AstroViz: A Parallel Visualization Tool for Astrophysical Applications

Christine Corbett Moran

2009-12-15

Abstract

Data analysis tools must keep pace with the scientific investigations that produce the data. This is especially true in astrophysics where numerical simulations are being pushed to higher resolution and larger scale than previously possible and scientists are pressed to filter and synthesize results from unprecedented amounts of data. To handle large quantities of data and the demand to visualize and analyze in real time, an astrophysics analysis tool must function in parallel. To be useful across a variety of sub-disciplines and simulation codes, an analysis tool must handle a wide variety of data formats, have a short learning curve, and be extensible to additions as codes and analysis objectives evolve. AstroViz is a data analysis and visualization tool, built upon the existing and widely used open-source application ParaView, that addresses these needs. AstroViz can be run either serially on a client machine or in parallel with a client-server architecture, in which case the data reading, analysis, and rendering for display are distributed via MPI on the server and controlled and displayed, in an interactive 3D view, data plot, or table, on the client. AstroViz handles a variety of common data formats including VTK legacy, Tipsy binary, HDF5, and comma separated values and a variety of common data analysis tasks including computing velocity dispersions, computing moments of inertia, halo finding, and data reduction via thresholding. Since AstroViz is implemented as an plugin for ParaView, it has a short learning curve and is easily extensible. AstroViz is an open-source and freely available tool.

Contents

1 Introduction

In Section 1, I provide an overview of scientific visualization, scientific visualization in astrophysics, and why there is a need for a common parallel, functional and extensible data analysis and visualization tool in astrophysics. In Section 2, I review some fundamental algorithms and data structures used throughout the implementation of AstroViz. In Section 3, I review some fundamental concepts in parallel computing which will later be built upon in addressing the parallel analysis and visualization capabilities of AstroViz. In Section 4, I introduce the open-source toolkits VTK[6] and ParaView[18] upon which AstroViz is built. In Section 5, I introduce the ParaView plugin AstroViz, its features, and the theoretical details of its implementation. In Section 6, I address questions of performance, providing quantitative numbers obtained on the Horus visualization cluster at the Swiss National Computing Center and the ZBox3 cluster at the University of Zurich. In Section 7, I give an outlook of further work on AstroViz. Finally, in Section 8, I review the results and provide some conclusions based on real world usage of AstroViz.

1.1 Scientific visualization

From the complex designs codifying the calendar of the ancient Mayans to scratchings in clay tablets in Mesopotamia indicating a quantity of goods possessed, visualization has been a powerful way to communicate numbers and mathematical entities for thousands of years. In ancient times visualization was a complex process done by hand. The process has evolved dramatically and with the advent of computers we have the ability to collect and analyze more information than ever before possible. While Edwin Hubble deduced Hubble's law from a plot of just 24 objects, scientists push today's computers to produce simulations of billions of interacting objects from which they then are pressed to synthesize physical insights. The field of scientific visualization, concerned with the display and analysis of scientific data, has likewise evolved to adopt computers as its primary tool.

Scientific visualization can be thought of at its base as a function mapping numbers to images, but in reality is a complex process involving several steps: filtering data, choosing a representation (2D, 3D etc.), choosing a desired level of interactivity, and customizing the manner in which the data is displayed. There are many programs which facilitate such a process. A brief overview of those most commonly used in the astrophysics community is given here. For displaying 2D or simple 3D plots, astrophysicists use programs such as gnuplot, SuperMongo, or scripting languages such as Python, Matlab or IDL. For interactive, possibly 3D visualization, astrophysicists use Tipsy[1], yt[23], or develop a custom tool on a code-by-code basis. For a combination of 2D and 3D plotting capabilities, realtime and offline analysis, scripting and graphical control astrophysicists use the applications VisIt[2] or ParaView[18].

There are several decisions to be made in the process of scientific visualization. There is a tradeoff between interactive visualization, which requires more computing power but can give greater insight via the availability of immediate feedback, and offline visualization, which can display data in more intricate detail but has a necessary delay between defining the

parameters of the visualization and viewing the results, due to the computational intensity of achieving this detail. There is also a tradeoff between displaying the entire data set, which may very well be too much for the scientist to process visually in a meaningful manner, and displaying only a subset of the data which the scientist can analyze but which may miss some information inherent in the overall structure. The overall goal for the scientist must also be considered. Common aims of scientific visualization are to uncover patterns in the data, compare the data to phenomena observed in nature, and to compare different methods, for example simulation codes, for generating the data.

1.2 Visualization in astrophysics: the need for AstroViz

In projects such as the Code Comparison Project[10], the need for a common data visualization and analysis workflow in numerical astrophysics becomes apparent. In comparing results from different codes, it is helpful to examine the data visually. If one is to organize a fair comparison, the same program must be used. Furthermore, if a program has the capability to view multiple images at once and link the response to interaction with these images, the researcher can zoom in and examine the same region of the output of different codes, greatly enhancing ease of comparison. For these reasons, along with the powerful analysis and parallel features and the extensibility thereof, the researchers in [10] chose ParaView in conjunction with developing a ParaView plugin encompassing some analysis tools for structure identification.

