradigm is because it gives the minimum overall computational complexity. It takes a maximum of $3n - 1$ times communication, where n denotes the longest side length, for any data block to receive connectivity information from the furthest block on the opposite diagonal. The temporal complexity for garnering all necessary leaves is hence as low as $O(\sqrt[3]{N_{proc}})$. And the number of processors involved in each communication is a constant of a maximum of six ($N_{proc/com} \leq 6$).

Another optional paradigm is to let each processor communicate with its 26 neighbors, including the adjacent

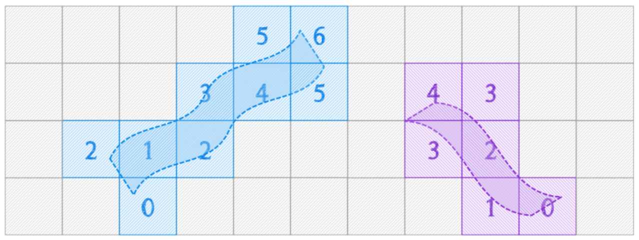

Figure 5. The processor-level region growing process in a 2D grid, in which it takes a maximum of 2n-1 times (3n-1 for 3D grid) for the outermost processor to grow to the furthest processor

diagonal blocks. Communicate with the adjacent diagonal block takes as much as half the time for any block to reach its furthest diagonal. However, $N_{proc/com}$ is also increased to 26. For data sets that features span over all blocks are subordinate, six-direct-neighbor paradigm outweighs the 26-neighbors paradigm in computational complexity.

To schedule the communication with neighboring blocks, we use two schedule flags, *toSend* and *toRecv*, to indicate whether a block need to send or receive leaves to/from its neighboring blocks. Whenever a block receives leaves that can be merged into its existing connectivity graph, we assume the neighboring blocks might receive further connectivity information from their neighbors. In this case, the *toRecv* flag is set to true until the connectivity graph is saturated; On the other hand, the *toSend* flag is set in accordance to whether one of the neighboring blocks need to receive leaves, which can be obtained by gathering the neighboring *toRecv* flags. Each data block will keep exchanging connectivity information until both *toSend* and *toRecv* flags are set to false.

The detailed scheduling algorithm is depicted in Algorithm 4.

4) The hybrid approach: Consider the fact that as data evolves over time, the volumetric features may drift but should not change drastically in neither size, shape nor location if the sampling interval when generating the data is sufficiently small. Base on this assumption, we can further optimizing the aforementioned decentralized-local-merge approach. Utilizing the prediction-correction approach for single processor feature tracking we can further reduces the communication cost required to complete the whole connectivity graph.

As depicted in as depicted in Figure 6, for every time step t_i, when the global connectivity graph is obtained, the local communicators will be updated for the next time step, t_{i+1}, with the union of processors that share the same set of feature-on-boundary with the current processor. The leaves from these data blocks are necessary to complete the global connectivity graph no matter which gathering approach is used. Hence, for these must-involve data blocks, we apply the all-gather-decentralize approach, allowing the minimum

Algorithm 4 Decentralized Local Merge

```
$toSend, toRecv \leftarrow true$
$\delta \leftarrow localLeaves$
while $toSend$ is true or $toRecv$ is true do
  $target \leftarrow myRank$ if $toRecv = true$
  $procsToSync \leftarrow gatherNeighbor(target)$
  for each $proc$ in $procsToSync$ do
    if $toSend = true$ then
      send $\delta$ to $proc$
    end if
    if $toRecv = true$ then
      receive $\delta\prime$ from $proc$
    end if
  end for
  $toSend \leftarrow false$ if $procsToSync$ is empty else $true$
  $toRecv \leftarrow false$ if $\delta = Merge(\delta, \delta\prime)$ else $true$
  $\delta \leftarrow Merge(\delta, \delta\prime)$ if $toRecv$ is true
end while
```

Algorithm 5 Prediction-enabled Local Merge

```
Let $P_{f_i} \equiv$ all processors that contains feature $f_i$
if $t = t_0$ then
  $localCom \leftarrow union(P_{f_i})$ for all $f_i$ in current data block
else
  for each $proc$ on boundary of $localCom$ do do
    $P'_{f_i} \leftarrow decentralizedLocalMerge(proc)$
  end for
  $localCom \leftarrow union(P'_{f_i})$
end if
```

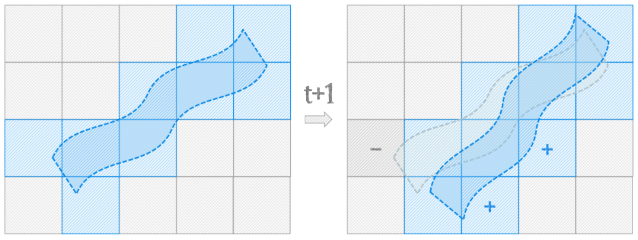

Figure 6. Utilizing the prediction-correction for fast connectivity information synchronization. In each time step the

