LARGE SCALE FEATURE EXTRACTION AND TRACKING

BY PINAKIN DHUME

A thesis submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Master of Science
Graduate Program in Electrical and Computer Engineering

Written under the direction of
Professor Deborah Silver
and approved by

______________________________

______________________________

______________________________

New Brunswick, New Jersey
Oct, 2007
ABSTRACT OF THE THESIS

Large Scale Feature Extraction and Tracking

by Pinakin Dhume

Thesis Director: Professor Deborah Silver

The goal of Scientific Visualization is to provide a more intuitive interpretation of the data being presented. Traditionally, visualization of 3D time-varying datasets is done using animation based on either iso-surfacing or volume rendering. However, for datasets with continuously evolving features it is difficult to follow and see patterns in 3D. An automated procedure to track features and to detect particular events in their evolution can help scientists to concentrate on regions and phenomena of interest.

Today, computations and simulations are performed on massively parallel computers leading to thousands of datasets where each datasets can be on the order of gigabytes. Visualizing and quantifying such data cannot be done on a single processor machine. Therefore, a distributed form of the feature extraction and feature tracking algorithms is required. In the Vizlab, we have developed a number of tools to extract and track features on a parallel cluster. However, there are cases where one would like to handle large datasets. In this thesis, we extend the feature extraction and tracking library to perform feature extraction on a large datasets with multiple steps, i.e., reading in only a portion of the data at once. In addition to the distributed feature tracking, we also enhanced the visualization component so that features can be accessed and rendered in a more intuitive fashion.
Acknowledgements

I would like to thank my adviser, Professor Deborah Silver for giving me the opportunity to work under her, and for the guidance and support, she has given me as an Adviser and Professor in the interactions that I have had with her in the last one year.

I am grateful to Prof. Marsic and Prof. Wilder for taking the interest and sparing some time from their busy schedule to be part of my thesis defense committee.

I would also like to thank my colleagues at Vizlab - Kristina and Rohini for their insightful suggestions and camaraderie, which helped me throughout my work on this thesis.

I would like to acknowledge Prof. Parashar and TASSL lab who gave me permission to run my code on their cluster and the CAIP staff - James Chun, William Kish and Steve Carter for their ready assistance whenever I needed it. I also wish to thank Dr.Gimenez and Hyen Chung Chun of the Department of Environmental Sciences at Rutgers University for their assistance in providing me with test data for this thesis.

I am thankful to my parents for their continuous love and support that has brought me to where I am today and for the guidance and thoughtful advice of my friends without whose companionship my Masters would be incomplete.
Dedication

To my mother Mrs. Urmila Dhume

my father Mr. Pradeep Dhume

and my sister Ms. Mrunmayi Dhume
Table of Contents

Abstract ................................................................. ii
Acknowledgements ...................................................... iii
Dedication ................................................................. iv

List of Tables ........................................................... viii
List of Figures ........................................................... ix

1. Introduction ......................................................... 1

2. Related Work ....................................................... 5

3. AVS Based Feature Tracking Code - Ostrk2.0 Package ............. 7
   3.1. The AVS Platform ............................................. 7
   3.2. Overview of Algorithm used ................................ 9
   3.3. Features of the Ostrk2.0 package ......................... 11
   3.4. The Interface ............................................... 13
   3.5. Using the software ........................................... 17
   3.6. Code behind the Ostrk2.0 software ...................... 20
   3.7. Revisions and Additions to the Ostrk2.0 package .......... 29
       3.7.1. Enhancements to Feature Tracking Code ............. 29
       3.7.2. Code clean-up and bug fixing ....................... 39

4. Distributed Feature Extraction and Tracking Code ................. 42
   4.1. Distributed Algorithm ....................................... 42
       4.1.1. Distributed Extraction: .............................. 42
       4.1.2. Merging strategy: ................................... 43
Appendix B. ................................................................. 104
   B.1. Perl script for automated running of feature extraction by parts .... 104

Appendix C. ................................................................. 111
   C.1. Distributed Feature Extraction and Tracking Input File Formats ... 111
       C.1.1. The vortsname file ........................................... 111
       C.1.2. The datainfo file ........................................... 111
       C.1.3. The datafile.list file ...................................... 112
   C.2. Distributed Feature Extraction and Tracking Output File Formats . 112
       C.2.1. The .dat file .................................................. 112
       C.2.2. The .attr file .............................................. 113
       C.2.3. The .oct file ................................................ 113
       C.2.4. The .trakfile file ......................................... 113

References ................................................................. 114
List of Tables

6.1. Time taken for large scale feature extraction . . . . . . . . . . . . . . . . 76
6.2. Feature count before and after global merging . . . . . . . . . . . . . . . . 86
List of Figures

1.1. The Feature Tracking based Visualization pipeline [3] . . . . . . . . . . 3
3.1. View of the AVS/Express Network Editor . . . . . . . . . . . . . . . . . 8
3.2. The Ostrk2.0 feature tracking algorithm . . . . . . . . . . . . . . . . . 10
3.3. Color coding when objects merge - Snapshot of consecutive timesteps
   from the vorticity data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.4. trk3d interface for user controlled visualization . . . . . . . . . . . . . 13
3.5. d3_fld_trk input interface . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.6. High-level block diagram of Ostrk2.0 Feature Tracking Package . . . . 17
3.7. Organization of code in Ostrk2.0 . . . . . . . . . . . . . . . . . . . . . . 20
3.8. The $d3_{,}fld_{,}trk$ network . . . . . . . . . . . . . . . . . . . . . . . . 21
3.9. The two basic types of meshes . . . . . . . . . . . . . . . . . . . . . . . . 24
3.10. Flowchart showing Surface Area calculation steps . . . . . . . . . . . . 31
3.11. The Initial Coloring Method selection scheme . . . . . . . . . . . . . . 33
3.12. Snapshot of one time-step of the vorticity dataset colored by mass . . 34
3.13. The color by mass transfer function . . . . . . . . . . . . . . . . . . . . 35
3.14. One timestep of the vorticity data colored randomly . . . . . . . . . . . 36
3.15. Algorithm used for the random coloring scheme . . . . . . . . . . . . . 37
3.16. A visual description of color tracking . . . . . . . . . . . . . . . . . . . 40
4.1. An illustration of overlapping processor partitions . . . . . . . . . . . . 43
4.2. The complete merging algorithm [3] . . . . . . . . . . . . . . . . . . . . 44
4.3. The partial merging algorithm [3] . . . . . . . . . . . . . . . . . . . . . . 45
4.4. An example of Distributed Tracking [3] . . . . . . . . . . . . . . . . . . 47
5.1. Illustration of modified dataset partitioning scheme . . . . . . . . . . . 55
5.2. Messaging between processors in adjacent parts . . . . . . . . . . . . . 58
5.3. Format of (a)A .msg file and (b)A local object table . . . . . . . . . . . . . . . 59
5.4. Sequence of tasks executed in Perl script . . . . . . . . . . . . . . . . . . . . 61
5.5. List processing for out-of-bounds error . . . . . . . . . . . . . . . . . . . . . . 65
5.6. Illustration of complex object overlap between processors . . . . . . . . . . 66
5.7. Single machine running for the entire dataset . . . . . . . . . . . . . . . . . . . 71
6.1. soil2.raw . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
6.2. soil11.raw . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.3. soil3.raw . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.4. Partial extraction results of soil2.raw for processor0, 2-D and 3-D . . . . . . . . 77
6.5. Partial extraction results of soil2.raw for processor8, 2-D and 3-D . . . . . . . 77
6.6. Partial extraction results of soil2.raw for processor16, 2-D and 3-D . . . . . . 78
6.7. Partial extraction results of soil2.raw for processor24, 2-D and 3-D . . . . . . 78
6.8. Partial extraction results of soil11.raw for processor0, 2D and 3D . . . . . . . 79
6.9. Partial extraction results of soil11.raw for processor8, 2D and 3D . . . . . . . 79
6.10. Partial extraction results of soil11.raw for processor16, 2D and 3D . . . . . . . 80
6.11. Partial extraction results of soil11.raw for processor24, 2D and 3D . . . . . . . 80
6.12. Partial extraction results of soil11.raw for processor32, 2D and 3D . . . . . . . 81
6.13. Partial extraction results of soil11.raw for processor40, 2D and 3D . . . . . . . 81
6.14. Partial extraction results of soil3.raw for processor0, 2D and 3D . . . . . . . 82
6.15. Partial extraction results of soil3.raw for processor8, 2D and 3D . . . . . . . 82
6.16. Partial extraction results of soil3.raw for processor16, 2D and 3D . . . . . . . 83
6.17. Largest extracted feature from the soil 2 dataset in 2D and 3D . . . . . . . . . 84
6.18. Largest extracted feature from the soil 3 dataset in 2D and 3D . . . . . . . . . 85
6.19. Largest extracted feature from the soil 11 dataset in 2D and 3D . . . . . . . . 85
6.20. Largest extracted feature from the soil 15 dataset in 2D and 3D . . . . . . . . 86
6.21. Partial extraction results for part0 of soil15.raw . . . . . . . . . . . . . . . . 87
6.22. Partial extraction results for part 1 of soil15.raw . . . . . . . . . . . . . . 88
6.23. Partial extraction results for part 2 of soil15.raw . . . . . . . . . . . . . . 88
6.24. Partial extraction results for part 2 of soil15.raw on processors 0, 8, 16 in 3D ................................................................. 89
6.25. (a) Illustration of part-wise division and (b) Final merged result for soil15.raw 90
6.26. Two views of the iso-surface visualization for soil15.raw .................. 91
6.27. Comparison of AVS based feature extraction and Large scale feature extraction results .................................................. 92
6.28. Visualization network used to view partial and complete results .......... 93
Chapter 1
Introduction

Scientific visualization is concerned with presenting data to users, in the form of images. The goal of this area is usually to provide a more intuitive interpretation of the data being presented for hypothesis building and modeling. Examples of this are the visualization of intense quantities of laboratory or simulation data, or the results from sensors out in the field. Another allied field is data mining that involves sifting through large amounts of data and picking out relevant information. Scientific visualization usually deals with data that has a natural geometric structure (e.g., MRI data or wind flows). The outputs of these simulations and experimental and sensor data are in the form of 3D scalar datasets. The soil sub sample data and fluid vorticity data used as test data for the work done in this thesis are examples of these.

In addition, time varying simulations are common in many scientific domains; these are used in the study of evolution of phenomena or features. Traditionally, visualization of 3D time-varying datasets is done using animation, i.e., each frame (time-step) is visualized using iso-surfacing or volume rendering, and then the various time-steps are run in sequence to decipher any visual patterns. However, for datasets with continuously evolving features like cloud formations, it is difficult to follow and see patterns in 3D. What is required is a technique to isolate and analyze certain regions of interest also called 'features' and highlight their underlying physical processes [6, 11]. For example, it is usually important to know what regions are evolving, whether they merge with other regions, and how their volume may change over time. Therefore, region based approaches, such as feature tracking and feature quantification are needed. Moreover, most of the standard visualization methods cannot give a quantitative description of the evolving phenomena, such as the centroid of a merged region or the value of its
moments. An automated procedure to track features and to detect particular stages or events in their evolution can help scientists to concentrate on regions and phenomena of interest by providing visual cues by coloring connected iso-surfaces based on computed quantifications. This not only effectively reduces the amount of data to focus on, but also curtails the visual clutter. Another important application of feature tracking is in data mining. By building a database of features, the results over multiple simulations can be used for 'event matching'. In previous work [19, 20, 4], the Vizlab has pioneered the use of feature tracking to effectively visualize time-varying datasets. The basic feature tracking based visualization pipeline is shown in Figure 1.1. The major steps include feature extraction, tracking, quantification, and enhanced visualization:

- **Feature Extraction** - The first step is to identify and segment features of interest from the dataset to be tracked. The method used depends on the definition of a 'feature', which can differ from domain to domain. Usually features are defined as threshold-connected components [17, 18, 19].

- **Feature Tracking** - In this step, the evolution of the extracted features is followed over time noting various events that occur during their evolution.

- **Feature Quantification** - Once features are extracted, they are quantified and information about them, e.g., mass, centroid, etc. can be calculated.

- **Enhanced visualization and event querying** - Using the accumulated tracking information, we can also perform additional visualization steps like event querying which involves gathering information leading to a certain event of interest or present a new visualization using the data (metadata) collected. One example of this is volume rendering of an individual feature.

These days, due to the increasingly large scientific computational power available, computations and simulations are performed on massively parallel computers, leading to very large datasets of several hundred million datapoints and thousands of time-steps. Visualizing and quantifying such data is difficult because of its time and complexity and cannot be done on a single processor machine. Therefore, we need to modify and
generate a distributed form of these feature tracking and quantification algorithms in order to accommodate such data. Each processor, over which the dataset is divided, will extract and quantify its local features. We then need a procedure to coalesce these observations, because a feature may span multiple processors, i.e., perform global feature identification and resolution, and global calculation of feature attributes once this coalescing is done. Prior work done in this area is documented in [10, 3]. We also need to take into account the individual processor constraints, e.g., RAM and hard-drive memory specifications. Hence, modifications are necessary to handle large-scale datasets and perform the feature extraction in parts. This also necessitates special changes to be made to the ensuing merging code used for the coalescing step, in order to accurately merge and quantify the large amounts of data involved. These modifications are the primary focus of this thesis.