As it stands today, in much of the community each code has its preferred analysis and visualization tools, along with a custom, code specific format. With the advent of the Hierarchical Data Format (HDF), the practice of creating a code specific format is quickly changing to instead use this common, platform independent format, to the advantage of researchers who wish to quickly and easily begin with data analysis and visualization of a new code. This process is still not the de facto standard, however, meaning that much of the community working with PKDGRAV [20], does analysis and visualization with Tipsy [1], much of the community working with Enzo [15] does analysis and visualization with yt [23], much of the community working with Gadget does analysis and visualization with VisIt [2], etc.

As numerical astrophysicists move towards petascale computing, serial data analysis and visualization tools are no longer appropriate. Simulations will not fit into a single machine's memory, even if pre-filtered. Moreover, as most computing is done remotely, data analysis and visualization is most efficient in a client-server mode, where the server runs where the data is hosted and the client runs on the researcher's local machine, a setup which minimizes data transfer. The challenges in parallel visualization and analysis, including client-server connections, parallel rendering, computer graphics libraries, and a graphical user interface are too great to be addressed by a single researcher each time a new code is developed. Yet, each code has its own challenges and each researcher their own analysis objects, meaning that a one size fits all solution is also unsatisfactory.

ParaView and its plugin architecture reaches the balance between these issues. ParaView is a powerful parallel data analysis and visualization library which additionally supports

building libraries, called plugins, by which a researcher can easily extend ParaView's functionality without going deeply into the internals of ParaView. This architecture is desirable because there is only one user interface to learn and functionality is easily added by these libraries at run time; once the relevant plugin is loaded, the features are immediately available for use.

AstroViz is a data analysis and visualization tool, implemented as a ParaView plugin that addresses the need of the astrophysical community to have a common, easy to learn, and scalable data analysis and visualization tool. The AstroViz plugin adds data format support to ParaView for reading the Tipsy binary format in parallel, reading in only a subset of particles as indicated by a simple ASCII marked particle file and reading in particle attributes via a simple ASCII format in parallel. AstroViz adds several analysis capabilities to ParaView. These include calculating the center of mass, Friends-of-Friends halo finding, smoothing attributes over those of neighbor particles, calculating the principle moments of inertia, computing the virial radius, and computing the following quantities as a function of radius: circular velocity, density, angular momentum, average radial and tangential velocities, velocity dispersion values, and average and cumulative values of attributes in the data set. Each of these analysis tasks functions in parallel.

Since AstroViz is implemented as an plugin for ParaView, it has a short learning curve and is easily extensible. A menu for using AstroViz features as well as common ParaView analysis features is added to the ParaView graphical user interface and color maps comparable to those available in Tipsy are provided. In Section 7, I detail plans to extend AstroViz's format handling and analysis tasks to the SPH, theoretical cosmology, and observational astrophysics communities. AstroViz is an open-source, freely available tool.

2 Fundamental Algorithmic Concepts

In this section, I present some fundamental algorithms and concepts which will later be built upon in the algorithmic design of the analysis features of AstroViz. First I describe the Kd-Tree data structure, which offers an efficient way to do d-dimensional range queries. For example, if the tree is built of three dimensional points, the points in the tree which reside in a 3D region of space can be efficiently located, a capability which is used by the *Virial Radius* finder, *Friends-of-Friends Halo Finder*, and *Neighbor Smooth* algorithms in AstroViz. Next, I review root finding methodology, that is how to, for a given function, efficiently find the point or points at which the function crosses zero. This methodology is used to aid the *Virial Radius* finding algorithm.

2.1 Kd-Trees

Kd-Trees are a way to create a binary tree partitioning of d-dimensional space[5]. The partitioning proceeds as follows: first a point p is chosen to be the root of the tree, then a split is made on one of the d dimensions. Points which have a value of this dimension greater than p's are added to p's right subtree while points which have a value of this dimension

less than p's are added to p's left subtree. There are multiple ways one can choose which dimension to split on as well as multiple ways to decide which point to insert as the left or right child of the splitting node. One example way to choose which dimension to split on would be to cycle through the dimension to split on as one goes down the tree. One example way of choosing which point to insert next is to pick the median of the right-subtree points with respect to the d dimension and make this the right child and then pick the median of the left-subtree with respect to the d dimension and make this the left child. These choices can help to ensure that the final tree is balanced. The number of points to store in a leaf node can also be varied, the tree could have each leaf node represent a single point, or the depth of the tree could be chosen such that each leaf node contains at most a fixed number of points. **Figure 1** depicts a 2D Kd-Tree constructed from 7 points.