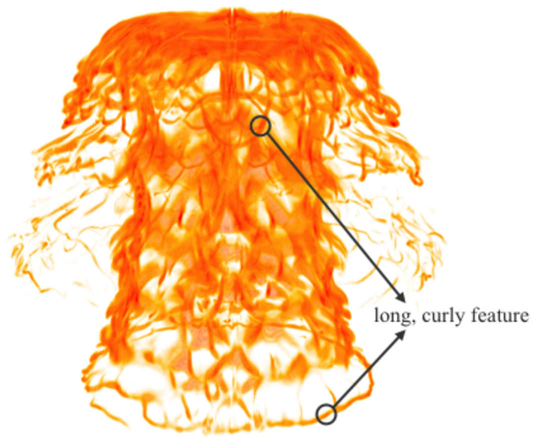

Figure 7. The rendering result of a single time step

one-time synchronization to finish gathering all leaves that are necessary for updating the connectivity graph based on the graph created at the previous time step t. Then, processor-level region growing, a.k.a the processor-level-decentralize approach is applied to extend the boundary of data blocks, obtaining newly connected blocks caused by the evolution of the volume data. Again, only those leaves that are changed and not yet sent will be exchanged. This ensure that we minimize the amount of data being sent over network.

The detail algorithm of the hybrid approach is given in Algorithm 5.

V. Result and Analyze

We first test our feature extraction and tracking algorithm on a 256*256*256 vortex data set obtained from a combustion solver that can simulate turbulent flames. In this data set, each voxel represents the magnitude of vorticity derived from velocity using a curl operator. As time evolves, vortical features may vary from small amassed blob features to long curly features that span over large portion of the volume, as depicted in Figure 7.

A. *Performance Result*

The volume data could be generated either in advance or on the fly, hence here we ignore the I/O cost and only focus on the computation time of the following three portions.

1. Extracting Features ($T_{extract}$);

Since we use the region-growing based algorithm to extract features, the computation time is mainly determined by the scale of the volume as well as the number of processors being used. Once the raw volume data and how it is partitioned, a.k.a. the size of each data block is determined, the computation time for extracting residing features remains approximately the same. For pre-generated data set, the size of each data block decreases as the number of processors increases and hence so does the time spent on extracting features. As depicted in Figure 8 $T_{extract}$ is approximately log-linear decreased as the number of processors grows from 8 to 16384;

2. Create Local Connectivity Graph (T_{create});

Despite the size of each data block, the computational cost for creating and updating local connectivity graph is dependent on the number of the features extracted within the original volume, or more precisely, the number of features that touches the boundary surface of their residing

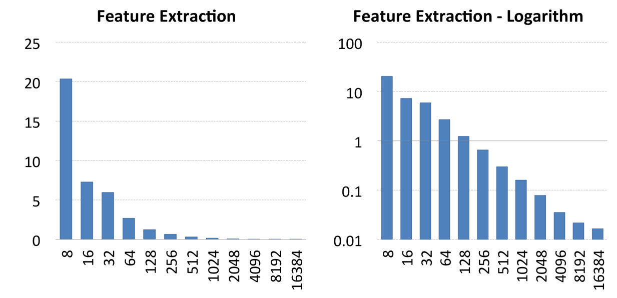

Figure 8. Computation time for feature extraction, log-linearly scalable as the number of processor increases

data block. Similar to $T_{extract}$, T_{create} will decrease as the number of processors increases for pre-generated data set, as the the number of feature-on-boundary decreases accordingly. For the combustion data set, it takes an average of 0.1 seconds to create the local connectivity graph, approximately 0.5% the time of $T_{extract}$ using the same amount of processors. This portion increases but does not succeed 1% in out test, hence T_{create} is not considered a bottleneck for the pre-generated data set. for in-situ visualization however...

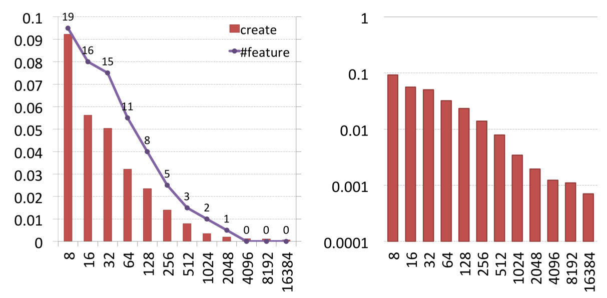

Figure 9. Computation time for creating local connectivity tree, log-linearly scalable as the number of processor increases and the time is approximately proportional to the number of features-on-boundary

3. Merge Local Connectivity Graph (T_{merge})

The three (only the first two are tested for now) merging strategies are the major factors of the scalability of our algorithm. Though the number of feature-on-boundary decreases as more processors involved, the communication time for the Global-Merge approach increases as N_p increases. The total temporal cost T_{global} for Global-Merge exceeds $T_{extract}$ after certain amount of processors, 2048 for the combustion data set, which makes the overall computation time rebounds (Figure 10). The Local-Merge approach on the other hand, scaled well up to 16384 processors for the combustion data set, as the communicational cost is as low as $O(\sqrt[3]{N_p})$. (Figure **??**).

