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

Rewrite the following two paragraph: motivation, challenge, technique approach, tested on...

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 map (map? 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 three 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.

Shall I compare our work here in the related work section? 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.Not enough, need to read more.

III. System Overview

Our implementation uses a similar system structurer that forms the basis of previous work [7]. Figure 1 depicts a high-level view and the work flow of a single time-step. The source volume data can be read either from a pre-generated data set or from a simulation program in-situ.

set that contains one flow attribute (e.g. vorticity). By explor- ing a single time step via transfer function manipulation, the user can identify features of interest. Because the descriptive power of a trans- fer function is often insufficient to precisely focus on a desired subset of features, we provide two mechanisms by which the user can refine the selected set. The first is a mouse-based picking tool which selects the feature under the cursor during a user click. In this context under the cursor is dictated by the 2D projection of each feature from the

current viewpoint. As such, this approach is most effective when the desired features are easy to see. Yet when the transfer function yields a large feature set, clutter would make feature-by-feature picking te- dious. To address this situation, our system provides the ability to filter the extracted features using several similarity measures. Each of these tests, described in detail in Section 4, provide a numerical answer to the question, How similar are these two features?. The careful appli- cation of these similarity measures can effectively reduce an initially cluttered group to a more useful collection. Once the user has identified the interesting features, our system tracks only these features in subsequent time steps. Many existing feature tracking techniques extract all features in each time step be- fore determining feature correspondence using some sort of similarity test. Such techniques are computationally expensive and fail to utilize inter-step correlation information. The aforementioned

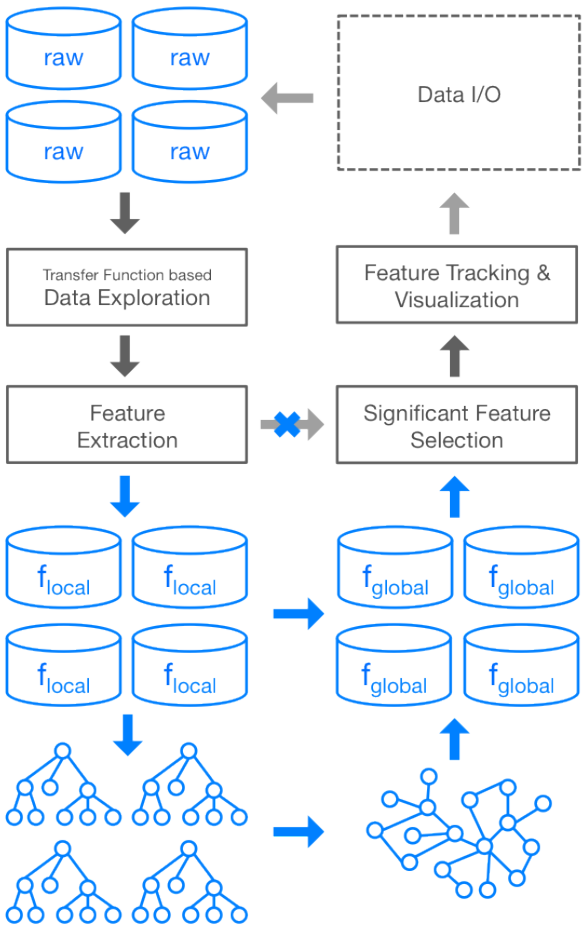


Figure 1. Work flow of a single time-step

technique by Muelder, et al. [19] does utilize correlation to improve efficiency, but it does not consider new similar features emerging over time. Our sys- tem combines these two methods. We use both inter-step correlation information and leverage feature similarity testing to track the users selection. Because of the latter ingredient, we are able to not only track directly selected features, but also to identify and track emergent features which are similar to the users initial set.

IV. Methodology

Though features can be extracted within individual processors using the conventional 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 the 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:

- The amount of communication required to build the connectivity graph;
- The number of processors involved in one communication;
- The amount of data that mush be changed in each communication step.