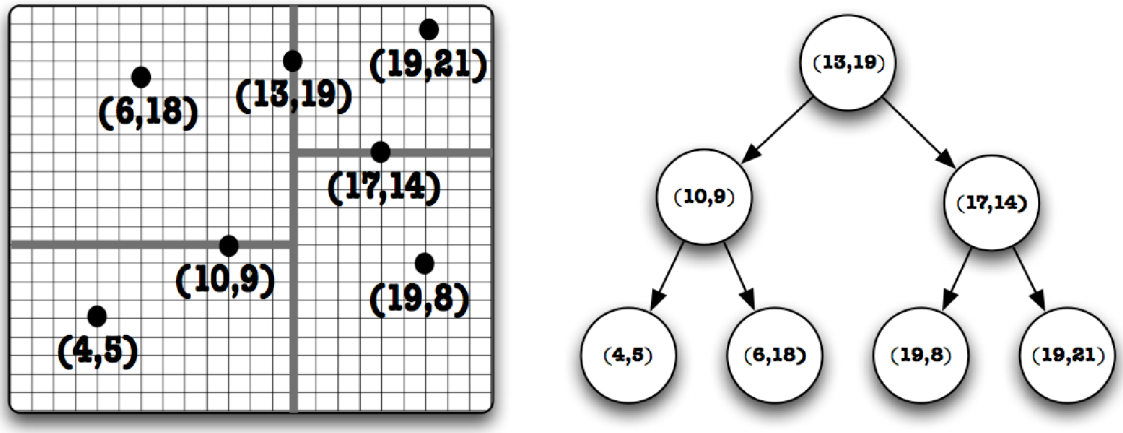


Figure 1: A 2D Kd-Tree constructed from 7 points. The left hand side depicts the decomposition of space into rectangles and the right hand side depicts the resulting Kd-Tree.

Building a Kd-Tree affords several advantages. A search for a point in the tree, a neighboring point in space, all points within a given radius, or the N-nearest neighbors of a point are all efficient operations. The computational complexity of these algorithms, as well as that for constructing the tree itself is briefly reviewed here. To construct a Kd-Tree of n points takes $O(n \log n)$ time and $O(n)$ space[9]. See **Algorithm 1** for the pseudocode for the tree build for the procedure followed in AstroViz. A query for points which reside in a rectangular range can be performed in $O(n^{1-\frac{1}{d}} + k)$ time where k is the number of reported points and d is the number of dimensions[9]. See **Algorithm 2** for the pseudocode for the tree build for the procedure followed in AstroViz. In the case of AstroViz where we most often work in 3D space $d = 3$. For a full description of the algorithms and derivation of their complexity, the reader is referred to the excellent *Computational Geometry: algorithms and applications* by Mark de Berg [9]. For full details of the implementation, the reader is referred to the *vtkKdTree* and *vtkPKdTree* classes in the ParaView source[18].

Algorithm 1: Algorithm to build a Kd-Tree, reproduced from [9] and generalized in a standard manner to d-dimensions.

`BuildKdTree(`p,$depth$`)`;
Input: Set of points p of dimensionality d and current $depth$
Output: The root of the constructed tree contains set p
if p contains exactly one point l **then**
  return a leaf containing l;
else
  $d_s \longleftarrow depth \mod d$; `//` d_s `is the splitting dimension`
  $H \longleftarrow$ hyperplane perpendicular to d_s going through l, the point with the median d_s coordinate of the points in p;
  $p_1 \longleftarrow$ set of points lower than this H;
  $p_2 \longleftarrow$ set of points higher than H;
end
$v_{left} \longleftarrow$ `BuildKdTree(`p_1,$depth+1$`)`;
$v_{right} \longleftarrow$ `BuildKdTree(`p_2,$depth+1$`)`;
Create a node v storing l with v_{left} as the left child of v and v_{right} as the right child of v;
return v;

Algorithm 2: Algorithm to search a Kd-Tree, reproduced from [9] and generalized in a standard manner to d-dimensions.

`SearchKdTree(`v,R,p`)`;
Input: The root of a Kd-Tree v, a range R, and a list of points found in that range
Output: All points at leaves below v which lie in R
if v is a leaf **then**
  if v is contained in R **then**
    Add the point stored at v to p;
    return
  end
end
for $v_{child} \in \{$ `LeftChild(`v`)`, `RightChild(`v`)`$\}$ **do**
  `/* Bounds(`v`) returns the geometric boundary of the subtree of` v
    `based on the splitting hyperplanes. It can in principle be`
    `stored at construction and retrieved in constant time. */`
  if `Bounds(`v_{child}`)` is fully contained in R **then**
    Add the points in v_{child} subtree to p;
  else if `Bounds(`v_{child}`))` intersects R **then**
    `SearchKdTree(`v_{child}, R, p`)`;
  end
end

2.2 Root finding

Root finding methodology is briefly reviewed here. When the derivative of a function g is directly and efficiently computable, a root finding method such as Newton's method[13] which relies on it is appropriate. Newton's method utilizes a linear approximation of the function g and its derivative g' to compute $g(x)$ from a previously computed value $g(x_{\text{old}})$. The point-slope formula then gives

$$g(x) = g(x_{\text{old}}) + g'(x_{\text{old}})(x - x_{\text{old}}). \tag{1}$$

The root of the equation $g(x) = 0$ is thus at

$$x_{\text{new}} = x_{\text{old}} - \frac{g(x_{\text{old}})}{g'(x_{\text{old}})}. \tag{2}$$

Newton's method then applies this formula iteratively

$$x_{\text{n+1}} = x_{\text{n}} - \frac{g(x_{\text{n}})}{g'(x_{\text{n}})}, \tag{3}$$

until $g(x_n) < \epsilon$, for some small tolerance value ϵ. The convergence can be measured by measuring the change in the error function from one iteration to the next. Newton's method converges quadratically in the error. See [13] for the full derivation of the rate of convergence of Newton's method.