As research and data collection techniques evolve, there is a need for improved and enhanced feature tracking and analysis. For example, the ability to do correct color tracking over multiple timesteps to follow regions of interest closely and the ability to
have different initial coloring schemes for coloring individual features based on one’s requirements. Furthermore, in some cases, a need may arise to perform feature tracking in multiple steps (timesteps other than ’1’), either for the purposes of looking for an irregular pattern or to skip timesteps in between to look for a particular event. In addition, more detailed analysis, for example, the calculation of surface areas of connected regions of interest along with the usual centroid and moments information are important sometimes to make specific observations or interpretation of certain data. Such enhancements are also discussed and demonstrated in this work.

The organization of this thesis is as follows: Chapter 2 is an overview of related work done in this field. Chapters 3 documents the AVS code for feature tracking mentioned in [41]. It also documents the modifications and enhancements added as referred to above. Chapter 4 provides a brief description of the distributed code for feature extraction and tracking. Chapter 5 details the algorithms and implementation done to modify the existing Distributed code to handle very large datasets on limited processor configurations. Chapter 6 presents the results obtained for the datasets tested on the code developed in Chapter 5. Finally, Chapter 7 discusses merits and limitations of the system and further work.
Chapter 2

Related Work

The work in this thesis is based upon enhancements to the feature extraction and tracking of [10, 3]. Below we present a brief review of these papers.

In this context, when we say feature tracking we refer to the generalized definition of 'features' as threshold-connected components obtained using a region growing algorithm [19, 20, 17, 7] without any specific domain knowledge. This definition may be extended depending on the application domain. These features or regions are similar to those obtained using the image segmentation and watershed algorithm presented in [13, 14]. Rather than just simple iso-surface visualization, feature extraction does an in-detail analysis and computation of the attributes of each feature extracted. Simple iso-surfacing treats each cell of the object independently and is inherently parallelizable. Feature analysis, on the other hand, requires complete knowledge of the feature [17] in question and any distributed algorithm for feature extraction must take that into consideration. The term feature tracking as applied here can be defined as the correlation of extracted features of a dataset over time. The concept was first put forward in [18], and an algorithm devised in [22] labeling it the 'correspondence problem'. The work done in [22] setup a basic framework for analyzing time-varying datasets.

[10] presented a distributed system for feature extraction on structured and unstructured datasets within a parallel computing environment. It gave an algorithm for dividing a single dataset (which cannot be fit into the memory of a uniprocessor machine) among multiple processors using slab-based partitioning for structured and cell-based partitioning for unstructured grids. Inter-processor communication via a message-passing interface (MPI) was used to coalesce the extraction results of individual processors. Features shared by adjacent processors were recreated using a parallel
merging technique. Parallel iso-surfacing using the 'Marching cubes’ algorithm was used for visualization.

The feature extraction in [10] was extended in [3] to form a complete distributed feature tracking system for multi-resolution grids, in particular, AMR (Adaptive Mesh Refinement) data. The implementation was done in a computational steering environment in order to do the visualization interactively in-situ. For this, it used the GRACE (Grid Adaptive Computational Engine) infrastructure [16]. A way to do real-time feature extraction and tracking was also given and implemented. To access the ongoing simulation, it used the DISCOVER (Distributed Interactive Steering and COllaborative Visualization EnviRonment) portal [15]. The approach suggested an alternate partial merge strategy for coalescing to reduce inter-processor communication time and a similar technique called partial tracking for the distributed tracking.

There have also been numerous other approaches to do ‘parallel and distributed visualization’. A technique based on ‘static partitioning’ is given in [2]. [8] presented an approach to do visualization using parallel iso-surfacing. A parallel volume rendering technique is introduced in [12]. However, connected components (features) provide much more information since we can perform attribute analysis on them. [21] presented an approach to do feature tracking on large scale datasets. However, the approach uses machine learning and needs special graphics hardware for visualization.

In this thesis we build on prior work done and give a general-purpose algorithm for doing feature extraction on large scale datasets extending the approach in [3]. The algorithm suggested provides a way to overcome the difficulty of handling a large dataset all at once (simply dividing the entire dataset among processors) using a ‘divide and conquer’ approach to the problem.
Chapter 3

AVS Based Feature Tracking Code - Ostrk2.0 Package

The Ostrk2.0 package is stand alone feature tracking software developed in C/C++ on the AVS/Express 6.2 platform by Vizlab [9]. This section of the thesis serves as a ready reference for users who are familiar with feature tracking and what to learn, use and expand this software as per their requirements.

3.1 The AVS Platform

AVS/Express, a product of Advanced Visual Systems, is an object-oriented versatile data visualization toolkit that allows you to create reusable application components or even complete elaborate data visualization applications. Fast data analysis, abundant visualization options within an intuitive graphical development environment make AVS/Express an excellent choice for any data visualization task. [1]. Figure 3.1 shows a snapshot of the graphical development environment within AVS/Express - the network editor.
Figure 3.1: View of the AVS/Express Network Editor
3.2 Overview of Algorithm used

The basic algorithm used for the feature tracking is an output-sensitive tracking algorithm [9]. It improves upon the efficiency of the algorithms in [19] by splitting the volume-overlap test (VO-test) into two steps.

In the first step, all potential features overlaps are determined by a simple binary overlap test (1-overlap, 0-no overlap) and stored into groups. In the second step, 'best matches' are determined by first testing amount of overlap within groups and then detecting events like bifurcation, creation, etc., thus reducing calculations by skipping overlap tests on features which do not potentially overlap. The overall complexity is reduced from $O(n \log n)$ to $O(n \log k)$, $n$ being the number of cells of the features in the time-step being tracked and $k$ being the number of cells of the features of the group being tracked. Thus, the algorithm is output-sensitive because it depends on 'k', which in turn is dependent on the tracking results (overlap relationships), and nature of the dataset. However, this algorithm is also less iso-value threshold dependent than most algorithms in that, if a user changes the threshold value, then if the features shrink (i.e., increase in threshold), then the previous group information can be used for the first step of the VO-test instead of starting from scratch. Figure 3.2 shows a flowchart of these 2 steps.
Figure 3.2: The Ostrk2.0 feature tracking algorithm
3.3 Features of the Ostrk2.0 package

The main utilities offered by the software are summarized as follows:

- **Feature Extraction** - The input dataset is segmented into its features as threshold-connected components based on an threshold pre-specified by the user. The user has to choose a percentage threshold. The actual value of threshold is then calculated by:

  \[ \text{actual\_thresh} = p \times (\text{max\_node\_value} - \text{min\_node\_value})/100 \]

  where \( p \) is the percentage threshold selected. This calculation is performed on a per frame (timestep) basis.

- **Feature Tracking** - The life-cycle of all extracted features is tracked over the number of time-steps of the dataset specified recording all ’events’ that may occur during an objects’ life-cycle specifically merging, splitting, continuation, dissolution or creation.

- **Enhanced Surface Animation** - Users can view an iso-surface visualization of all time-steps with color coding added to highlight feature events. For example, suppose a feature A in time step 1 splits into features B and C in time-step 2. Then both features B and C will receive the same color as A. Note that the color given to A depends on the initial coloring scheme set (discussed in detail in later sections).

![Figure 3.3: Color coding when objects merge - Snapshot of consecutive timesteps from the vorticity data](image)
• **Enhanced Volume Animation** - In addition to enhanced surface animation, there is also a provision for volume animation. In this mode, users can view a volume rendered animation of the time-steps wherein all voxels outside the desired features have been given node values of '0' thus reducing unwanted crowding resulting in a cleaner visual effect.

• **Surface Isolation Animation** - The interface in Ostrk2.0 allows you to select a particular feature from the surface animation window (last time-step only) and view its evolution separately in a different window.

• **Attribute analysis and Printing** - The software computes various attributes about the features extracted during the segmentation phase. These attributes like volume, mass, centroid, etc., can be printed on the screen by picking a particular object from a time-step in the enhanced surface animation window.

• **Graph Plotting** - The interface also has a window where the user can view how some frame attributes like number of objects, etc., vary over the time (duration of the tracking).

• **Storing of Feature tracking results** - All attributes for individual objects as well as for all time-steps are stored in files in a pre-defined directory under the users’ run path. The files also include a record of the events which occur in the life-cycle of an object, e.g., splitting or dissipation.
3.4 The Interface

The interface of Ostrk2.0 is intuitive and modular. The following figures explain the interface.

Figure 3.4: trk3d interface for user controlled visualization

Figure 3.4 is a snapshot of the output trk3d interface. This is where the user can interactively view the tracking results. Surface and volume animation, isolated surface animation and object attribute display are all done within this interface. The interface can be roughly divided into 5 parts - 4 output windows and a control panel to the right that is used to control them.

- **Surface Animation window** - The window to the top left is the surface animation window. The user can view an iso-surface animation of the frames being tracked in this window. The iso-surface information is taken from the .poly files.
of the time-steps created during feature extraction.

- **Isolated Surface Animation window** - The window directly below the surface animation window is the isolated surface animation window. Here, the user can view the tracking history of a single feature selecting it from the surface animation window. This tracking history is available only when a feature from the final frame is selected.

- **Volume Animation window** - This window is the one on the top right. If feature tracking with volume rendering is chosen, the user can view a volume rendered animation of the time-steps in this window.

- **Graph Plotting window** - This window lies below the volume animation window. All graphs of attributes pertaining to the feature tracking are displayed here.

- **Control Panel** - This panel allows the user to control the size of the 4 windows above as well as what is displayed in them. It can be sub-divided into:

  - **Animation Control** - These buttons allow the user to select a particular window and control the animation loop using the play, reset, forward, back and cycle buttons.

  - **Plot Control** - The user can select the type of graph he wants using a drop-down list provided. He can also choose a gridded or non-gridded graph display as desired.

  - **View Control** - The user can select a window (R1, R2, R3 or R4 buttons) and adjust its height and width using the buttons provided.

  - **Attribute Printing window** - An object is selected from the surface animation window by simultaneously holding the Ctrl key and clicking on it. The attributes of a selected feature are displayed in this window. There are adjustable scrollbars provided and a button to clear its contents.

Figure 3.5 provides a view of the interface panel via which the user passes input to be processed to the d3_fld_trk application.
• **field file basename (text input field)** - The user types the path and basename of the dataset field files here.

• **start\_val,end\_val and incr(text input field)** - The user enters the starting and ending frame numbers and increment in the dataset field files here.

• **precision (slider)** - The user can use this to select the precision in the field file numbers using this slider.

• **small obj sld (slider)** - Using this slider control, the user selects the minimum volume limit for an object to be considered during tracking.

• **thresh dial (dial)** - The threshold can be selected as a percentage between 0 and 100 using this dial.

• **track\_now, vol\_render and reset (toggle buttons)** - These buttons are used to toggle start/stop the tracking mode, select/deselect the volume rendering option and reset the tracking mode respectively.
Figure 3.5: d3_fld_trk input interface
3.5 Using the software

The main application in Ostrk2.0 software is the $d3_{.fld_{trk}}$ which is used for feature tracking of 3D datasets. To understand how to use the package, we will first look at the high-level flowchart depicting the steps in the feature extraction and tracking process.

![High-level block diagram of Ostrk2.0 Feature Tracking Package](image)

Figure 3.6: High-level block diagram of Ostrk2.0 Feature Tracking Package

- **Input Stage** - The input stage is where the user can feed in the dataset sequence on which the tracking is to be performed and also enter pertinent parameters. The key parameters to be entered are:

  - **Dataset file name** - The user needs to specify the name of the files to be tracked. Actually, the user specifies the *labelname* for the time-series to be tracked and their complete path location on disk. For example, for the vorticity dataset, the files are named vorts1.data, vorts2.data, and so on. Hence the labelname to be entered is 'vorts'. The pathname also points to the location of the field files (.fld) for the files rather than to the actual dataset files: this point will be explained further in the next section.

  - **Start value, End value and Increment** - The user has to enter the number of the timesteps from where tracking is to start, end and also the increment in which files should be tracked. For the vorticity dataset, if the
start value, end value and increment are 11,13 and 1 respectively then the
files which are going to be tracked are vorts11, vorts12 and vorts13.

– **Precision** - This indicates the style of numbering of the files to the software.
  For example, if the files to be tracked are named vorts01,vorts02, etc., the
  user still enters 1 and 2 in the start and end values respectively and chooses
  a precision of 2. This is done in order to keep the values passed as start and
  end values uniform without any trailing zeroes. The default precision is 1.

– **Small Object Volume Slider** - When tracking dataset frames with a large
  number of small objects which are extraneous to the regions of interest, the
  user can choose to blank out these unwanted features having volume (defined
  as number of nodes) lower than a certain limit by using this slider.

– **Percentage Threshold** - As mentioned earlier, the user specifies the thresh-
  old for feature extraction as a percentage rather than an absolute value.

After entering in all the relevant parameters, the user can begin the extraction
and tracking process by clicking on 'Track Now' toggle button. He can also choose
whether to have tracking with volume rendering or without. The advantage is
that if there is no real need for volume rendering for a particular case, then the
tracking can be done faster by simply choosing to track without it.

• **Feature Extraction and Tracking Stage** - Clicking the 'Track Now' button in
  the user interface spawns a sub-network which performs the feature extraction and
  tracking. Extraction and tracking is done in a multi-pass process. In each pass,
  the extraction and tracking process is performed on adjacent timesteps. Finally
  the results of the tracking are stored in a trakTable that contains correspondences
  between objects in successive frames.

• **Visualization Stage** - The visualizer stage is responsible for taking all the in-
  formation gleaned and analysis done and outputting it in a form which the user
  can understand easily. Generating renderable objects which are sent to the dis-
  play for the volume animation as well as forming an object sequence which is the
evolution history of a selected object for the isolation animation are all functions of this stage.