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

A. Feature Extraction

Feature extraction is a process that first detects a feature, and then calculates quantitative attributes characterizing the feature so what?. In general, a feature can be any interesting pattern, structure, or object that is considered relevant for investigation. In our application, a feature is defined as the collection of voxices encompassed by a certain iso-surface. Such a 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. Usually, scientists prefer to focus on features that are meaningful as dictated by their underlying research studies. Potential features detected by the region-growing algorithm, however, are often cluttered and contain unexpected noise. Therefore, an iterative refinement process is conventionally applied to improve the results after the potential feature set being generated.

Tangelder and Veltkamp [14] presented the applicability and performance of context-based 3D shape descriptors and corresponding retrieval methods. Their work suggested that when designing an interactive refinement process, we need to carefully select a set of feature descriptors, which should be deformation robust, such as size, location, or skeletal curvature of features. In addition, we also need to consider computational efficient in our design, which TODO On the other hand, we note that the refinement process based on these feature descriptors can typically reduce the number of features, which also can lower computational cost of tracking.

After specific features have been identified from a single time step, we can track their evolution over time using a prediction-correction approach. The feature locations in a future time step can be predicted from the location of features in the previous time steps. Once 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.

This prediction-correction approach was proved effective and efficient for tracking feature on a single processor [7]. However, when the size of the volume data becomes too large to be fit into one processor, a cluster of multiple processors are often needed to process the data in parallel. 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 volumetric features may span over multiple processors and partial features on different nodes 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 Graph

If we partition a volumetric data set into blocks in form of a regular processor grid, it is very likely that some of features may cross multiple processors and blocks. We note that the cross-section of such a feature on the boundary of two adjacent blocks should match. Leveraging this property, we could connect separate parts of a feature on the adjacent data blocks by comparing their sections on the correspondent boundary surface.

Since data is distributed, each processor is not aware of the partial features identified on the other processors. Exchanging the voxels on the sectional area between two adjacent processors would be sufficient for finding possible matches of partial features between these two processors. However, we choose to exchange more abstract data, that is the minimal and maximal (min-max) coordinate values and the geometric centroid of the cross-sectional area, to reduce the amount of data exchanged over network. The min-max coordinate values are not optional 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 2.

A voxel-width ”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

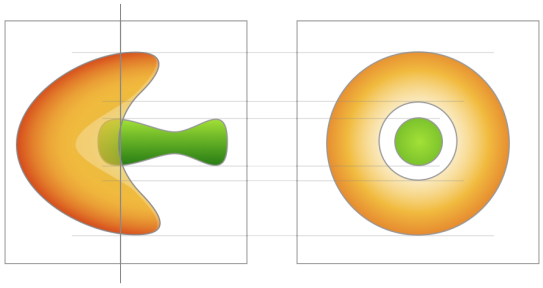


Figure 2. A special case where two features share the same centroid on the section.

in real-time. Instead, we use a simplified approach which requires much less communication cost. That is, if the spatial distance between two centroid points of two neighboring partial features is less than two voxels, the two features are considered a match and belong to the same feature.

The creation of local connectivity graph will not introduce extra computation cost as it can be done along with the direction of region growing. A new edge is appended to the existing local connectivity graph when a local feature touches the block boundary. In the graph, the new edge connects one existing block with the new detected block of a feature. The ending vertices of an edge is denoted using the block indices. Thus the ID of the new detected block is assigned as one ending vertex of the new edge. In addition, the global index of the centroid point is assigned as the edge value.

Algorithm 1 shows the detailed algorithm of creating local connectivity graph.

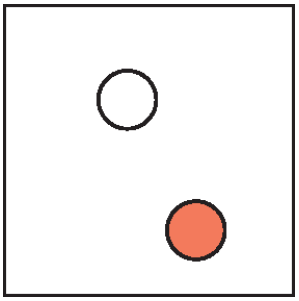


Figure 3. Give a figure to depict the creating of graph process

// TODO (put to result, performance analyze) Memory cost, each feature on boundary will use two INTs, 1 as global centroids coordinate, the other as target node number. even if there's 1000 features on the boundary, it would cost more than 1000 * 5 * 4 = 20mb to store this graph, neglectable compare to the volume data itself. (But kind of large if used for communication)

C. Merging Local Connectivity Graph

1) The naive solution: Local connectivity graphs need to be merged to obtain the overall description of a partitioned