The disadvantage of Newton's method, while it provides relatively quick convergence, is that the value of the derivative of the function g must be known. Another method–the method of secants–is used to find the root of a function for which we do not have access to its derivative or for which this derivative would be expensive enough to compute to justify the hit in the convergence rate from using the method of secants, a hit which fundamentally stems from using a more incomplete picture of the function g within the algorithm. Essentially it replaces the $\frac{1}{g'}$ term by an approximation using the secant line between the value of x at the previous iteration and the value at the next iteration $\frac{x_n - x_{n-1}}{g(x_n) - g(x_{n-1})}$. Replacing $\frac{1}{g'(x_{\text{n}})}$ by this in Newton's equation gives us the method-of-secants

$$x_{\text{n+1}} = x_{\text{n}} - \frac{x_n - x_{n-1}}{g(x_n) - g(x_{n-1})} g(x_{\text{n}}). \tag{4}$$

The convergence rate of this method is super-linear. See [13] for the full derivation of the convergence rate.

A few modifications to the method of secants will give us the Illinois root finding method. Essentially, the idea is to use a few heuristics to achieve better in-practice convergence rates than the basic method of secants and avoid potential worst case scenarios which would cause the method to stray far from the root. To this end, we begin with a pair of points which bracket the root, then always keep the root bracketed by retention of the endpoints. By keeping one endpoint negative and one positive, if the precondition that the root is

bracketed is satisfied before execution of an iteration, it will continue to be so afterward. Thus, rather than simply discarding the older of the two guesses for the root's location, we discard the point with the same sign as $g(x_{n+1})$. By only examining intervals in which the root lies, we ensure that we never stray far from the root. The use of a second heuristic, to artificially halve the value of g when we retain an endpoint more than once, gives the full Illinois root finding method. This is a practice which discourages retaining an endpoint for too long; as g is halved after each step, eventually a secant step will be taken to move the endpoint. Thus, while the theoretical convergence rates of the method of secants and the Illinois method are identical, in practice the two heuristics used by the Illinois method result in most situations in faster convergence rates as compared to theoretical worst case.

3 Fundamental Concepts in Parallel Computing

In scientific computing there are often problems scientists want to address which require too much memory, storage, or time to be run on a single modern desktop computer. High performance computers aim to make serial computations faster and enable running processes in parallel. One can quantify how much faster an algorithm will run on a parallel computer with N processes via the *speedup*, where

$$Speedup = \frac{T(1)}{T(N)} \tag{5}$$

and $T(p)$ is the time it takes to execute the algorithm on p processes. Every algorithm has some serial component so as processes are added to the computation of a parallel algorithm, a decreasing return is seen on algorithmic speed. If s is the fraction of the algorithm that is serial, then the algorithm will instead have the following speedup, known as Amdahl's law,

$$Speedup = \frac{T(1)}{T(1)(s + \frac{1-s}{N})} \tag{6}$$

One can explore the scaling of an algorithm with both the problem size and the number of processors. *Strong scaling* is the scaling of the running time with increasing number of processors while keeping the problem size fixed. *Weak scaling* is the scaling of the running time with increasing problem size and increasing number of processors–essentially how things scale if we have every processor do the same amount of work even as we scale up the number of processors available.

3.1 Classification

There are two types of parallelism: *data parallelism* and *task parallelism*. In *data parallelism* each process runs the same code but acts on a different piece of data. In *task parallelism* each process performs different operations. There are two main architectures of parallel computers, *shared memory* and *distributed memory* machines and two corresponding multi-platform programming frameworks to take advantage of these, *OpenMP* and *MPI*. *Shared*

memory computers have multiple processors sharing a global memory space and thus can efficiently exchange information. *Distributed memory* computing clusters connect a network of computers, each having their own local memory, which communicate via messages over the network. Many modern clusters mix the two by using a distributed memory model but one where the individual computers in the network have multi-core processors which operate under a shared memory model.

OpenMP is an API which supports multi-platform shared memory multiprocessing in C, C++ and Fortran. It is mainly used for loop parallelization and is easier to program for and debug than parallelism using distributed memory, because in *OpenMP*, parallel instructions can be gradually added, most serial code does not need changed, and the program can still be run in serial if desired. Modern distributed memory computer clusters use MPI, which stands for *Message Passing Interface*, and is a specification of an API handling communication between computing nodes. It used to be that such an interface was written in an ad-hoc case-by-case basis for each supercomputing architecture. This practice created a great deal of overhead with switching architectures. Since MPI's introduction in 1994, it has become the de-facto standard for handling communication between different nodes of a distributed memory computing cluster. There are many implementations of the API, most commonly in the C or C++ languages.

3.2 Ghost cells

Ghost cells are a concept relevant to *data parallelism*. In designing a data parallel algorithm sometimes it is not possible to achieve correct performance in a clean data parallel way. Consider the external faces algorithm in which the goal is to compute the cell faces which have no neighbors. If data is divided among processors, the naive algorithm may produce incorrect results for instances where two neighboring cells are placed on different processes. The introduction of the concept of ghost cells aims to solve this problem at the cost of a minimal amount of data replication across processors.

Ghost cells are cells which belong to one processor but are nevertheless duplicated on other processors. Typically a ghost cell is one which shares a boundary with a cell on another process. The ghost cell is then duplicated on the boundary processes. Some algorithms may not function correctly even in the presence of a single layer of ghost cells. We may need to repeat this procedure, in effect creating ghost cells of ghost cells. Consider an algorithm which requires the N nearest neighbors of a cell. For a naive implementation of this algorithm it would only be guaranteed to function correctly in the presence of N layers of ghost cells, as in the worst case each neighbor would reside on a different process. For situations such as these we add the concept of a ghost cell *level*. A ghost cell of level 1 is a cell which shares a boundary with a cell on another process. A ghost cell of level 2 is a cell which shares a boundary with a ghost cell of level 1. Armed with this concept, for every algorithm we can define a level of ghost cells which must be duplicated across processors for the algorithm to correctly function. Typically algorithms are designed to minimize the number of ghost cells required for correct functionality. Thus a data parallel algorithm is again achieved if we introduce the number of ghost cells necessary for correct operation and, after computing

the output of the algorithm, throw away or ignore the ghost cells.