- **2D to 3D converter stage** - For tracking 2D datasets, the user has to run the `d2 fld trk` application of the Ostrk2.0 package. This is similar to `d3 fld trk` except for an additional 2D-to-3D stage. The 2D dataset is first converted into a 3D dataset using the 2D-to-3D convertor stage and then tracking is performed. Basically, this stage adds another layer of nodes to the 2D dataset and gives them the min/max node value. Note, such a dataset cannot be volume rendered.
3.6 Code behind the Ostrk2.0 software

The code for the application is organized into 4 major directories - interf, os, ftrack and d2tod3.

- **interf** - All the code associated with the modules concerned with user interface functionality is contained in here.

- **os** - All the code associated with reading in the field structure for each dataset and performing feature extraction resides here.

- **ftrack** - All the code associated with doing feature tracking and the ensuing visualization resides here.

- **d2tod3** - All the code for converting a 2D dataset into a 3D one to facilitate tracking on it is stored here.

This section lists the modules which bring about the desired functionality of the software and gives a more detailed outline of the actual code structure for users who may want to modify the package to suit their requirements. A network editor view of the actual
Ostrk2.0 AVS $d3_{\text{fld}}_{\text{trk}}$ application is shown. The figure depicts the network after it has already spawned the tracking and visualization sub-network $trk3d$.

![Diagram of $d3_{\text{fld}}_{\text{trk}}$ network](image.png)

Figure 3.8: The $d3_{\text{fld}}_{\text{trk}}$ network

- **vorts\_trk\_ui (macro)** - This macro basically contains all the sub-items that go into the data-entry panel where the user inputs data parameters at the beginning.

- **loop(module)** - This is a pre-defined AVS module. It basically serves as a looping mechanism and can output running counter values based on input consisting of a start\_val,end\_val and increment. The output values are then used to perform a repeating operation much like the general purpose for-loop in C. For our purpose the loop macro takes in its input from the vorts\_trk\_ui macro and is used for looping the field reading and other processing functions that follow.

- **create\_listfile(module)** - This module uses the information input by the user
(vorts_trk.ui) to create a list of data files that need to be processed (feature extracted and tracked). It builds the names of files in the list by using the dataset label provided initially. The list at its output is passed on to the genfldfile and spawn_trk modules. For example, the list file vorts_comp011131.list contains the following:

```
/caip/u61/pinakin/msthesis/ftrack-AVS/vortsfld/GENERATED_TRACK_FILES/
vorts11 vorts12 vorts13
```

The first line specifies the absolute path where all output files will be created (including the listfile). The default location is under a new directory called /GENERATED_TRACK_FILES under the directory where the field files are found. The following lines till the end of the list file are the constructed names of files to be read and processed by the ensuing modules. Note that the name of the listfile includes the information what component number of the input data is used (in this case, the default, comp0). The code behind this module lies in the file create_listfile.cxx under the interf directory. Here the user can modify logic to create the list file.

- genfldfile(module) - This module is an intermediate step between the previous module and the Read_Field macro. It basically parses the listfile create above and constructs field file names and passing them as input for the Read_Field macro which processes the actual field file. The code behind this module is in the correspondingly named .cxx file in interf dir.

- Read_Field(macro) - This is another pre-defined macro which implements one of the basic but essential data importing steps for 3D dataset visualization in AVS. It reads in the field file for a particular dataset and creates an AVS field structure (that describes the information contained in the dataset) for it.

**The AVS field structure:** To support the different types of datasets that exist in terms of their shapes and varying levels of uniformity, AVS has developed a general-purpose data-structure called a field. A field completely describes an input dataset in terms of dimensions, number of datapoints, type of dataset, etc.
Therefore, once a dataset has been read in into a field structure you can use any of a vast number of AVS visualization techniques to render it. The basic components of an AVS field are discussed below [1]:

- **Grid** - The grid defines the locations in space of the nodes. Therefore depending on how regular your data is, you can either define just the coordinates of the boundary of the grid or explicitly mention the location of each node.

- **Node Data** - The data value at each node. This can be scalar, for example, mass, or a vector, for example, components of velocity along each direction.

- **Cells** - A cell defines connectivity between nodes enabling surface rendering of the grid. This connectivity can be implicit or may need to be specified. A field may have multiple cells. Cells of a single type make up a cell set. Common types of cells include triangular, quadrilateral, tetrahedral, hexahedral, etc.

- **Cell Data** - This is similar to node data but concerns the group of nodes which define a particular cell.

- **Mesh** - The cell sets and the grid together form the mesh.

AVS/Express defines four main mesh types that differ in the amount of structure or uniformity they contain [1]:

- **Uniform** (a sub-class of the structured field type)

- **Rectilinear** (a sub-class of the structured field type)

- **Structured** - For structured meshes, usually AVS express can determine the location of the nodes and connectivity information based on the structure itself. For example, in the case of uniform meshes, the distance between any two nodes along a given axis is the same. Hence, you need specify only the coordinates of the minimum and maximum nodes in the data set. For rectilinear meshes, the distance between nodes along a given axis may vary, so you need to specify the co-ordinates of all nodes along the axes.
Co-ordinates of all other nodes can be deduced by the intersections of lines passing through the axis nodes. In such structured data sets, the connectivity can be computed by AVS/Express based on the order in which nodes are provided and the datasets’ dimensions. A simplistic way to think of this connectivity algorithm is that for each row, the first node connects to the second node, the second node connects to the third, and so on. Each node in a row also connects to the nodes directly above and below it in the rows above and below that row.

– **Unstructured** - For unstructured meshes, you must provide the coordinates of every node in the data set explicitly. This is the only type of mesh where you have to specify cell sets.

The three structured mesh types - structured, rectilinear, and uniform mesh, differ from an unstructured mesh in that they all specify dimensions. The dimensions of a mesh are the number of nodes along each axis of the grid. Pictured below are two meshes, an unstructured and a structured mesh.

![Unstructured Mesh](image1.png) ![Structured Mesh](image2.png)

Figure 3.9: The two basic types of meshes

You cannot specify the number of nodes in any direction in the unstructured mesh. However, in the structured mesh above, the dimensions of mesh is 2x5 (2 nodes in each of 5 rows).

**AVS Field File format:** To build a field data-structure for a dataset, AVS requires that you have a corresponding field file (.fld) which provides some basic information about the dataset. Consider the following field file which describes
one time-step of the vorticity dataset:

```
#AVS field file #First 5 chars must be #AVS
ndim=3 #number of dimensions
dim1=128 #dim along axis 1
dim2=128 #dim along axis 2
dim3=128 #dim along axis 3
nspace=3 #No. of physical co-ordinates at each point
veclen=1 #No. of data values at each point
data=float #Data Type - byte, float, integer or double
field=uniform #Field Type - uniform, rectilinear or irregular

#Instruction on reading data for each data component
variable 1 file=/caip/u61/pinakin/msthesis/ftrack-AVS/vortsdata/vorts11.data filetype=binary
```

The file format for the most part is explained above. The line beginning with the AVS command variable indicates the actual location of the data to the `Read_Field` macro.

- **spawn_trk(module)** - This module reads in information from the newly created field structure and does pre-processing before the tracking and visualization macro takes over. It sets certain parameters in order to pass on relevant information to the trk3d macro. It checks whether:

  - The field is uniform or not which decides whether a .vol file will be generated or not.
  - The number of cell sets present (using FLDget_cell_set method).
  - The type of each cell set. Only prismatic, hexahedral and quadrilateral types are supported by the software so far.
  - It sets a parameter IsTracking = 1 to indicate to trk3d that tracking should be done and also passes all necessary parameters from vorts_trk_ui via V-commands.
  - Finally, it checks whether the trk3d network has already been spawned in which case it resets the connections to it.
• **trk3d(macro)** - This macro is the main block of the package and consists of sub-modules to do feature extraction, tracking and visualization on the input data. This macro also provides the interface for viewing the visualizations under user control. The different modules inside can be categorized into modules for extraction, tracking or visualization and are individually explained in further detail below:

  - **os(module)(feature extraction)** - This block is primarily concerned with feature extraction. All of the main attribute files such as the .attr, .ucoed, .vol(if Volume Rendered),.poly and .trak are created during processing by this module. The main source file for this module is os.cxx and this and all other associated code files are present in the os directory.

  - **os.cxx** - This file contains the main code for the os module. Here, the field structure is analyzed and the appropriate tracking function depending on the cell type.

  - **obj.hpp** - This file contains the functions which do the object analysis and attribute writing to files.

  - **seg(.cxx/.hpp)** - These files contain functions which perform the actual segmentation of the dataset.

  - **input(.cxx/.hpp) and obj(.cxx/.h)** - These files define data structures for storing list of objects and object information.

  - **color(.cxx/.hpp)** - These files contain methods for the initial color coding of iso-surfaces in each time-step.

  - **genOrigVolfldfile and genOrigVolfldfile2(module)** - These modules are used to generate field file names for volume rendering ( _vol_orig.fld file) given either a listfile and time or polyfile name as input.

  - **unifd2tod3(macro)** - This macro is used by the d2 fld_trk application in the case of 2D dataset tracking for converting a 2D dataset to a 3D one. The source code for the associated module is in the d2tod3.cxx file under interf dir.
– ViewLooppoly(macro)(tracking and visualization) - This macro contains the main modules for feature tracking and visualization, all of which are explained further below.

  * ftrack(module)(feature tracking) - All the feature tracking functionality is contained inside this block including rewriting of .poly files based on color coding of events during feature evolution. The trakTable file is also generated by this module. It is part of the ViewLooppoly macro inside trk3d. Code files associated with this module are in the ftrack directory.

  trk.cxx - This is the main source file for the ftrack module. It contains the basic framework of the tracking mechanism. It checks to see whether the current frame being tracked is the first, last or one in-between and decides whether to do/continue tracking or not accordingly.

  ReadFiles(.cxx/.h) - These files contain methods to read in .trak,.oct or .list files for tracking purposes.

  tracking(.cxx/.h) - These files contain the actual tracking functions which perform the basic overlap and best match tests. It also contains methods to rewrite the new .poly and .trakTable files.

  * Looppoly(macro) - This macro houses the looppoly module and some standard AVS modules like surfloop. All the actual surface animation and isolation animation is controlled and executed by this macro.

    · looppoly - This module executes the surface animation. It reads the polygons from the .poly file of the current frame and sends them to the AVS port to be rendered. It also calculates a variable FormerObjNum which is passed to FindPickedNode module to calculate global index of a picked object (looppoly.cxx).

  * FindPickedNode(module)(visualization) - This module contains methods to calculate a global index for the object selected by the user(for either attribute display or isolation animation). This global index is then
passed to \textit{trkPick1} module. The source code is listed in \texttt{FindPickedNode.cxx} under ftrack directory.

* \textit{trkPick1 (module)}(\textbf{visualization}) - This module contains methods to read in the attributes of the picked object from the already written files(for displaying attributes of a picked object), and also to create an object sequence array to pass onto \textit{trkPick2mod} module for isolation animation purposes. The object sequence array basically contains the evolution history of the selected object. The associated source code files are \texttt{trkPick1.cxx} and \texttt{BldNodeRelatedGraph(.cxx/.h)}(under ftrack).

* \textit{trkPick2mod (module)}(\textbf{visualization}) - This module is for the purposes of isolation animation. It receives an object sequence array from \textit{trkPick1} module and reads and sends the polygons for the corresponding objects from the .poly files to the AVS port to be rendered (\texttt{trkPick2mod.cxx}).

- \textit{plot2d(module)}(\textbf{Graph Plotting}) - This module is contained within the \texttt{View3} macro inside \texttt{trk3d}. Its function is to plot graphs of selected attributes read from .attr or .uocd files. The code behind this module can be found in \texttt{plot2d.cxx} file under ftrack directory.

Note: The Ostrk2.0 package also contains other modules such as the \textit{compute_obj_no} and \textit{flip_flop} module used for debugging and the \textit{viewpoly} macro which can be used to directly render .poly files for testing purposes. \texttt{d2 fld trk}, as mentioned earlier, is the application for tracking 2D datasets. The \texttt{d3 hdf trk} application is for tracking 3D datasets in hdf5 format. A detailed explanation of both of these is beyond the scope of this thesis. The procedure for installing and running the Ostrk2.0 package is included as an appendix.
3.7 Revisions and Additions to the Ostrk2.0 package

The Ostrk2.0 package is a useful tool for feature tracking and visualization of 3D time-varying datasets. However, this package was developed some years ago necessitating some code cleanup and bug-fixing. For example, the color tracking mechanism which colors iso-surfaces to mimic the events occurring during their evolution had broken down and needed to be fixed. Also, some of the code was confusing and not organized correctly thus requiring the revamping of some of the code files. In addition, there was fresh interest in this tool requiring some enhancements to be added with a view to having improved feature tracking and analysis. All these modifications and additions are documented in this section in detail.

3.7.1 Enhancements to Feature Tracking Code

This section describes new features added to the Feature tracking package for increased functionality. This includes the calculation and printing of additional attributes (surface area), the capability to track in multiple time-steps if required, and the ability to choose from 3 initial coloring schemes as desired which can be selected from the user interface.

1. Surface Area calculation and analysis

The existing Feature Tracking code was replete with various object attribute analysis including object volume, mass, squared integrated content, calculation of centroid and mass moments of inertia. All this analysis is contained in the .attr and .uocd files that are generated during the feature tracking routine. However, for the soil datasets (in section 6) we also needed surface area.