Algorithm 1 Creating Partial Connectivity Graph

$edgeList \leftarrow newList()$
$featureList \leftarrow newList()$
if Current time step $t = t_0$ **then**
  // Initialize random seeding points
  $seedList \leftarrow randomVortices()$
  for each $seed$ in $seedList$ **do**
    $feature \leftarrow expendRegion()$
    append $feature$ to $featureList$
  end for
else
  for each $feature$ in $featureList$ **do**
    $feature \leftarrow predictRegion()$
    $feature \leftarrow adjustRegion()$
    $start \leftarrow$ current processor ID
    $end \leftarrow$ target neighboring processor ID
    $min, max \leftarrow$ min-max boundary coordinate
    $index \leftarrow$ global voxel index of centroid
    Edge $e = Edge(start, end, min, max, index)$
    append e to $edgeList$
  end for
end if

$adjustRegion()$
if Voxel v on boundary surface **then**
  $updateMixMaxBoundary()$
  $updateBoundaryCentroid()$
end if

feature. A naive approach to gathering local connectivity graphs is to let each processor exchanges the shared edges with its neighbors. After exchanges, two edges are merged at a processor if they satisfy the following three conditions:

- The starting and ending vertices are reversely matched;
- The min-max boundary coordinate do match;
- Edge centroid is located within direct neighbors.

Recall that the starting vertex of an edge represents the current processor rank, and the ending vertex represents the rank of a neighboring processor whose partial feature is adjacent to the local one. The edge value represents the global coordinate of the geometric centroid and the min-max boundary on the shared boundary surface. If two edges match to each other, the two partial features must share a 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 the same feature sharing a same feature ID.

Algorithm 2 shows the detailed process to merge matched edges (henceforth referred to as *Reduce*).

This naive solution may work for data sets with small features. However, if there are long curly features partitioned evenly over the processor grid, each processor needs to consecutively communicate with its neighbors to incremently 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($N_f \times N_p$) times communication, where N_f is the number of features within one time step. In addition, since one processor cannot predict how many edges it will receive from its adjacent processors, it is hard to schedule the communication process.

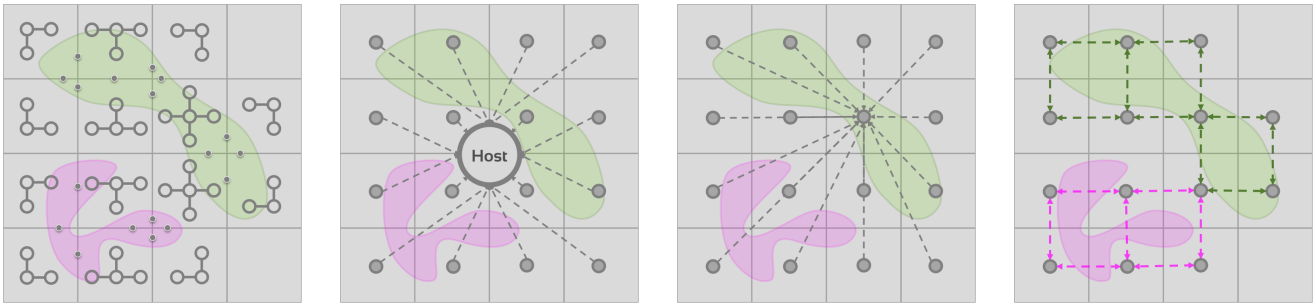
Algorithm 2 Reduce: Merging Matched Edges

Require: $localEdges$, $recievedEdges$
for each ei in $localEdges$ **do**
  for each ej in $recievedEdges$ **do**
    if $ei.start$ = $ej.end$ and $ej.start$ = $ei.end$ **and** $ei.min$ = $ej.min$ and $ei.max$ = $ej.max$ **and** $ei.centroid \approx ej.centroid$ **then**
      if $ei.id < ej.id$ **then**
        $ej.id \leftarrow ei.id$
      else
        $ei.id \leftarrow ej.id$
      end if
    end if
  end for
end for

2) The centralized approach: A possible solution is to use a master-slave hierarchy to reduce the number of communication required for merging all edges. The master-slave hierarchy can be constructed using a separate host processor. When the feature extraction process is done, the edges of a local graph represents the number of features across the block boundary, and also encodes the location information of each local feature. All local graphs are then gathered to the host processor. After this *Gather* operation is done, that is the host processor has collected all local graphs, the *Reduce* operation starts to merge the edges from each partial graph to construct a single global connectivity graph, as depicted in figure 4(b).