Figure 2 depicts the difference in the operation of the external faces algorithm operating on data split across three processors in the case of no ghost cells on the left and one level of ghost cells on the right. We can see the algorithm produces the incorrect results without ghost cells but correctly identifies the external faces if we duplicate one level of ghost cells across processors. The ghost cells are removed in the end when the data is recombined and, at the cost of additional storage space, we have achieved a correctly functioning data parallel algorithm.

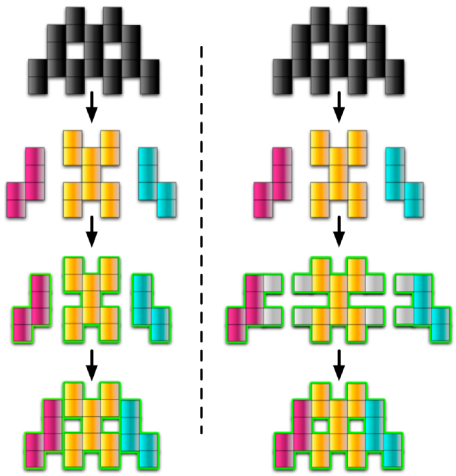


Figure 2: External Faces Algorithm, the left figure without utilizing ghost-cells, the right figure with ghost-cell utilization. The result of the algorithm is depicted via the green border.

4 VTK and ParaView

4.1 VTK

VTK is an open-source visualization toolkit[6] initially created in 1993 as companion software to the 3D graphics textbook *The Visualization Toolkit*[16]. It is released under an open-source BSD license and as a result an active developer community has formed around the toolkit with many researchers and professionals integrating it into commercial and open-source applications and working to further improve its codebase. Although VTK is open-source, its development is overseen by a commercial entity, Kitware, Inc.[3], a company formed by the initial VTK authors to provide commercial development support and consulting related to VTK and a suite of other open-source visualization applications. Additionally, the development of VTK is funded by several U.S. national laboratories including Sandia National Laboratory, Los Alamos National Laboratory and the U.S. Army Research Laboratory.

The visualization model in VTK is known as a pipeline, which is essentially a series of procedures transforming data into graphical information. More specifically, the pipeline consists of objects to represent data, objects to operate on data, and a direction of data flow. In VTK, a class which produces an object to represent data from an input file is known as a *reader*, and a class which operates on data is known as a *filter*. These terms will be used throughout this thesis. The basic design of VTK consists of a C++ class library and a wrapper layer that allows the use of these classes via Java, Tcl or Python. Finally, there are optional additions to VTK utilizing MPI which aid the creation of VTK readers and filters which function in parallel. These additions consist of a data distribution filter, a parallel Kd-Tree data structure, and an MPI Controller class which simplifies MPI calls and allows arbitrary VTK objects to be sent and received via MPI. **Figure 3** depicts the VTK system architecture.

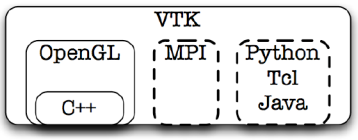


Figure 3: VTK system architecture

4.2 ParaView

ParaView is an open-source graphical user interface for VTK with additional functionality including the capability to perform rendering in parallel[18] and a client-server architecture enabling visualization and analysis to be performed on a server while being viewed and driven from a client. ParaView, like VTK, is open-sourced under a BSD license and its development is overseen by the commercial entity, Kitware, Inc.[3].

ParaView is multi-platform, extensible via its plugin architecture, and natively supports many common data analysis tasks and data formats. As it builds upon VTK, any VTK functionality can in principle be invoked. In practice not all VTK functionality is exposed by default but can easily be exposed or extended via the plugin architecture previously mentioned and discussed in more detail below. Exposing VTK functionality is as easy as writing a short XML file. **Figure 4** reviews the ParaView architecture. A description of

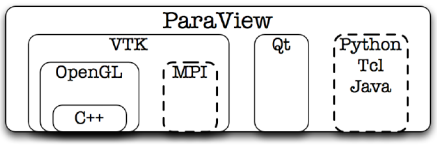


Figure 4: ParaView system architecture

the full extent of ParaView's core data analysis and visualization features is beyond the

scope of this thesis, but a brief overview will be given of those features most interesting to astrophysicists. The reader is referred to [18] for further details. ParaView includes native support many data formats including VTK, CSV, and HDF5, as well handling structured and unstructured data and multi-block and AMR data sets. Data selection, subsampling, and extraction is enabled and isosurfaces can be generated from all datasets. ParaView has an intuitive GUI with a variety of features as depicted in **Figure 5**. In addition ParaView is scriptable via the Python language and can be run as a batch application using the Python interface.