Also from the comparison of the computational cost as depicted in Figure 11, we can see that the Global-Merge is suitable for scenarios that we only need a small cluster of processors, and Local-Merge for creating connectivity graph using a relatively large amount of the processors.

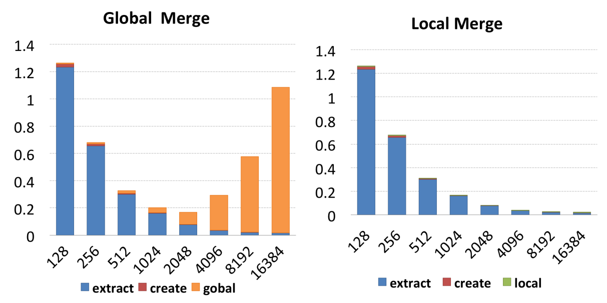

Figure 10. Total computation time comparison for different merging strategy. The decentralized-global-merge strategy scaled up to 2048 processors but the merging time outweighs the extraction time when using more processors; The decentralized-local-merge strategy scaled log-linearly up 16384 processors for the combustion data set.

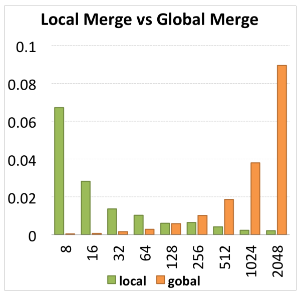

Figure 11. The comparison between the computation time for the local and global merging strategy. The global merge strategy works well for a small number of processors while the local merge exceeds after a certain number, as for the combustion data set, 128 processors, is used.

VI. Conclusion

To be revised In this paper, we presented a decentralized approach that all feature connectivity information are created and preserved among distributed processors. Traditional approaches perform connectivity test on each processor and subsequently correspond them in a host processor after gathering all or partially merged connectivity information. Our approach does not follow this paradigm. Rather, instead being sent back to the host, the local connectivity information are computed and preserved only in the local processor. There is no copy of the global feature information preserved in the host, and the host only acts as the interface from where the criterion of feature of interest is broadcast. In this way, the computation of merging local connectivity information is distributed to the slaves, which can effectively remove the potential communication bottleneck on the host processor. Moreover, there's no need to set a barrier and wait for all connectivity information being sent back to the host, thus if one of the features spans over a large number of processors but was not selected by the user, the potentially long computation time for this feature will not be considered. This makes it ideal for an interactive system, where users can select the feature of interest and instantly receive the visual feedback as the feature evolves.

References

[1] D. Silver, "Tracking and visualizing turbulent 3d features," *Visualization and Computer Graphics*, vol. 3, no. 2, pp. 129–141, 1997.

[2] K. Reinders, "Feature-based visualization of time-dependent data," Ph.D. dissertation, Delft University of Technology, 2001.

[3] G. Ji, H.-W. Shen, and R. Wenger, "Volume tracking using higher dimensional isosurfacing," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, ser. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 28–. [Online]. Available: http://dx.doi.org/10.1109/VISUAL.2003.1250374

[4] G. Ji and H. Shen, "Feature tracking using earth mover's distance and global optimization," *Proceedings of Pacific Graphics 2006*, 2006.

[5] J. Caban, A. Joshi, and P. Rheingans, "Texture-based feature tracking for effective time-varying data visualization," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 6, pp. 1472 –1479, nov.-dec. 2007.

[6] P.-T. Bremer, E. M. Bringa, M. A. Duchaineau, A. G. Gyulassy, D. Laney, A. Mascarenhas, and V. Pascucci, "Topological feature extraction and tracking," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012007, 2007.

[7] C. Muelder and K.-L. Ma, "Interactive feature extraction and tracking by utilizing region coherency," in *Visualization Symposium, 2009. PacificVis '09. IEEE Pacific*, april 2009, pp. 17 –24.

[8] J. Chen, D. Silver, and M. Parashar, "Real time feature extraction and tracking in a computational steering environment," *Proceedings of the 11th high ...*, 2003.

[9] M. Grundmann, V. Kwatra, M. Han, and I. Essa, "Efficient hierarchical graph-based video segmentation," *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2141–2148, Jun. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5539893

[10] A. W. Klein, P. pike J. Sloan, A. Finkelstein, and M. F. Cohen, "Stylized video cubes," in *In ACM SIGGRAPH Symposium on Computer Animation*, 2002, pp. 15–22.

[11] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 9, pp. 1124 –1137, sept. 2004.

[12] J. Liu, "Parallel graph-cuts by adaptive bottom-up merging," *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2181–2188, Jun. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5539898

[13] R. Huang and K.-L. Ma, "Rgvis: region growing based techniques for volume visualization," in *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*, oct. 2003, pp. 355 – 363.