The calculation of surface area is easily done because we already have the polygon information (node co-ordinates and triangle vertices) for all features of a time-step stored. Given this information, we can calculate the surface area of a single polygon (triangle) of a feature and hence the total surface area of that object. Recall that the surface area of any triangle can be calculated as:

\[ S.A = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)} \]
where:

\[ s = \text{semi-perimeter of triangle} = \frac{(a+b+c)}{2} \]

\[ a, b, c \] are the lengths of the 3 sides of the triangle.

Knowing this, we can calculate the surface area of any object by a summation of the areas of its constituent polygons. The flowchart depicting the steps used is shown in Figure 3.10. In the improved feature tracking code, an additional attribute file is generated (.addl) which contains the surface areas of all features for that frame also specifying the minimum and maximum surface areas for that frame and the objects with those area values.
Figure 3.10: Flowchart showing Surface Area calculation steps
2. Selection of Initial Coloring Schemes

Coloring of iso-surfaces for effective visualization is a very crucial part of the feature tracking paradigm. Color tracking enables us to follow regions of interest and their interaction with each other which might otherwise go unnoticed. Therefore, the initial coloring scheme chosen is very important and quite often domain specific. The original Ostrk2.0 package came with only a single default initial coloring scheme - to color the timeframes during feature extraction for eventual color tracking. The approach used was to color by volume. A simple transfer function was formulated which assigned a color between red (R=255, G=0, B=0) and blue (R=0, G=0, B=255) to each object of the frame based on its relative volume between the minimum and maximum object volume values for that timeframe.

As part of the work towards enhancement of the feature tracking code, two additional initial coloring schemes were added. The user can select a particular scheme at the initial interface screen where he enters other input parameters. Figure 3.11 shows a snapshot of the user interface with the radio button utility to select a desired coloring method.
Figure 3.11: The Initial Coloring Method selection scheme
**Coloring by Relative Mass** - This initial coloring scheme, which colors the objects of a time-step based on their relative mass, is very similar to the original color by volume scheme used. Here too, a transfer function is formulated that stretches between maximum and minimum value of mass for objects of a particular timeframe and assigns a color between red and blue according to the relative weighting factor calculated. The exact formulation of the transfer function is shown in Figure 3.13. This transfer function could be modified either by using a different weighting factor or maybe by adding a default starting value to R, G, B values or combinations of them. In fact, this type of coloring scheme based on variations in some property of the features in a dataset can be extended to any other attribute for example squared integrated content or maybe surface area.

![Figure 3.12: Snapshot of one time-step of the vorticity dataset colored by mass](image)
Figure 3.13: The color by mass transfer function
**Coloring Objects Randomly** - This is another method which can be used for the initial coloring of a series of time steps to be tracked. In some applications of feature tracking, it may be more useful to color the timesteps initially by a color scheme not based on any particular attribute. One way of implementing this is to use a pseudo-random coloring scheme for coloring objects such that each object in a frame has a unique color assigned to it picked at random. An implementation of this type of coloring was done and is explained below.

An initial color palette is established consisting of a number of predefined colors (at present 88). Each of these colors is chosen so that their RGB triplet is such that their shades are easily distinguishable from each other. During feature tracking, when colors are to be assigned to objects, the following pseudo-random selection scheme based on array shuffling is used. The colors in the Palette are indexed from 0 to num_cols-1. (num_cols is the total number of colors in the palette). From the preset palette (array) of colors, an index is chosen at random. The pseudo-random number generator used was the Mersenne Twister (from the Boost C++ libraries). The color at this index is then assigned to the first object. The color at this index is then swapped with the last color in the palette. The next time a random index is chosen between 0 and num_cols-2. This procedure is repeated for each subsequent object in the frame.

![Figure 3.14: One timestep of the vorticity data colored randomly](image-url)
Figure 3.15: Algorithm used for the random coloring scheme
In this way, we avoid repeating colors and a unique color is assigned to each object in a frame. Such a coloring scheme can prove useful in cases where a large number of similarly sized objects have to be color tracked - since each of them is assigned a unique color independent of any attribute, e.g., area, volume etc.. However, this coloring scheme is limited by the total number of colors in the preset palette and the need for them to differ significantly in shade from each other for effective color-coding.

If desired, a user can add as many coloring schemes as he wants by modifying the code in the color (.cxx/.hpp) files. For example, in some cases, depending on the application domain the user may want to highlight a certain region and give it a certain color while coloring the rest of the features a different color.

3. Non-consecutive time-step tracking

The basic feature tracking paradigm provided in the Ostrk2.0 package presents a way to correlate extracted objects between consecutive time frames. It also identifies and keeps a record of the events that occur in the evolution of a feature over the duration of the tracking. A key assumption of the model is that successive time-steps do not vary by much in time. However, in some situations it may be more beneficial to track objects over non-consecutive time-steps. This could be for several reasons, for example, a user may want to track events that occur in an object’s lifecycle at regular intervals separated by a known amount of time.

For this reason, the capability to track in time-step increments was added to the existing feature tracking software. To be able to handle any time-step increment changes need to be made to the code in several places. Some of the major changes are listed below.

- Changes to the list file creation module (interf) to create a correct list file based on the input increment so that the genfld module can feed the correct field file names to the Read_Field module.

- Changes to the looppoly module (ftrack) to be able to view a proper surface and volume animation of the non-consecutive time-steps to be tracked.
• Changes to the `trkPick1`, `FindPickedNode` and `trkPick2mod` modules (ftrack) to be able to handle attribute printing and isolation animation for non-successive time steps.

### 3.7.2 Code clean-up and bug fixing

**Modifications to color tracking code:**

The original color tracking mechanism worked as follows (Refer Fig. 3.16).

- An initial coloring scheme - based on the relative volume of the objects is applied to each time-step in the list of files to be tracked. This color information is written to the corresponding time-step’s .poly files along with other iso-surface connectivity information at the time of the .poly file creation during the feature extraction stage.

- During the feature tracking stage after the final .trakTable file is created, the .poly file of the first time-step is read in. Then, the objects in all subsequent frames are colored based on the .trakTable file recorded events:
  
  - If an object in the first frame (tracking is done taking two consecutive frames at a time) persists in the second frame, i.e., *continuation*, then it receives the same color as its counterpart in the first frame.

  - If an object splits into two or more separate entities in the second frame, i.e., *splitting* or *bifurcation*, then the child objects are given the same color as their parent object in the first frame.

  - If two or more objects combine to form a single entity in the second frame, i.e., *merging*, then the color of the largest object in the first frame is applied to the new merged object in the second frame.

- Based on this coloring algorithm, recoloring is effected by rewriting the .poly files of all the succeeding time steps with the new color information.

However, upon close inspection, it was found that the rewriting of color files had broken down and was not being done correctly leading to incorrect and incomplete color
tracking. The entire coloring tracking code was checked, debugged, and the .poly file rewriting part revamped to ensure accurate color tracking. The color tracking was checked on a number of datasets for varying number of time-steps to test its correctness.

![Figure 3.16: A visual description of color tracking](image)

**Code clean-up:**

It was found that the code was disorganized in some areas, leading to confusion in the location of methods and classes. Further, in some cases, functions were defined in header files themselves. This is a potential programming pitfall and although the code may compile on earlier compiler versions, it is a bad programming practice and may lead to linking errors post compilation such as duplicate instances of identifiers (variables or functions) within the object files. A related problem in multi-file programming is the compile time error due to duplicate definitions that occurs when a header is included.
more than once for a given source file (usually through other header files) - this leads to a class or structure being defined twice leading to the error.

To solve the first problem, most functions (except for inline functions and template functions) were moved out of the header files replaced by only a prototype declaration, as is standard practice. Inline functions are usually left inside header files because they are not compiled the usual way and are instead replaced by their contents wherever they are called necessitating a complete function definition instead of just a prototype declaration, i.e., the linker is not involved. The second exceptions are template functions: these are compiled not at their place of definition, but rather at their place of usage. Hence, they do not give the same errors as normal functions. Since templates are compiled when required, the implementation of template functions must be in the same place as their definition, i.e., within the header file.

The second error can be avoided by including only the headers you need for every source file. An additional safeguard is to use inclusion guards #ifndef and #endif which tell the compiler not to define a symbol if it has been already defined.
Chapter 4

Distributed Feature Extraction and Tracking Code

A distributed feature extraction algorithm was originally implemented and tested by Vizlab in [10] and expanded to include feature tracking in AMR datasets in [3]. In this thesis, we extend the algorithm used to accommodate very large datasets while taking into consideration the typical memory specifications available on most processors. To understand the modified algorithm, a brief overview of the original distributed feature extraction and tracking code is given in this chapter followed by the extended algorithm and its implementation in the following chapter.

4.1 Distributed Algorithm

The distributed algorithm can be divided into the following sub-parts:

- Distributed extraction
- Complete/partial merge strategy
- Distributed tracking using partial tracking algorithm and calculation of best match

4.1.1 Distributed Extraction:

The feature extraction algorithm used is a region-growing algorithm [23], where features in each dataset are segmented as connected-components based on a user-defined threshold. Like in the case of the undistributed algorithm, polygonal iso-surfaces and attributes are also calculated for the features. Initially, each processor which receives a part of the dataset independently performs this feature extraction locally, i.e., on its assigned part. Since the algorithm is a distributed one, a feature in the original dataset
can span multiple processor parts (potentially all processor parts). Hence, during the extraction process, each processor also checks to see if an extracted object contains any boundary cells. For this reason, adjacent processors share a common boundary region of 1 voxel thickness called a *ghost region*. Thus, if a processor has an object containing a boundary cell which is also contained in an object of an adjacent processor sharing that boundary with it, then those objects are actually a single object and need to be merged. Merging is accomplished by updating object tables and recalculating attributes and rewriting files.

![Figure 4.1: An illustration of overlapping processor partitions.](image)

### 4.1.2 Merging strategy:

The merging step requires some sort of communication between processors to exchange boundary information and corresponding iso-surface data. Depending on how communication takes place, two merge strategies are discussed.

- **Complete Merge Strategy** - In this strategy, all processors first initially perform feature extraction on their local data. Each processor forms a message containing information of objects with boundary cells and the corresponding cell information. All processors then communicate this boundary information using
a binary swap algorithm [3] as shown. A processor which receives the message updates its object table if it has any objects overlapping those boundary cells. At the end of this a global object table is formed at one processor which broadcasts this information back to all other processors for them to recompute attributes.

Figure 4.2: The complete merging algorithm [3]

- **Partial Merge Strategy** - In this strategy, all object connectivity information is not passed on in a single step. Initially, only adjacent processors share boundary information via message passing and only object tables on those processors are updated as necessary. After the first step, all local object tables are sent to a visualization processor which centrally computes a global table and uses this
information for visualization when required. For example, if processors P1, P2, and P3 have a single object overlapping between them, then during the first step P1 updates its object table stating that object A (P1) = B (P2). Similarly, processor P2 updates its object table stating that object B (P2) = C (P3). During the final merging step, the visualization processor will declare A, B, and C to be part of the same global object and recompute its global attributes. Further, the visualization processor uses this information to update .poly files (by giving all merged objects a single color) and .oct files (rewriting new attributes based on merged objects).

Figure 4.3: The partial merging algorithm [3]
The complete merge strategy requires a lot of communication - potentially log (n) time since a feature can span all ‘n’ processors and because resolving connectivity is inherently a serial task. The partial merge strategy is independent of number of processors used. Furthermore, the partial-tables sent to the visualization processor constitute a small overhead as compared to sending all the iso-surface information or part of the actual dataset.

4.1.3 Distributed Tracking and Best Match calculation:

The feature tracking algorithm is a distributed version of the one given in [18]. The algorithm consisted of two parts - the first where an overlap test is performed on all extracted objects from the current time-step with their next time-step counterparts. Doing only this is not sufficient because an object from the next time-step can overlap with an object in the current time-step without there being any relation over time between them. Therefore, the second part of the algorithm is used to calculate the best match among overlaps to accurately determine connectivity in time. The first part of the algorithm is easily transmuted to a distributed form using a technique similar to the partial merge algorithm used during feature extraction. In this case, assuming that the partitioning of all time-steps is identical, each processor calculates local overlap tables for its portion of the dataset. This method called partial tracking was introduced in [5]. However, the same cannot be applied to the best match part because if each processor were to calculate a best match on its own, this would result in erroneous results [3]. Instead, a central visualization processor reads is used to read in the local overlap tables generated by the individual processors, and compute a global overlap table and calculate the best match for every object in the dataset. The figure shows an illustration of feature tracking using the partial tracking strategy. In the final image of time-step 2, each feature gets the same color as its best match from time-step 1.
Figure 4.4: An example of Distributed Tracking [3]
4.2 Distributed implementation details

The code for the above, distributed algorithm for feature extraction and tracking is organized into four separate directories - objseg, finalmerge, ftrack and score corresponding to the sub-steps explained above.

- **objseg directory** - This folder contains the code for the feature extraction step. It contains implementations of the data-structures like node, cell and cell list, sorter to sort the cells, object and object list (which also stores boundary cells), and Global Table. Each processor loads its local portion of the dataset into these structures during the extraction phase. At the end of this phase, each processor generates a .poly file, .oct file (containing object attributes), .trak and a .table file (local object table) for its local region. The code in this directory can be compiled to create an executable using the MPI compile script: mpiCC -o object ./objseg/*.cpp -w This creates an executable 'object'.