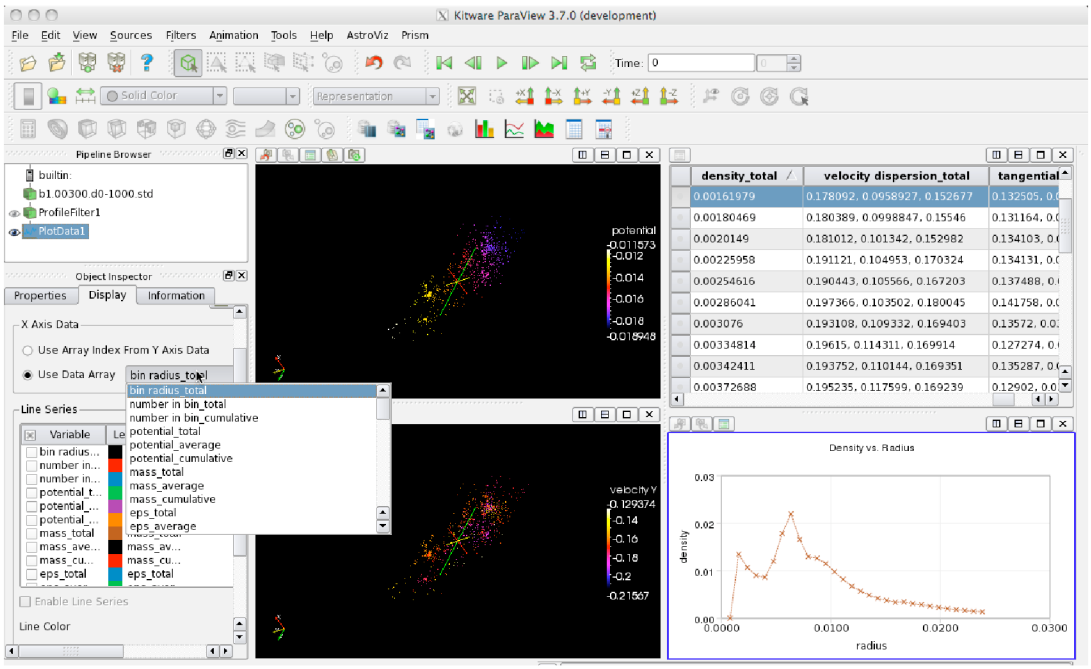


Figure 5: ParaView analysis

4.2.1 ParaView plugins

ParaView plugins are a simple way of extending ParaView. Plugins are easier to create and deploy than directly modifying source code and create an interface for developers to write modular software projects and compile them against ParaView to produce libraries. The user can load any number of these libraries at runtime immediately extending ParaView's functionality.

Implementation wise, a plugin consists of C++ source code to implement the plugin features, XML files to allow ParaView to use the plugin, and a *cmake* file to set up the plugin's build environment. **Figure 6** depicts the plugin architecture. There are several types of ParaView plugins. On the server-side there are filters, readers, and writers. On the client-side there are hooks into the server-side filter, reader, and writer functionality as well as the ability to create GUI modifications including customized object panels, toolbars,

and views. There is an active community of ParaView plugin developers which has resulted in the availability of scores of plugins loadable at runtime which quickly and simply extend ParaView's functionality for specific tasks. Some examples are plugins which add informatics visualization capabilities, optimizations for viewing point particles, and plugins enabling domain specific analysis tasks, for example, those aimed at the climatology, medical, or physics domains.

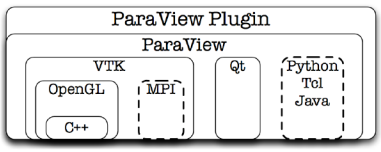


Figure 6: ParaView plugin system architecture

4.2.2 Parallel features

ParaView represents data as collections of points and cells and most calculations of filters on these points and cells are independent. Thus data parallelism is a good choice for ParaView. Each process performs the same operation on its piece of data and where necessary ghost cells are utilized to ensure correct results at processor boundaries. Moreover, ParaView uses the IceT parallel rendering library[14] which enables each process to create an image based on its partition of geometry with these images collectively composited. This has the benefit that at no point must the entire dataset be collected on a single node to do rendering.

The ParaView parallel pipeline is essentially the serial pipeline replicated on each process with additional communication overhead for certain filters. The IceT parallel rendering library sits at the end of the pipeline. The pipeline is depicted in **Figure 7**. First, the parallel reader on each process reads a subset of the data. Next, an optional redistribution of the data is done if the reader is unable to itself ensure it reads data specific to a convex region of space. This data distribution procedure can also create ghost cells as needed by later steps in the pipeline. Finally the identical VTK analysis task is executed on each process and the IceT parallel rendering library handles the rendering on each node, compositing the image and sending it to the client to be displayed.

The full technical details of the IceT parallel rendering library are given in[14]. Here I will give a quick overview of the powerful features IceT offers as well as a few limitations that must be worked around if necessary. The library uses a concept called *sort-last* parallel rendering as opposed to *sort-middle* or *sort-first*. *Sort-last*'s key difference is that it renders static versions of the image before compositing these into a final image while *sort-middle* or *sort-first* allocate a particular region of the screen to a process, sending the process the appropriate geometry for each frame. The image compositing process performed by IceT is depicted in **Figure 8**. As we can see in the figure, each process performs the same rendering operations it would in serial only it performs them on a subset of the entire geometry. The compositing is then done in a binary-tree, placing the portion of the geometry closest

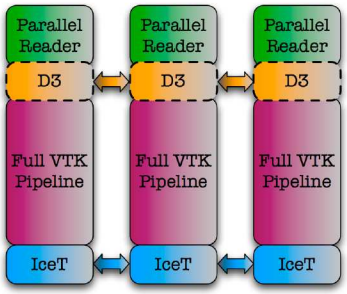


Figure 7: ParaView parallel pipeline

to the viewer at the front. This gives us a clue as to one of the limitations of *sort-last* rendering. If we wanted to use transparency of some sort, this method would no longer be useful. The rendered image on a process may now depend on rendering results from other processes, meaning we have to employ a different strategy to efficiently render and composite appropriately.