The merit of this centralized approach lies in that it requires inter-processor communication only once. Moreover, the global graph of feature information can be preserved in the host, and the host can response feature queries directly without pulling information from the slaves again. However, this approach has an obvious drawback. Since all partial graphs are sent to the host, there exists potential bottlenecks, both in communication and computation, on the host.

3) The decentralized approach: A better solution is to decentralize from *Reduce* on a single host processor to all processors that are available. After the feature extraction process is done and so does the creation of local connectivity graphs, an *Allgather* process starts to exchange all local connectivity graphs within each processor to all the others. Each processor will first collect a full copy of all local



(a) Local connectivity graph (b) Centralized merge with host (c) Centralized merge without host (d) Decentralized merge

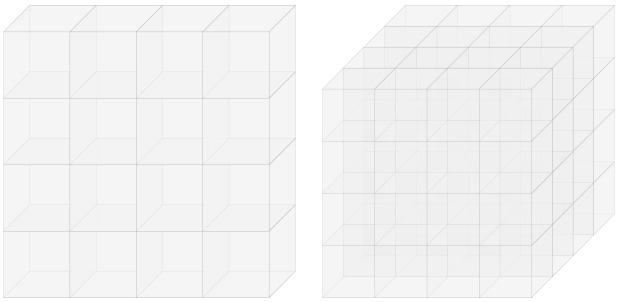
Figure 4. Caption of subfigures (a), (c) and (d)

connectivity graphs followed by the same *Reduce* process to merge the edges into a single concise connectivity graph, as depicted in figure figure 4(c).

Though the "redundant" host processor is no more required when applying for computation resources, this approach does not actually solve the bottleneck problem since now every processor is acting like the host for that they still need to gather all local graphs and try merging them together. For real world data set, however, it is rarely the case that a single feature will span over every processor. In other word, it is unnecessary for a processor to gather edges of features that are not local. To reduce the redundant communications with every other processor in the grid, the decentralized approach could be further improved that we only consider those processors that are directly adjacent to the current one. That is, each processor 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 processor. An other option is to communicate with 26 direct neighbor, but this increases the number of processors being communicated with while not much reduced the number of times to "traverse" the whole processor grid. Instead of gathering connectivity information from all other processors in the grid, each processor only gathers local graphs of their neighboring processors. This could be considered as a higher level of region growing process, which starts from one seeding processor and then grows to its adjacent processors by exchanging and merging connectivity information in a bread-first fashion until all cross-boundary features are connected.

To schedule the local communication with neighboring processors, we use two flags to indicate whether a processor need to send or recieve local edges from/to its neighboring processors. Whenever a processor gathers edges that can be merged into its existing graph, it is considered that it might still need to gather edges, and the *toRecv* flag is set to true until the graph is saturated ¡- need to check; On the other hand, the *toSend* flag is set in accordance to whether one of its neighboring processors need to recieve edges. Each



(a) Region growing for 2d grid (b) Region growing for 2d grid

Figure 5. TODO: Add detail to illustrate the process

processor will keep exchaning edges untill both the *toSend* and *toRecv* flags are set to false. Note that it only takes $3n-1$ times to reach the farest processor on the opposite diagonal for a 3D grid, where n denotes the longest side length, the time complexity for collecting all necessary edges is hence as low as $O(\sqrt[3]{N_p})$.

The detailed scheduling algorithm is depicted in Algorithm 3.

4) The hybrid approach: Consider the fact that as the volume data ? evolves over time, the vortical 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, we can further optimizing the aforementioned decentralized-local-merge approach using a similar prediction-correction approach that further reduces the number of communications required to complete the whole connectivity graph.