- **finalmerge directory** - This folder contains the code for the second step of the partial merge. Since this program is run on a single processor, no MPI directives are necessary. The code includes methods to read in the .table files generated during extraction, compute a global object table, and update .poly and .oct files. When this program is run overall .poly and .oct and .trak files for the entire dataset are generated along with additional .dat file(Containing timestep information) and .attribute file(Containing additional attributes). This file is compiled using the command: mpiCC -o merge ./finalmerge/*.cpp -w

- **ftrack directory** - This folder contains the source files which implement distributed tracking through partial tracking. The code in this directory can be compiled using: mpicc -o ftrak ./ftrack/*.c -lm -w to generate an executable 'ftrak'.

- **score directory** - The best match calculation code is housed in this folder. The appropriate compilation command is:

  mpiCC -o match ./score/*.cpp -w
4.3 Running the distributed code

To perform feature extraction and tracking using the distributed implementation, we need to run the 4 steps involved one after another depending on whether we want to do only feature extraction or complete tracking, assuming that the *vortname*, *datainfo* and *datafile.list* files are filled correctly. These files contain the basic information about the dataset similar to the field file of the AVS feature tracking code. Their exact formats are included as an appendix. The steps for running the code on sample data of 3 time-steps from the vorticity data are given below:

**Run the Feature Extraction Code using the following script:**

```bash
mpirun -np [number_of_processors 'n'] -machinefile [optional list of processors] object
```

The Feature Extraction code will create 'n' table files.

**Run the merge code using:**

```bash
merge ./result/vorts11 'n' (Repeat for every timestep using same number of processors as in 1.)
merge ./result/vorts12 'n'
merge ./result/vorts13 'n'
```

**Run the Feature tracking code using:**

```bash
mpirun -np [n] -machinefile [optional list of procs] ftrak
```

**Run the best match code using:**

```bash
match ./datafile.list 'n'
```

The result folder will contain the final .traktable file containing object evolution history.
Chapter 5
Large Scale Feature Extraction

5.1 Limitations of the existing code

The existing work done on distributed feature extraction and tracking in [10, 3] provides a solid framework on which further work can be based. The aim was to be able to perform feature extraction and tracking on datasets larger than can be processed using a single processor. The work done in [3] also expanded this to handle AMR datasets and presented a way to do real-time tracking (doing feature extraction and tracking in-situ while the simulation is going on). The real-time tracking however requires an interactive steering environment to control the simulation. In this thesis, we extend the work done to provide a general-purpose way to perform feature extraction on large scale 3D datasets having number of nodes (datapoints) of the order of $10^8$ and above. With datasets of this size, it is not enough to simply divide the entire dataset among the processors available because the parts allotted are still too large to fit into the memory of a single processor. The method suggested does not require a specialized hardware or software and the code can be run on a modest sized cluster with 4-8 processors each having standard memory specifications easily available. The algorithm is further expanded to include the capability to run the code on a single machine without the need for a parallel cluster. An implementation using the method suggested was also developed in C/C++ and MPICH (a free portable implementation of the Message-Passing-Interface (MPI) standard). The implementation was tested with 3 large scale datasets. The results obtained are included in the next section.
5.2 Motivation

The distributed code for feature extraction works by loading parts of the dataset into numerous data-structures during the dataset reading process. For example, the node and cell structures are used to store all the node values and cell information respectively for the local portion of the dataset read by any of the processors on which the code is running. The number of nodes is given by:

\[ S_x \times S_y \times S_z \]

while the number of cells is given by:

\[ (S_x - 1) \times (S_y - 1) \times (S_z - 1) \]

where \( S_x, S_y, S_z \) are the dimensions along the x, y and z directions of a particular segment of the dataset handled by one processor. These values can increase to enormous amounts for very large datasets. Other data-structures used include a sorter class for cell sorting, an object class to store object information and a temporary queue for holding cell indices during the region-growing algorithm. Additionally, as the size of the dataset grows the size of the messages that must be passed between processors also increases. The overall result of all this is significant memory consumption. For large datasets, having number of nodes \( 10^8 \) or more, this can mean overloading or complete usage of the memory of that processor causing the running process to be killed or a segmentation fault. One way to assuage this memory consumption is to divide the dataset into a large number of processors such that no individual processor is overloaded to the extent that the process gets killed. This solution is useful to some extent, however it is not practicable cost or space wise to have such a considerable number of processors, nor does it make sense to harness the computing power of 16-32 processors to run this single code for one large dataset except maybe in exceptional cases. Therefore, an alternative way to resolve this problem is needed.
5.3 Divide and Conquer technique

One solution is to divide the large dataset into a number of parts. Each part is then run independently on the designated number of processors (usually 4-8). The number of parts into which the dataset is divided is decided based on the largest size of chunk (actually maximum number of nodes) that a single processor can manage without breaking down. Thus, this method combines the use of multiple processors working in parallel to complete a single part at the same time using a nominal number of processors to complete the overall task. This idea can be further extended to divide each part further and run the code in a distributed manner on a single processor.

5.4 Implementation of the method

In order to setup the existing feature extraction model to use this modified strategy, the code had to be modified in several places. Firstly, a way had to be devised to divide the dataset into parts. Then the code had to be changed to handle the dataset by parts instead of a single run. To avoid the tedious task of manually running the code multiple times for each part, the part-by-part execution was automated. Finally, the merging code had to be modified for handling the large number of files generated by the extraction step. Numerous optimizations added to the code to handle certain problems that arise when working on large datasets are also documented below:

- Calculation of number of parts and division of work
- Asynchronous messaging and extended partial merge method
- Automation of individual part running
- Modified final merging technique
  - Handling out-of-bounds problem
  - Handling single object multiple identity problem.
  - Other changes required for large dataset processing.
5.4.1 Calculation of number of parts and division of work

To implement the strategy above for handling large datasets, the dataset needs to be divided into a number of parts. The number of parts should be such that an individual part can be run on the available processors without any problems like memory overloading. Therefore the number of parts should be decided based upon a single processors' capability to successfully complete the task allotted to it. The calculations done were based on 8 processors of a cluster (the Frea cluster - TASSL lab, CAIP), each of which had a memory capacity of 1Gb of RAM and around twice that amount of swap space available. After a series of test runs, it was found that a single processor with these specifications could handle approximately 11010048 nodes at a time. Using this as a measure, a node_limit was defined which is the number of nodes that could be handled by 8 such processors working in parallel. The number of parts is then calculated as:

$$num_{parts} = \frac{(Sx \times Sy \times Sz)}{node\_limit}$$

This value is rounded up to the next higher integer.

After calculating the number of divisions, each processor on which the code is run needs to be assigned a portion of the task to complete. In the original partitioning algorithm [3] explained earlier, processors were assigned chunks of dataset with overlapping boundaries or ghost regions. Work was assigned by dividing each part of the dataset into chunks along one direction and passing the lower and upper dimension bound of that local portion to the processor designated to process it. However, the new partitioning scheme needs to take into account part boundaries as well as dataset boundaries. The partitioning paradigm is extended in the following way.

The dataset can be considered as one continuous chain of parts with intermediate boundaries where one part ends and the other begins. Therefore, considering each part being handled by 8 processors (the number of processors used can be defined by the user as a variable NUM_PROCS in advance), we can view the dataset as being processed by \( t \) processors overall where \( t = \text{NUM\_PROCS} \times \text{num\_parts} \), NUM_PROCS processors working on a part at a time. Using this each processor is now assigned an additional id number (new_procid) between 0 and \( t-1 \). This can be calculated easily knowing the
current part (part\_no) being processed and the actual processor id (proc\_Id, between 0 to 7 for 8 processors) as:

\[
new\_procid = proc\_Id + (NUM\_PROCS \times part\_no)
\]

Now, the task of assigning work is quite straightforward. A basic chunk size is calculated as:

\[
xlen = (Sx/t)
\]

rounded to the next higher integer because as we are partitioning by chunks in the x-direction. The lower and upper bound for each processor (of t processors) is now given as:

\[
\text{lower bound} = new\_procid \times xlen
\]

\[
\text{upper bound} = (new\_procid + 1) \times xlen
\]

except for the last processor in the last part where

\[
\text{upper bound} = Sx - 1
\]

as the dataset ends there.

After partitioning in this manner, any processor running this code for any part of the dataset can easily calculate the portion of dataset assigned to it and begin processing that part. Figure 5.1 shows the partitioning.
Figure 5.1: Illustration of modified dataset partitioning scheme
5.4.2 Non-blocking messaging and extended partial merge

The feature extraction algorithm implemented in [3] uses a partial merge algorithm for communicating boundary information between processors. In the first step, neighboring processors send messages to each other containing information about any objects shared in the common boundary between them. This was originally implemented using blocking send and receive directives in MPI. For a blocking send, the MPI send directive will not return till the send buffer can be reused (i.e., the message has been transferred to the receiver’s buffer) or until it finds a matching receive directive at the destination. Although it may be a safer practice to use this kind of send directive, for transferring large amounts of data (as in the case of a large number of shared objects). Another alternative is to use a non-blocking send - MPI_Isend that does not block and returns without requiring a matching receive at the destination. Usually this is used in conjunction with the non-blocking receive MPI_Irecv and MPI_Wait. MPI_Wait is used to query to see if a particular non-blocking operation has completed successfully, the status of the operation is returned in a flag variable. The use of non-blocking directives prevents any unnecessary hold ups in the running of the program by avoiding the sometimes unnecessary blocking for a corresponding receive at the destination. However, this puts the onus on the programmer to ensure that sends and receives are inserted correctly wherever required.

In the partial merge algorithm implemented for the feature extraction code, each processor (except for the processor 0) sends a message to the processor on its left, regarding information about any objects occupying cells on the boundary common to them. Based upon this information the receiving processor updates its local table of object connectivity. This algorithm needs to be extended to fit into the part-by-part processing setup. Now, in addition to the message passing between processors working on a certain part of the overall dataset, there has to be a way to pass boundary information between processors on either side of part-boundaries. This can be done as follows:

Each processor working on the left-most chunk of a part of the dataset creates a message similar to the message created in the usual partial merge. This message needs to be
transmitted to the processor sharing that part boundary with it, i.e., the processor working on the right-most chunk of the previous part. Note that in this case, we cannot use MPI to transfer messages between processors because individual parts are processed sequentially and not at the same time. For example, consider the diagram given, processor L2 (new_procid) does not know of the existence of processor R1, etc. So instead of passing the message outright, each such processor on the right side of a part boundary (L2, L3, etc) writes the message to a file. The contents of the file are exactly the contents of a message formed by any processor involved in the partial merge. For easy identification and processing, the file is named in the following fashion: right_side_proc to left_side_proc with the extension .msg. Files are written during the first step of the partial merge stage when processors are forming messages. When the processing of the next part to the left begins, the corresponding receiver processor reads this file and updates its local object id table with any new object connections as before. So processor L2 would write its message to a file called L2toR1.msg during processing of part 2 of the dataset. When part 1 is being processed processor R1 reads in this file and updates its table accordingly. Also, since messages are passed from right to left, it makes sense to process the dataset from its right-most part first to its left-most part. This ensures that there is always a message file generated ready to be read. Figure 5.3 shows the format of a .msg file and a local object id table formed.
Figure 5.2: Messaging between processors in adjacent parts
Figure 5.3: Format of (a) A .msg file and (b) A local object table
5.4.3 Automation of individual part running

After the first two modifications above, the code has now been modified to process the dataset by parts. An individual part can now be processed by invoking the `mpirun` command using `NUM_PROCS` number of processors with the additional command-line argument `part_no` at the end specifying which part is being processed. This parameter needs to be given to the program to calculate the `new_procids` of the processors for that part in order to do correct task-assignment as explained in the sections before this. Running the code part-wise this way is fine if there are only 1-2 parts to be run. However, this can be a tiresome task if the number of parts is large. Furthermore, while running the code this way, the user has to keep track of when a part has finished running to run the next part. The user also has to remember what the current `part_no` is. This could lead to user errors like running the same part twice or worse skipping a particular part altogether. Therefore, a scheme to automatically process each part one after another by calling the `mpirun` script on each part while keeping track of the current part number is desired.

The following scheme was designed using a Perl script to automate the part running. The calculation of number of parts is now done within the script. The Perl script now reads in information about the dataset, number of timesteps, etc. and issues the command for running the code on each part. This includes passing all required arguments, including, the current part number and dimensions to the C++ program. The sequence of steps executed by the Perl script are given below. These steps are also summarized in the flowchart shown.

1. Read in dataset name and no. of timesteps from file `vortsname`.
2. Read in dataset specifications from file `datainfo`.
3. Calculate `number_of_parts` for dataset.
4. Call C++ feature extraction program on each part.
5. Repeat steps 2 and 3 for every timestep to be tracked.
The advantage of this automation scheme is that the user is relieved from the need to know the exact syntax of running the code on each part. The user just ensures that the vortsname and datainfo files have accurate information about the dataset and then issues a single command to execute the Perl script. The Perl file intro_run.pl is included as part of the appendix section.
5.4.4 Modified final merging technique

The final merge is the second step of the partial merge algorithm [3]. As explained in the previous chapter, on completion of the first feature extraction step, a .table file is generated among others for each processor (of the total \( t \) processors) that contains the final local object table for that processor. These local tables contain information about all objects that lie on the boundary and need to be merged (because they are actually part of the same object). In this second step, a centralized visualization processor parses through the local object tables of all processors (the .table files) which took part in the feature extraction process and creates a global table of objects. This global object table is then used to create updated .poly and .dat files for the overall dataset.