Doing data analysis and visualization in parallel is achieved by launching a ParaView server as an MPI job on a distributed or shared memory machine and connecting to the machine from a client. To keep a high frame rate, ParaView has several strategies. First, it can either perform rendering on the client, on the server, or on some combination of the two. Secondly, it employs the the concept of Level-Of-Detail (hereafter LOD), which is a geometric decimation during an interactive render. When the user interacts with the data a decimated model is typically used and only when the mouse is released is the image rendered at full resolution. This interactive decimation is shown by **Figure 9** and the final full resolution image is shown in **Figure 10**. We can see in the realtime interaction that, while the individual velocity vectors are less apparent, in the lower LOD image the key features are nevertheless still visible. Finally, image compression in the transfer between client and server can be employed. When and where to perform rendering and the situations under which to use decimated models or compression are all parameters controlled by the client.

4.2.3 Data distribution

Most ParaView filters function optimally if data is distributed spatially across processes as this minimizes the need for ghost cells and makes operations which have some inherent spatial locality to them more efficient. It is also important that the amount of data each process is responsible for is balanced so that no processor is responsible for a disproportionate amount of data and thus prone to becoming a bottleneck. If the data has some defined structure to it, it is easy to do this directly while reading the file as each process knows exactly what coordinate extent it is responsible for reading and where in the file this coordinate extent corresponds to. If the data has no predefined structure, then the entire file must first be read in before the data can be (optionally) redistributed spatially.

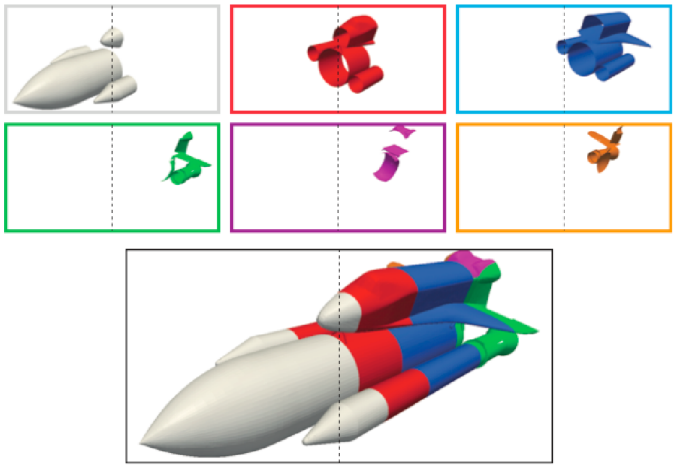


Figure 8: Parallel rendering with IceT. Figure reproduced from [14].

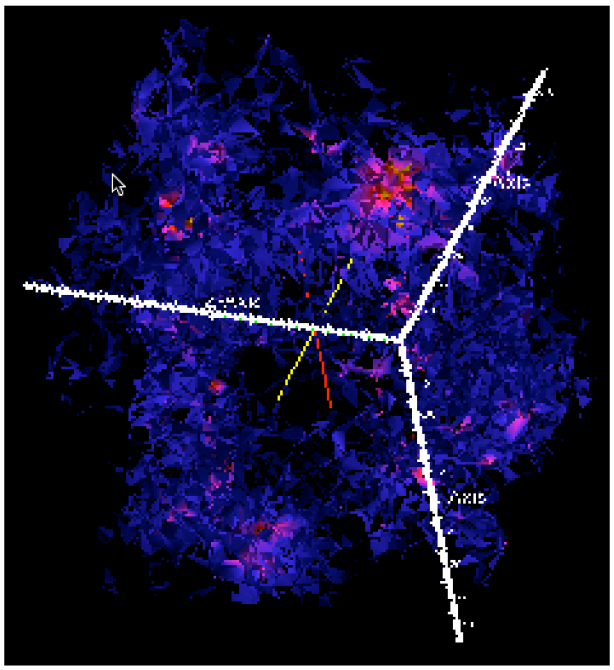


Figure 9: ParaView realtime interaction illustrating the LOD concept. Displayed here is the interactive lower LOD image.