For every time step, t_i, when the global connectivity graph is obtained, new local communicators will be updated for the next time step, t_{i+1}, with the union of processors that share a same edge with the current processor, as depicted in Figure 6. However, the edges from these processors are required to complete the global connectivity graph no matter which approach is used. Hence, for these must-involve processors, we apply the all-gather-decentralize approach, allowing the minimum one-time synchronization to finish gathering all

Algorithm 3 Local Merge

Require: $adjacentProcessors$, $localEdges$

$toSend, toRecv \leftarrow true$ // init scheduling flags
$\delta \leftarrow localEdges$ // init data to be sent
while $toSend$ = $true$ or $toRecv$ = $true$ **do**
  $target \leftarrow toRecv$ = $true$? $myRank$: null
  $procsToSync \leftarrow Allgather(target)$
  for each $proc$ in $procsToSync$ **do**
    if $toSend$ = $true$ **then**
      send δ to $proc$
    end if
    if $toRecv$ = $true$ **then**
      receive $\delta\prime$ from $proc$
    end if
  end for
  $toSend \leftarrow procsToSync$ is empty ? $false$: $true$
  $toRecv \leftarrow false$
  $localEdges \leftarrow Reduce(localEdges, \delta\prime)$
end while

$Reduce(localEdges, \delta\prime)$
for each $edge$ in $\delta\prime$ **do**
  if $edge$ is new **then**
    add $edge$ to δ
    $toRecv \leftarrow true$
  end if
end for

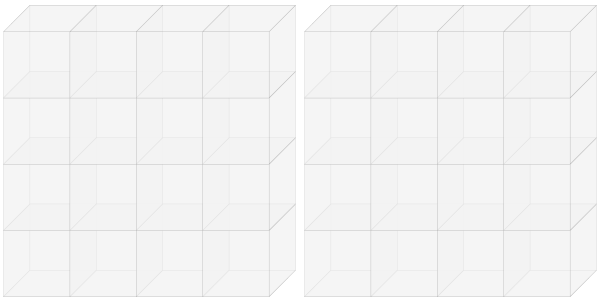


Figure 6. TODO: processor level region growing

edges 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 partial-level-decentralize approach is applied to extend the "boundary" of processes, obtaining newly connected processors caused by the evolution of the original volume. In addition, only those edges that are changed and not synchronized will be sent. This ensure that we minimize the amount of data being sent over network.

The detail algorithm of the hybrid approach is given in Algorithm 4.

Algorithm 4 Prediction-enabled Local Merge

if Current time step $t = t_0$ **then**
  $G_{P_i}^{t_0} \leftarrow localMerge()$ for all processor P_i
  $P_{f_i} \leftarrow$ of processors that contains feature f_i
  $localComm \leftarrow union(P_{f_i})$ for f_i in features in current block
else
  $boundary_block \leftarrow union(P_fi)$ for f_i in features in current block
end if

V. Application

A. Feature Selection and Refinement

To allow a user to select or highlight certain features is commonly required in various applications. Since the feature extraction is performed independently within each processor, we need to design a method to let each processor know if the features on its adjacent processor are selected or not. Intuitively, we can implement this by sending a message to the adjacent PE whenever a feature was detected to touch the surface boundary. But if the target feature spans over multiple PEs, this sending / receiving procedure would take several rounds to end. This is potentially a big problem when the PE/Volume ratio is relatively high such that each feature spans over a lot of PEs. Another problem is that, if a PE has two selected features whose connectivity information arrive in different rounds, it requests to compute twice, which again, will become a problem when PE/Volume ratio is large.

By introducing the global connectivity graph in our approach, whenever part of the feature was selected, the unique feature id will be sent back to the host processor and then be broadcast to all PEs containing it. Thus, the selection can be finished only in one round.

Based on the coordinates user specified or clicked on the volume rendering result, we can identify the user selected processor and feature. Then the host processor imply broadcasts the selected feature id to all those processors who has the partial feature to be highlighted.