Problems with the existing approach:

1. Out-of-bounds problem

   The problem:

   In the existing approach, the global object table is generated in two steps. During the first step, each of the local object tables in the .table files is parsed. For each row in a local table, a corresponding entry is made in a temporary list (implemented as array \( \text{obj}_\text{gid} \)) to indicate that the objects are linked. To indicate a link, the object id of the parent object is written to the location assigned for the child object in the \( \text{obj}_\text{gid} \) array. The type of linkage depends on whether the two objects have been linked to other objects previously or are new entities. Initially, the entire \( \text{obj}_\text{gid} \) array is initialized to -1. Hence, any object not previously seen will still have its assigned location in the array marked by -1. As already discussed, a row in an object table has the format \( \text{proc1 obj1 proc2 obj2} \) where proc1, proc2 and obj1, obj2 are the adjacent processor ids and corresponding local ids for the shared object respectively.

   - Case 1 - If both obj1 and obj2 are new entities (with -1 in \( \text{obj}_\text{gid} \)), then \( \text{obj}_\text{gid} \) is modified to indicate that obj2 is linked to parent obj1.
• Case 2 and Case 3 - If either obj1 is a new entity (Case 3) or obj2 (Case 2) but not both, then the object which has a -1 in \textit{obj\_gid} is declared as the child of the others’ parent object and a corresponding entry is made.

• Case 4 - If neither obj1 nor obj2 are new entities and each is linked to a different parent object, then a special case occurs (single object, multiple identity) which is discussed in the next section.

In this way, all possible linkages are recorded in the \textit{obj\_gid} table. It is possible for two objects of the same processor to be connected to each other through an object of the neighboring processor (combination of cases 1 and 3 or 2 and 3). During the second step, all objects of the dataset are considered one by one, and each one is assigned a final global id based on what linkages it has logged from the first step. Since we proceed sequentially in this step starting from object 0 of processor 0, object 1 of processor 0 and so on, a problem can arise if an object is dependent on another object of the same processor which is yet to be processed. Recollect that an object which is the parent object of a group of objects initially has ‘-1’ stored in its array location in the \textit{obj\_gid} array. For example, say an object of processor 1 with id 5 needs to be assigned the global id belonging to the object of the same processor with local id 14. Since 14 comes after 5 in the sequential order, it still has an unassigned global id (i.e. -1) and therefore the program tries to access an invalid location of the global ids array (\textit{gid }[-1 * MAXMERGE+0]) to fetch a global id for object 5. The location is invalid since it is an address outside of allocated memory bounds. Due to memory location unpredictability, this might not cause any problems immediately but eventually causes the program to crash due to Segmentation Fault error.

Solution:
We can resolve the out-of-bounds problem as follows. A temporary list is created before the global id assignment begins. As we go through the sequential assignment of global ids to each object in the dataset, if we encounter any object which has to be assigned the global id of an object yet to come (i.e., easily identified
since the global id of the still to come object will still be -1), then that object is pushed into the temporary list. After we finish the processing of all possible objects (except for the ones pushed into the list) for one file, we begin processing of the objects pushed into the list. The algorithm we follow is:

(a) For each object 'O' in the list :

(b) Attempt to assign a global id as before.

(c) If the global id is assigned correctly (i.e., is not -1) then remove that object from the list.

(d) Else, keep the object in the list for the next round of processing.

(e) Repeat steps 1, 2 and 3 until the list is empty.

The corresponding flow diagram is shown.

The reason for repeating the procedure multiple times is that an object in the list might be dependent on another object lower in the list and might not get removed during the first run. Similarly, an object lower down the list might need to be assigned the global id of an object above it. Therefore, it is not enough to simply run through the list once either forward or backward. Once the list is empty, all objects of that processor have been successfully assigned a global id and we can continue processing objects of the next processor. In this way we avoid the accessing of invalid memory locations and prevent the code from prematurely crashing.
Figure 5.5: List processing for out-of-bounds error
2. **Single object multiple identity problem**

The problem: In the original implementation of the feature extraction code and our modified version for large scale datasets, the dataset is divided into parts. Each part is further divided into chunks along the x-dimension. For some datasets having a number of complex objects split between adjacent processors, the following problem can arise. Consider the diagram below showing an object overlapping between two adjacent processors.

![Diagram of complex object overlap between processors](image)

**Figure 5.6: Illustration of complex object overlap between processors**

The original object has a 'G' shape and is shared between processors 0 and processor 1 which have a common boundary. Due to the nature of the partition, the original object gets divided into 4 sub-objects two parts each in the local regions allotted to processor 0 (named object 0 and object 1) and processor 1(named object 2 and object 3) respectively. The part of the original object that lies in processor 2’s local region is shown hashed. Considering this split, the entries in processor 1’s local object table will reflect the following relationships:

- \(0 \rightarrow 2\) (object 2 is part of object 0)
- \(1 \rightarrow 3\) (object 3 is part of object 1)
- \(0 \rightarrow 3\) (object 3 is part of object 0). Thus, it seems ambiguous whether object 3 should be assigned the global id for object 0 or object 1. However we know that, in reality both object 0 and object 1 are also the parts of the same object. This
error is possible due to the partitioning method used which results in the dataset being divided along a single dimension into narrow strips. One possible solution would be to divide the dataset along multiple dimensions into blocks instead of chunks; this may reduce the number of such irregular divisions.

However, we need to formulate a method that will take care of such irregular partitions which may occur because this can result in an erroneous number of final objects being extracted from the dataset.

Solution:

Looking at the example above, we know that in such cases all the 4 parts are members of a single object and should be labeled as such. That is, all 4 objects should finally be assigned the same global id for the dataset. Also, of the 4 objects, object 0 and object 2 have already been identified as being part of a single object. What remains is to label objects 1 and 3 as being part of that same original object.

In the existing implementation of the code, there was no provision for such cases (Case 4 as discussed previously) and only an error message was displayed for each such ambiguous case.

Since we already have a running list of what objects are to be merged with each other in the obj.gid array, a simple approach to solve the problem could be to parse through all the entries in the array. When the ambiguous entry occurs, we modify the values of parent object for all objects with the incorrect value to the new value. However, this involves sequential array traversal and for large scale datasets with a large number of entries to be parsed this would not be an efficient method. To simplify this we do some book-keeping by creating a link table. As we process each entry in the local object tables for the first time, this link table is filled in. It stores a record of all objects that are to be merged together in separate rows of the table. A new row is added to the table each time a new parent object is discovered (an object which has a '-1' in its obj.gid entry and has not been entered into the table up to then). For other objects, a link is added to the row of the parent object they are linked to. For example, consider a dataset split into two processors 0 and 1 with 7 objects in each. Processor 0 has the
following entries in its local object table:
0 1 1 1
0 3 1 1
0 3 1 4
0 2 1 5
0 7 1 7
These entries would result in the following link table:
1 ⇐ 8 ⇐ 3 ⇐ 11
2 ⇐ 12
7 ⇐ 14
(Note that actual ids are stored in the table, for this example object 0 3 has actual id 0+3=3 while object 1 1 has actual id 7+1=8 since there are 7 objects in processor 0’s local region). Now, if we have an additional entry 0 3 1 5, this would cause the same ambiguity explained above. Object 1 5 (i.e. 12) has been linked earlier with object 2. The new entry indicates a link with object 3 i.e. effectively with object 1 (since object 1 is overall parent). We know that therefore, objects 1 and 2 are part of the same object thus; rows 1 and 2 of the link table need to be combined. Further, it makes logical sense to assign the parent object with the lower of the two ids to be the new parent for the combined group. In our example, this results in the following final link table:
1 ⇐ 8 ⇐ 3 ⇐ 11 ⇐ 2 ⇐ 12
7 ⇐ 14
It is now a relatively simple task to reflect these changes in the obj_gid array by simply accessing those locations corresponding to objects affected by the row merge in the relevant row above and modifying them. Thus, we only have to modify the comparatively fewer objects locations indicated by the link table instead of inefficient serial array traversal. This is at the small cost of storing the link table structure in memory. Therefore, using the link table structure and the logic explained we can resolve the problem of a single object being identified as multiple ones. The technique suggested has been implemented and tested for a
smaller dataset (soil15.raw). For more complex objects, other approaches may be required.

5.4.5 Other changes required for large dataset processing

All the major modifications and additions that were made to the final merge code have been explained in preceding sections. There are a number of small changes that were needed to handle the processing of large scale datasets. Since the number of objects in such large scale datasets is huge which in turn also means that the number of objects being merged is also possibly quite large, some constants and array sizes that were earlier defined keeping in mind standard datasets need to be modified. Most of these changes were made in the header file associated with the final merge code. Additionally, since the number of objects is very large, the arrays used for storing and processing them (obj_gid, gid, etc) consume a lot of memory space. Consequently it becomes necessary and is more efficient to allocate memory for these arrays dynamically from memory in the 'free store' and deallocate them after processing is complete.

In the final merging step, after a global list of object ids has been formed, it is used to update .poly and .oct, etc. files for the complete dataset. Owing to the large number of objects in the dataset, these files especially the .poly file (containing node co-ordinate and polygon information for each object) can become very large occupying several gigabytes of disk space. Usually, as a safety precaution the file management system of the operating system limits the size of files that can be written to a default upper limit. We need to adjust this limit to be able to write such large files. For example, on the Linux OS (Red Hat 8.0 for our test case), the command `limit` (on tcsh shell) can be used to view and change the limits set by the OS for `filesize` and other parameters.

When opening and writing large files using standard C/C++ commands like `fopen` on 32-bit systems, we need to compile our programs with Large File Support (LFS). If this is not done, the program will not be able to perform open/read/write operations on the large .poly files generated. To enable LFS to the code, we include the following lines to the code before any other files are included:
#define _LARGEFILE_SOURCE
#define _FILE_OFFSET_BITS 64

This transparently replaces all relevant functions and data structures used in the code with their appropriate 64-bit versions (fopen64(), etc.) allowing for I/O operations on larger files.

**Separation of .poly files and largest feature extraction:** As noted above, due to the large number of features involved, the files generated especially the .poly file are large (around 2-4Gb on disk). This makes the files difficult to read or transfer quickly for easy access. Typically, we do not need the entire .poly file all at once unless we are able to view the entire dataset iso-surfaces. Hence, the option has been provided to separate the .poly files while writing. Thus, we can have a single .poly file for every extracted object of the dataset. This makes it easy to view the iso-surface of any object we desire.

In addition, the final merging program has also been modified to extract the largest object (largest iso-surface). This is then written to a separate .poly file.

### 5.4.6 Modifications for single machine running

The sections above describe the changes that were made to the existing code in order to handle large scale data when running the code on a cluster of processors. The same paradigm can be further extended to run the code on a single machine. This means that all the feature extraction of objects and subsequent merging will now be done on a single machine. Such a feature is useful for processing large scale datasets in the absence of a processor cluster. The general idea is to divide each of the parts (that the dataset is partitioned into) into smaller units that can be handled by a single processor.

- The same processor will run the code multiple times till the entire dataset has been processed. As in the case of the part-wise running method, during each run the processor will use the input parameters - part_no, division_no (a number between 0 to 7 for each part), and local processor id (always 0 for single machine run) to calculate the bounds for the chunk of dataset to be processed in that step.
• The extended partial merging algorithm developed is used. Instead of only at chunk boundaries, all message-passing will be implemented by file I/O. During each run of the code (except for the last run), the single processor writes a message file and reads the file written during the previous run. In this way, local object tables are updated as before.

• Since, the number of files generated during the first step is the same as in the case of the cluster run code, the final merging code remains the same. For example, a large scale dataset that divides into 3 parts and is run on a cluster using 8 processors will generate 24 .table files. The same dataset when run using the single machine version of the code will run 24 times to generate the same number of files as before.

![Figure 5.7: Single machine running for the entire dataset](image)
An implementation of the steps noted above was done. The same large-scale datasets were tested on the single machine version of the code and the results verified with the ones generated previously. A user can select to run the code on a single machine by setting the appropriate parameter ($\text{single\_mach}=1$) in the Perl script file intro\_run.pl.

### 5.5 Running the large scale feature extraction code

The steps for running the modified feature extraction code for large scale datasets are:

1. Modify the $\text{vortsname}$ and $\text{datainfo}$ files with the dataset and time-step information.

2. Modify the Perl file:
   - Define the value of $\text{single\_mach}$ (=1 for single machine run,=0 otherwise)
   - Define the value of $\text{numprocs}$ with the number of processors desired.
   - Define the value of $\text{Mfile}$ with the appropriate machine file for the cluster.
   - Define the value of $\text{node\_limit}$ to according to processor memory constraints.

3. Run the feature extraction step with command: `perl intro\_run.pl`.
   This will perform feature extraction on the input data either part-wise or chunk-wise depending on whether cluster run or single machine run is specified as above.
   Individual processor files will be generated in the ./result folder.

4. Run the merging code with the command: `./merge ./result/<file\_base\_name> <num\_files>`
   where num\_files is the number of .table files generated in the first step. The result folder will now contain the final merged result files.
   Note: The extraction and merge codes are to be compiled in the same way as the original distributed code.
Chapter 6
Results and Discussion

The extended algorithm for large scale feature extraction explained in Chapter 5 was implemented and tested on 3 large scale datasets. The following datasets were obtained courtesy the Department of Environmental Sciences at Rutgers University (courtesy Prof. Daniel Gimenez, Soil Science/Soil Physics).

- Soil dataset 1 of dimensions 900 X 800 X 400 (soil2.raw)
- Soil dataset 2 of dimensions 1797 X 1588 X 168 (soil11.raw)
- Soil dataset 3 of dimensions 1000 X 1000 X 200 (soil3.raw)

Studies have found that elevated atmospheric CO2 decreases aggregate sizes and pore neck diameters of soil structure. These studies implied that the change in soil structure could be an important feedback to study the effect of climate change on infiltration rate, soil temperature, and soil water content. This has led to an investigation of the soil structures under different climate changes. To obtain the 3D datasets, soil samples were first taken at selected sites along a CO2/temperature transect. The soil sample size was 5.5cm in diameter and 12cm in height. Axial x-ray computer tomography (microCT; model MS, General Electric Medical Systems) was used to obtain 3D images with resolution of 2000 by 1500 by 1000 pixels (4.5 by 3.7 by 2.5 cm). The images were processed in Microview (GE) to reduce their size. Using an Image J (public domain Java-based image processing program developed at the National Institutes of Health) plug-in, each 3D image was converted into a binary image using thresholding. The aim was to obtain information about the size, shape and surface areas of these soil pores. Feature extraction can be used to obtain this analysis. The datasets have already been thresholded and only contain zeros and ones now.
All 3 datasets were processed using the large scale feature extraction algorithm. Due to their size, it is not possible to use the standard uniprocessor algorithm. Also, given the memory constraints on standard processors, we cannot directly run the original distributed code [3] on them. Hence, it is necessary to process them using the previously explained modified algorithm. On running the modified code on these datasets, they are processed by dividing them into 4 parts, 6 parts and 3 parts for each of the datasets in the list shown above.

The following figures show a volume rendered view of the 3 datasets tested using the Volview [Kitware] visualization software.

Figure 6.1: soil2.raw
Figure 6.2: soil11.raw

Figure 6.3: soil3.raw
Owing to the size of these large scale datasets and the enormous number of objects that need to be extracted, the time taken to perform feature extraction on them is considerably more as compared to an average size dataset. Table 6.1 gives information about the time taken for processing their individual parts and for the final merging and file updating.

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Part No.</th>
<th>Communication Reading Data</th>
<th>Processing (Segmentation+ Polygon Gen)</th>
<th>Global Merge</th>
<th>File Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>soil11</td>
<td>5</td>
<td>34.98 secs</td>
<td>309.69 secs</td>
<td>16.45 secs</td>
<td>67.2s</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.13 secs</td>
<td>284.52 secs</td>
<td>11.82 secs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10.75 secs</td>
<td>260.79 secs</td>
<td>14.12 secs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.778 secs</td>
<td>252.29 secs</td>
<td>13.53 secs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>12.83 secs</td>
<td>331.84 secs</td>
<td>14.46 secs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>9.32 secs</td>
<td>319.40 secs</td>
<td>14.11 secs</td>
<td></td>
</tr>
<tr>
<td>soil2</td>
<td>3</td>
<td>17.29 secs</td>
<td>112.96 secs</td>
<td>14.26 secs</td>
<td>57.8s</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>56.11 secs</td>
<td>202.19 secs</td>
<td>24.67 secs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7.97 secs</td>
<td>202.22 secs</td>
<td>13.25 secs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2.225 secs</td>
<td>202.07 secs</td>
<td>11.19 secs</td>
<td></td>
</tr>
<tr>
<td>soil3</td>
<td>2</td>
<td>28.62 secs</td>
<td>140.53 secs</td>
<td>13.91 secs</td>
<td>46.7s</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>26.29 secs</td>
<td>140.17 secs</td>
<td>12.94 secs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>58.78 secs</td>
<td>140.20 secs</td>
<td>21.92 secs</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Time taken for large scale feature extraction

It is seen that most of the processing time is spent on reading data from the disk and the updating of the final files. The figures that follow show some of the partial results obtained by individual processors while performing feature extraction on each of the test datasets.

The figures show the locally extracted data for one processor from each of the parts that the datasets are split into. These partial results are merged during the final merging step to obtain the overall extraction results on the entire dataset.
Figure 6.4: Partial extraction results of soil2.raw for processor0, 2-D and 3-D

Figure 6.5: Partial extraction results of soil2.raw for processor8, 2-D and 3-D
Figure 6.6: Partial extraction results of soil2.raw for processor16, 2-D and 3-D

Figure 6.7: Partial extraction results of soil2.raw for processor24, 2-D and 3-D
Figure 6.8: Partial extraction results of soil11.raw for processor0, 2D and 3D

Figure 6.9: Partial extraction results of soil11.raw for processor8, 2D and 3D
Figure 6.10: Partial extraction results of soil11.raw for processor16, 2D and 3D

Figure 6.11: Partial extraction results of soil11.raw for processor24, 2D and 3D
Figure 6.12: Partial extraction results of soil11.raw for processor32, 2D and 3D

Figure 6.13: Partial extraction results of soil11.raw for processor40, 2D and 3D
Figure 6.14: Partial extraction results of soil3.raw for processor0, 2D and 3D

Figure 6.15: Partial extraction results of soil3.raw for processor8, 2D and 3D
Figure 6.16: Partial extraction results of soil3.raw for processor 16, 2D and 3D
In order to gauge the relative size of an extracted object, the largest object from each dataset have been extracted. The following figure shows the largest iso-surface objects extracted from each of the soil datasets on which feature extraction was performed.

![Figure 6.17: Largest extracted feature from the soil 2 dataset in 2D and 3D](image)

Table 6.2 summarizes the results obtained indicating the number of objects extracted from individual processors and the overall number of objects obtained at the end after all partial results have been merged.

The merging algorithm is used to resolve all feature connectivity between processors for the same time-step so that features that are essentially a single entity are not identified as multiple objects. It is seen that on an average, 20-25% of the original number of features extracted are merged during the global merge process.
Figure 6.18: Largest extracted feature from the soil 3 dataset in 2D and 3D

Figure 6.19: Largest extracted feature from the soil 11 dataset in 2D and 3D
Figure 6.20: Largest extracted feature from the soil 15 dataset in 2D and 3D

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Part No.</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>Final Object Count after Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>soil1</td>
<td>5</td>
<td>376</td>
<td>401</td>
<td>475</td>
<td>471</td>
<td>464</td>
<td>479</td>
<td>409</td>
<td>211</td>
<td>13127</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>335</td>
<td>383</td>
<td>337</td>
<td>421</td>
<td>341</td>
<td>432</td>
<td>328</td>
<td>297</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>379</td>
<td>320</td>
<td>326</td>
<td>312</td>
<td>342</td>
<td>339</td>
<td>318</td>
<td>306</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>343</td>
<td>339</td>
<td>378</td>
<td>419</td>
<td>397</td>
<td>429</td>
<td>405</td>
<td>406</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>299</td>
<td>331</td>
<td>405</td>
<td>446</td>
<td>396</td>
<td>354</td>
<td>370</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>347</td>
<td>322</td>
<td>338</td>
<td>329</td>
<td>356</td>
<td>399</td>
<td>363</td>
<td>331</td>
<td></td>
</tr>
<tr>
<td>soil2</td>
<td>3</td>
<td>560</td>
<td>562</td>
<td>572</td>
<td>574</td>
<td>600</td>
<td>596</td>
<td>615</td>
<td>633</td>
<td>15547</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>626</td>
<td>675</td>
<td>577</td>
<td>540</td>
<td>473</td>
<td>543</td>
<td>561</td>
<td>561</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>586</td>
<td>551</td>
<td>587</td>
<td>508</td>
<td>587</td>
<td>653</td>
<td>594</td>
<td>587</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>735</td>
<td>713</td>
<td>748</td>
<td>702</td>
<td>810</td>
<td>703</td>
<td>698</td>
<td>562</td>
<td></td>
</tr>
<tr>
<td>soil3</td>
<td>2</td>
<td>602</td>
<td>641</td>
<td>601</td>
<td>560</td>
<td>498</td>
<td>494</td>
<td>502</td>
<td>548</td>
<td>12351</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>647</td>
<td>722</td>
<td>668</td>
<td>726</td>
<td>658</td>
<td>589</td>
<td>584</td>
<td>561</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>735</td>
<td>846</td>
<td>906</td>
<td>867</td>
<td>740</td>
<td>774</td>
<td>776</td>
<td>664</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Feature count before and after global merging
Given the large dimensions of the datasets, the corresponding .poly files obtained from their processing pose a problem in being loaded into memory (on a standard machine with 512 Mb RAM). Therefore, it is not possible to display an iso-surface visualization for the complete dataset. However, in order to completely demonstrate the utility of the algorithm implemented, a smaller dataset soil15.raw of dimensions 256 X 128 X 128 (obtained from soil11) was processed and the results obtained for

- Individual processors
- Each part
- Overall dataset

are shown. Note that these results were obtained by lowering the node limit to a suitable value in order to spread the processing of the dataset over multiple parts (3 parts).

Figure 6.21: Partial extraction results for part0 of soil15.raw
Figure 6.22: Partial extraction results for part 1 of soil15.raw

Figure 6.23: Partial extraction results for part 2 of soil15.raw
Figure 6.24: Partial extraction results for part 2 of soil15.raw on processors 0, 8, 16 in 3D
The final extracted output obtained after merging the partial results of Part 0, Part 1, and Part 2 of soil15.raw is shown. In this way, a large scale dataset can be feature extracted using a 'divide and conquer' strategy. The figure also shows an approximate split of the extracted dataset based upon the partial results displayed earlier.

Figure 6.25: (a) Illustration of part-wise division and (b) Final merged result for soil15.raw

Figure 6.26 below shows two different views of the iso-surface visualization of the final merged result for the soil15.raw dataset.
Figure 6.26: Two views of the iso-surface visualization for soil15.raw
In order to corroborate the results obtained, the same soil dataset soil15.raw was also processed using the AVS based uniprocessor feature extraction code. The results obtained from the AVS code were compared with the merged result obtained from the multi-pass large scale extraction code. The comparison shows that the final result from the large scale feature extraction code is an almost exact match to the result obtained from the AVS code thus verifying the algorithm. Figure 6.27 shows the iso-surface visualizations of the two results under comparison. Note that the difference in color schemes is due to the fact that a random coloring scheme is applied to each object in the final extracted results before the .poly file is updated.

Although the results obtained are very similar, as indicated in the figure above there is a slight discrepancy in the number of objects extracted by the two methods under comparison. This discrepancy is currently under investigation.

Figure 6.27: Comparison of AVS based feature extraction and Large scale feature extraction results
The AVS based network used to obtain the visualizations in this section is given below.

Figure 6.28: Visualization network used to view partial and complete results
Chapter 7

Conclusion and Future Work

The need for effective analysis and visualization of increasingly large time-varying data has necessitated the use of a distributed approach for feature extraction and tracking. Further, considering realistic processor configurations, it is not always possible to apply this distributed approach directly for large scale datasets having 100 million or more data points.

In this thesis, I extended previous work done on distributed feature extraction and tracking to provide a general-purpose technique for feature extraction of large scale datasets using either a cluster of processors or a single processor. The method suggested does not require any special software or exclusive hardware and can be easily ported from one system to another. An implementation of the algorithm presented was also done and tested on several large datasets. The results provided demonstrate the utility of the algorithm. A number of enhancements to improve the functionality of the basic stand-alone feature extraction and tracking software are also discussed.

Currently the partitioning of the large scale dataset is done only along a single dimension. This can be adapted to partition into blocks instead of strips to and tested to see if the results obtained are better. The extended partial merging method developed could be modified and applied to the distributed feature tracking technique to perform feature tracking of large scale datasets effectively. Although, a number of coloring schemes to aid visual tracking were suggested, a way to interactively color the original dataset based on the region of interest desired would be useful to implement. This would give the user even more control over the visualization of desired features and be a step towards developing the next generation of feature tracking systems.
Appendix A

A.1 Installation instructions for Ostrk2.0

The Ostrk2.0 package is available as a standard IAC package that is easily imported into AVS/Express (For more information refer to www.iavsc.org). The following are the steps to install the Ostrk2.0 package including any dependencies it relies on.

- Download the latest Boost library from www.boost.org. You will need to uncompress the library files to a suitable directory and create a link to this directory in order for your AVS code to compile successfully.
- Create a directory called IAC.
- Download, gunzip and untar the iac.tar.gz (from the IAC website) into the IAC directory. This will create an iac_proj sub directory and other necessary files.
- Download and extract the ostrk2.tar.gz file in the IAC directory.
- Run the install.sh script in the IAC directory.
- In the file ./include/xp_defs.mk, add -I boost library path to the XP_INC_OPTS line. If this is not done, the code will not compile correctly.
- Compile the code using the command 'base -compproc express -exit'.
- Start the AVS/Express application by typing 'bin/linux/express' at the prompt.

A.2 Output file formats for Ostrk2.0 package

The following files are generated by the Ostrk2.0 package [9].
• .poly file - This file contains information for Iso-surface visualization like Node co-ordinates and polygon vertices.

• .attr file - This file contains information about every object in a particular time-step. This includes centroid, moments, max node value, etc..

• .uocd file - This file contains information pertaining to the frame. Some of the main points are No. of objects, Cell information.

• .trak file - This file contains information about Number of Objects and Number of Nodes. Based on this, memory is allocated for the frame.

• .trakTable file - This file contains the results of tracking. It gives the evolution history of each object frame by frame along with an indication of any events that occur.

• .addl file - This file contains additional attribute information - Surface Area for all objects with the min. and max values for the frame.