B. Feature Tracking

VI. 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.

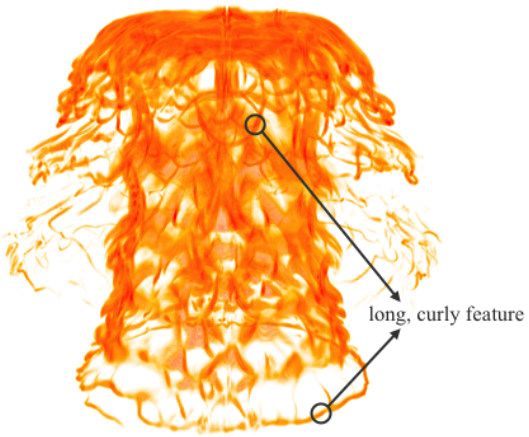


Figure 7. The rendering result of a single time step

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;

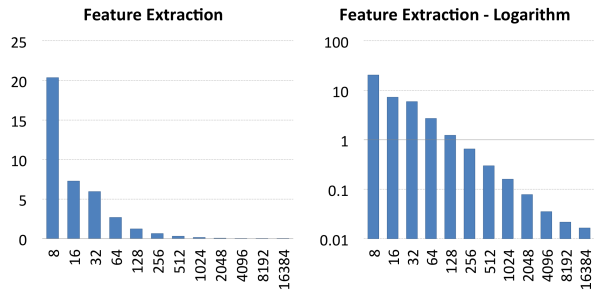


Figure 8. TODO: processor level region growing

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 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, figure? hence T_{create} is not considered a bottleneck for the pre-generated data set. for in-situ visualization however...

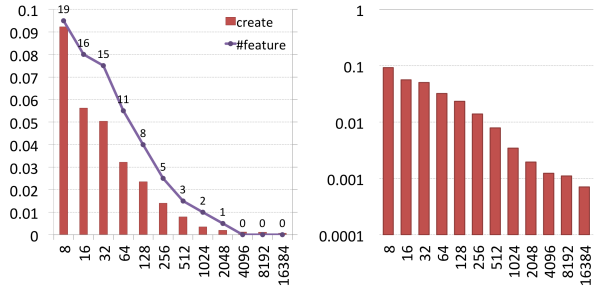


Figure 9. TODO: processor level region growing

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 11).

Also from the comparison of the computational cost as depicted in Figure 12, 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.

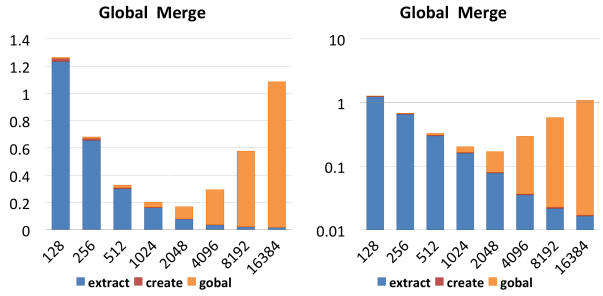


Figure 10. TODO: processor level region growing

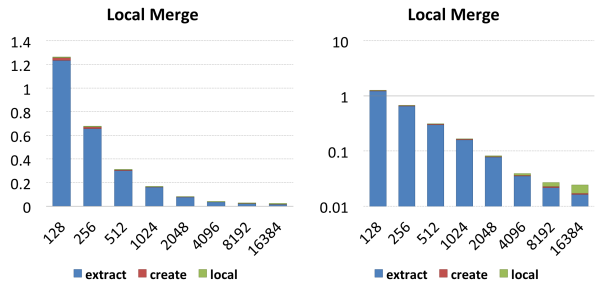


Figure 11. TODO: processor level region growing

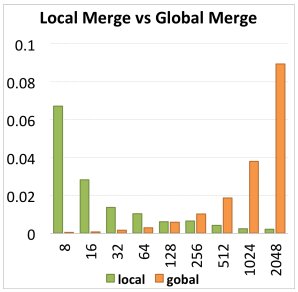


Figure 12. TODO: processor level region growing

B. Visualization Result

VII. Conclusion

To be revised In this paper, we proposed 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.

Acknowledgment

The authors would like to thank... more thanks here

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.

[14] M. Schlemmer, M. Heringer, F. Morr, I. Hotz, M.-H. Bertram, C. Garth, W. Kollmann, B. Hamann, and H. Hagen, "Moment invariants for the analysis of 2d flow fields," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 6, pp. 1743 –1750, nov.-dec. 2007.