A description of their format follows:

A.2.1 The .poly file

Format:
[red] [green] [blue]
[num_of_nodes]
[x0] [y0] [z0]
[x1] [y1] [z1]
[x2] [y2] [z2]
[x3] [y3] [z3]
[x4] [y4] [z4]
.
.
.
[num_of_connections]
Sample :

0 158 96

6774

67.234009 102.000000 112.000000
68.000000 102.000000 111.367813
68.000000 101.106598 112.000000

13488

3 1 3 2
3 4 6 5
3 7 9 8

0
A.2.2 The .attr file

Format:

doject [objID] attributes:
Max position: ([maxX], [maxY], [maxZ]) with value: [max_nodevalue]
Node #: [num_node_min]\nMin position: ([minX], [minY], [minZ]) with value: [min_nodevalue]
Node #: [num_node_max]
Integrated content: [integrated_content]
Sum of squared content values: [sum_squared_content_value]
Volume: [volume]
Centroid: ([x], [y], [z])
Moment: Ixx = [Ixx]
Iyy = [Iyy]
Izz = [Izz]
Ixy = [Ixy]
Iyz = [Iyz]
Izx = [Izx]

Sample:
doject 0 attributes:
Max position: (63.000000, 56.000000, 127.000000) with value: 12.139389
Node #: 2087999
Min position: (50.000000, 66.000000, 117.000000) with value: 6.073445
Node #: 1925426
Integrated content: 52753.316406
Sum of squared content values: 420312.062500
Volume: 6857
Centroid: (54.960667, 66.905724, 118.313576)
Moment: Ixx = 81.173683
Iyy = 175.838699
Izz = 76.425446
Ixy = -105.836067
Iyz = -90.048668
Izx = 46.851215

------------------------------------------

object 1 attributes:
Max position: (64.000000, 55.000000, 0.000000) with value: 11.885983
Node #: 7104
Min position: (56.000000, 69.000000, 0.000000) with value: 6.073481

A.2.3 The .uocd file

Format:
	[time]
	[num_objects]
	[objID] // starts from 0
	[volume] [mass] [centroidX] [centroidY] [centroidZ] // volume is the points number
	[Ixx] [Iyy] [Izz] [Ixy] [Iyz] [Izx]
	[pointID] [pointX] [pointY] [pointZ] [point_value] // pointID starts from 0
	[pointID] [pointX] [pointY] [pointZ] [point_value]
	[pointID] [pointX] [pointY] [pointZ] [point_value]
	[pointID] [pointX] [pointY] [pointZ] [point_value]
	[pointID] [pointX] [pointY] [pointZ] [point_value]

...
Sample:
10.000000
38
0
12637 83432.195312 68.521065 97.046165 99.475235
234.301193 51.264282 245.409286 -85.529694 79.158249 -222.195129
1798593 65.000000 99.000000 109.000000 10.520865
1798721 65.000000 100.000000 109.000000 10.489776
1798722 66.000000 100.000000 109.000000 10.289307
.
.
.
1
6316 42605.246094 62.313812 117.506172 39.228920
39.796001 66.488274 73.155128 -18.061323 55.640301 -30.467131
703935 63.000000 123.000000 42.000000 10.001476
704063 63.000000 124.000000 42.000000 9.899324
704064 64.000000 124.000000 42.000000 9.899070

A.2.4 The .trak file

Format:
[file_basename] [time] [mass] [volume] [centroidX] [centroidY] [centroidZ]
[file_basename] [time] [mass] [volume] [centroidX] [centroidY] [centroidZ]
[file_basename] [time] [mass] [volume] [centroidX] [centroidY] [centroidZ]
[file_basename] [time] [mass] [volume] [centroidX] [centroidY] [centroidZ]
.
.
.
Sample:

/home/lianjian/avsexp/vortsfld/GENERATED_TRACK_FILES/vorts10 10.000000
83432.195312 12637 68.521065 97.046165 99.475235
/home/lianjian/avsexp/vortsfld/GENERATED_TRACK_FILES/vorts10 10.000000
42605.246094 6316 62.313812 117.506172 39.228920
/home/lianjian/avsexp/vortsfld/GENERATED_TRACK_FILES/vorts10 10.000000
40073.558594 5771 62.039474 46.921017 4.576792
/home/lianjian/avsexp/vortsfld/GENERATED_TRACK_FILES/vorts10 10.000000
109392.312500 16884 93.668022 26.429691 59.045506
/home/lianjian/avsexp/vortsfld/GENERATED_TRACK_FILES/vorts10 10.000000
28768.537109 4321 114.335655 69.195396 89.172546

A.2.5 The .trakTable file

In this file '-1' is used as delimiter to indicate an event
Format:
frame #2
11 -1 13 26 // obj 11 of frame1 splits into 13 and 26 in frame2
19 21 -1 17 // obj 19, 21 of frame1 merge into obj 17 in frame2
1 -1 1 // obj 1 of frame1 continues as 1 in frame2
2 -1 2
.
.
.
-1 23 // new born obj 23 in frame 2
9 -1 // obj 9 of frame 1 disappears in frame2
11 -1
Frame #3
A.2.6 The .addl file

This file is generated as part of the Enhanced Ostrk2.0 package discussed at the end of Chapter 3.

Format:

Object ID: [objID] Surface Area: [area]

Max Volume: [max_volume] (Obj ID: [max_vol_obj])
Min Volume: [min_volume] (Obj ID: [min_vol_obj])
Max Area: [max_area] (Obj ID: [max_area_obj])
Min Area: [min_area] (Obj ID: [min_area_obj])

Sample:
Object ID:0 Surface Area: 352.19
Object ID:1 Surface Area: 2053.54
Object ID:2 Surface Area: 6049.51
Object ID:3 Surface Area: 1763.81
Object ID:4 Surface Area: 1299.46
Object ID:5 Surface Area: 1818.71
Object ID:6 Surface Area: 1693.24
Object ID:7 Surface Area: 1729.64
Object ID: 8 Surface Area: 1863.88
Object ID: 9 Surface Area: 1827.8
Object ID: 10 Surface Area: 440.099
Object ID: 11 Surface Area: 1427.66
Object ID: 12 Surface Area: 1537.67
Object ID: 13 Surface Area: 991.861
Object ID: 14 Surface Area: 1373.97
Object ID: 15 Surface Area: 801.772
Object ID: 16 Surface Area: 314.809
Object ID: 17 Surface Area: 942.163
Object ID: 18 Surface Area: 893.915

Max Volume: 15342 (Obj ID: 2)
Min Volume: 6 (Obj ID: 34)
Max Area: 6049.51 (Obj ID: 2)
Min Area: 2.61006 (Obj ID: 34)
Appendix B

B.1 Perl script for automated running of feature extraction by parts

#!/usr/bin/perl5.8.5

#--script to run Feature Extraction Code
# Pinakin Dhume 08/02/07
# Description:
# The script reads in information about number of files from vortsinfo file
# The script reads in information about each timestep from the datainfo file
# For each timestep it calculates number of parts based on preset node limit
# It sequentially runs each part to generate the result files
#--

#use Net::Ping;
use Shell;

# define constants here
$single_mach=0; #defines whether the extraction program will run on
               #single processor or a cluster
#For single processor running, each part is further divided
#into 8 divisions as before,and a single processor does them sequentially
#default : 0 - cluster, 1 - single machine
$numprocs = 8;
$node_limit = 1024*512*168;#128*128*128; #Set according to $numprocs value
# $file = "/caip/u61/pinakin/tw_info.dat";
$file = "tw2";
$div_num = -1; # will be modified later for single_mach runs

print "Feature extraction code started ... !\n";

# Read vortsname file
open(VORTS,"vortsname") or die "file vortsname does not exist !";
$num_files = <VORTS>;
print "Number of timesteps : $num_files";
for($i=0;$i<$num_files;$i++) {
    $file = <VORTS>;
    chop($file);
    print "\nFilename is : $file";

# Now read datainfo file
open(DATAINFO,"datainfo") or die "file datainfo does not exist !";
@file_data= <DATAINFO>;
close DATAINFO;

#extract ndims
$ndim = $file_data[0];
chop($ndim);
($label,$ndim)=split(/\=/,$ndim);
if($ndim==3){
    print "\nNo. of dimensions :$ndim";
} else{
    print "\nNo. of dimensions should be 3 !";
    exit(0);
$NX = $file_data[1];
chop($NX);
($label,$NX)=split(/\=/,$NX);
print "\nX-dimension :$NX";

$NY = $file_data[2];
chop($NY);
($label,$NY)=split(/\=/,$NY);
print "\nY-dimension :$NY";

$NZ = $file_data[3];
chop($NZ);
($label,$NZ)=split(/\=/,$NZ);
print "\nZ-dimension :$NZ";

$data = $file_data[4];
chop($data);
($label,$data)=split(/\=/,$data);
print "\nData path :$data";
$fdata=$data.$file;
print "\n Complete dataname :$fdata";

#extract basename

$base = $file_data[5];
chop($base);
($label,$base)=split(/\=/,$base);
print "\nBasename :$base"
($Name,$ext)=split(/\./,$file);
$fbase=$base.$Name;
print "\n Complete basename :$fbase"

#$Name contains substring

#extract time
$time = $file_data[6];
chop($time);
($label,$time)=split(/\=/,$time);
print "\nData time :$time"

#extract thresh
$thresh = $file_data[7];
chop($thresh);
($label,$thresh)=split(/\=/,$thresh);
print "\nThreshold :$thresh"

#extract rule
$rule = $file_data[8];
chop($rule);
($label,$rule)=split(/\=/,$rule);
print "\nRule exists :$rule"

# Calculate number of parts
$num_parts = int(((NX*$NY*$NZ)/$node_limit)+1);
print "\nNo. of parts : $num_parts";

#Parameters to be passed
#------------------------
# $part - part number
# $Name - name without extension
# $ndim - No. of dims
# $NX - x dim
# $NY - y dim
# $NZ - z dim
# $fdata - full datafile name with path
# $fbase - full basename with path
# $time - time value
# $thresh - threshold
# $rule - Rule value
# $num_parts - No. of parts
# $div_num - Division No. - =-1 for cluster,
# = $pcount for single_mach run
#------------------------

#Now call main program passing all parameters
$temp=0;
$part=$num_parts-1;
# $part=0; #test
#$sh=Shell->new();

while($part>=0)
{
    if($single_mach==0)
    {

print "\nCluster Run Mode";
$out = $Name.$part;
print "\n PART $part BEGINS";
$div_num=-1;
#@args=("/usr/local/mpich-1.2.7p1/bin/mpirun -np 8 -machinefile /caip/u61/pinakin/tw_nodes.txt object","2");
system("/usr/local/mpich-1.2.7p1/bin/mpirun -np $numprocs -machinefile $Mfile object $part $Name $ndim $NX $NY $NZ $fdata $fbase $time $thresh $rule $num_parts $div_num") == (0 or $?);
#system("/usr/local/mpich-1.2.7p1/bin/mpirun -np $numprocs object $part $Name $ndim $NX $NY $NZ $fdata $fbase $time $thresh $rule $num_parts") == (0 or $?);
if($? == -1){
  print "failed to execute";
}
else{
  print "exited with value $? ";
}

#print "\nEND OF PART $part";
}
if($single_mach==1)
{
print "\nSingle Proc Run Mode";
$pcount=7;
    $out=$Name.$part;
    $out=$out.$pcount;
$div_num =$pcount;
print "\n PART $part BEGINS";
while($div_num>=0)
{ 
print "\n DIVISION $div_num BEGINS";
system("./object $part $Name $ndim $NX $NY $NZ
 $fdata $fbase $time $thresh $rule $num_parts $div_num") == (0 or $?);
if($? == -1){
print "failed to execute";
}
else{
print "exited with value $? ";
}
print "\n DIVISION $div_num ENDS";
$div_num--;
}

}

print "\nEND OF PART $part"
part--;

}

print "END OF FILE :$i";
}

print "ALL FILES DONE \n FEATURE EXTRACTION COMPLETE"

close VORTS;
Appendix C

C.1 Distributed Feature Extraction and Tracking Input File Formats

- vortsname - i/p datasets info.
- datainfo - timestep info.
- datafile.list - timestep list for tracking

A description of their format follows:

C.1.1 The vortsname file

Format:
<num_of_files>
<dataset_1>
.
.
.
<dataset_last>

C.1.2 The datainfo file

Format:
ndim= <num_of_dim>
NX= <x_dim>
NY= <y_dim>
NZ= <z_dim>
data= <datafiles_path>
basename= <resultfiles_path>
datetime= <1.0>
thresh= <thresh_percent>
rule= <rule_value>

C.1.3 The datafile.list file

Format:
./result
<datasetname><first_time_step_no>
.
.
.
<datasetname><last_time_step_no>

C.2 Distributed Feature Extraction and Tracking Output File Formats

• .dat
• .attr
• .oct
• .trakfile
• .addl
• .poly

C.2.1 The .dat file

Format:
#Without rule, threshold, object number
<thresh %><num_of_obj>
0.0 1
C.2.2 The .attr file

Format:
//Threshold <thresh_val>
<num_of_obj>
4 (delimiter)
(For each obj)
#Object #<obj_num>
Numnodes 1 1 <num_cells>
Mass 1 1 <mass_val>
Centroid 1 3 <centx><centy><centz>

C.2.3 The .oct file

Format:
- time (fix)
- dim of own partition
- Number of objects
For each object
- ptr -> Get Number
- No. of Cells, Mass, Centx, Centy, Centz
- Ixx, Iyy, Izz, Ixy, Iyz, Izx
- 0 0 0 0
For each cell (for min mode)
Nodex, Nodey, Nodez, S Value

C.2.4 The .trakfile file

Format:
For each object
- datafile name, datatime, Mass, num_cells, centx, centy, centz, Ixx, Iyy, Ixy, Iyz, Izx

Format of .addl and .polyfiles same as before.
References