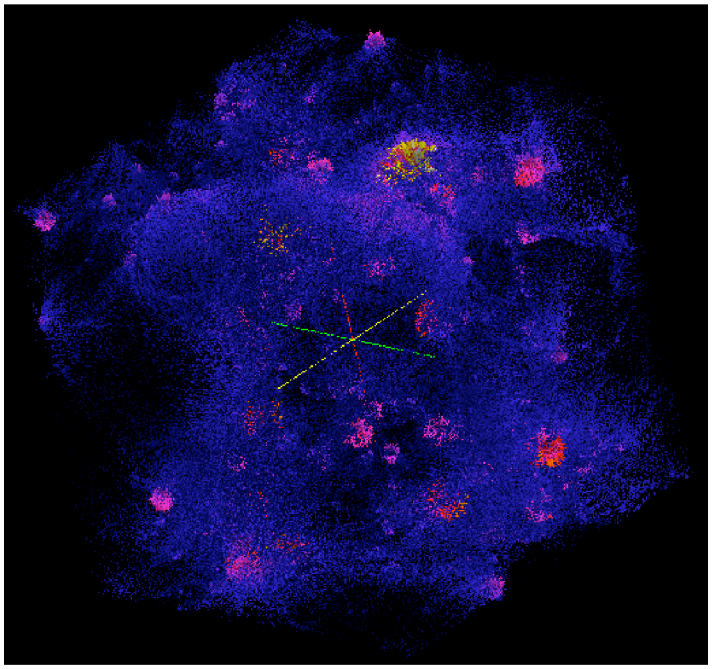


Figure 10: ParaView realtime interaction illustrating the LOD concept. Displayed here is the high resolution, full LOD image, displayed only after realtime interaction has ceased.

Although distributing the data spatially is best for the operation of most filters, it can take time to run on a large dataset. The ParaView D3 filter handles data distribution in ParaView and AstroViz and can be executed alone or automatically as a component of the Tipsy binary reader. It operates by constructing a Kd-Tree in parallel, after each process has read in its portion of a distributed dataset. It then assigns contiguous convex spatial regions to processors and redistributes data accordingly. **Figure 11** depicts data read in by the Tipsy reader before being distributed spatially. While the file seems to have had some inherent spatial distribution, in the middle of the spiral we can see this breaks down. For filters which require ghost cells, at the middle of the spiral some cells may be touching all three other processors meaning three ghost cells for every such cell would be created. If we instead run the data distribution filter we get the significantly more regular partitioning depicted in **Figure 12** and any later filters run will save in time and spatial complexity.

5 AstroViz ParaView plugin

In this section, I review each feature added by the AstroViz ParaView plugin and where applicable discuss the theoretical details of the calculation and the implementation in both serial and parallel. First, I describe the graphical user interface additions, next the file format support added, and finally the analysis features available as a component of AstroViz.

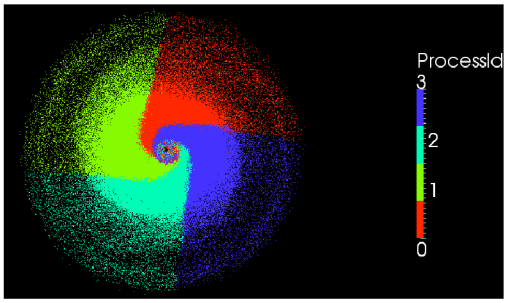


Figure 11: Naive partitioning of unstructured data

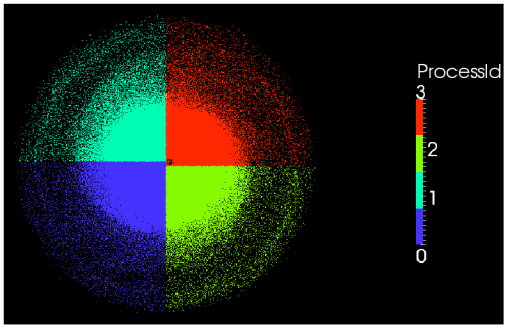


Figure 12: Partitioning of unstructured data after running data distribution filter

5.1 Graphical User Interface

The ParaView GUI is a fully functional and intuitive user interface and the plugin architecture of development allows the developer to add analysis features without fundamental modifications of the GUI. Thus, the GUI modifications by AstroViz are minor but are nevertheless valuable in assisting the user with use of AstroViz and with the possible transition from analysis with the Tipsy program to analysis with ParaView and AstroViz.

The AstroViz menu allows the user to find all analysis features added to ParaView by AstroViz in a single place as well as providing a quick link to some of the most commonly used analysis tasks available natively in ParaView. The menu is available under the submenu **AstroViz Analysis** of the **Filters** menu, and contains the options **Add Additional Attribute**, **Center of Mass**, **Friends-Of-Friends Halo Finder**, **Histogram**, **Neighbor Smooth**, **Plot Data**, **Principle Moments of Inertia**, **Profile**, and **Virial Radius**. **Histogram** and **Plot Data** are analysis features available to ParaView natively; the other menu items correspond to features added by the AstroViz plugin. The **WRBB** and **Astro** colormaps available in Tipsy are also available as part of the AstroViz distribution.

5.2 File Formats

ParaView natively supports all astrophysical codes which output to the HDF5 format. These include Ramses[22], Enzo[15], Gadget2[17], Athena[21], and PKDGRAV[20] among many

others. Some of the PKDGRAV and GASOLINE community still rely on Tipsy binaries to store simulation output, however, and many users of Gadget2 still rely on Gadget2 binaries for storage. To serve these communities, AstroViz provides support for parallel reading of Tipsy binaries and in Section 7 plans are detailed for a Gadget2 binary reader. Furthermore, some of the community relies on legacy ASCII formatted files. AstroViz supports these although an efficiency hit is seen as a result when running in parallel.

5.2.1 Tipsy binary reader

The Tipsy binary reader reads Tipsy files in the standard (platform independent) format. The reader is fully parallel so each process reads in only the fraction of data it is responsible for. More precisely, each process p_i reads from $p_i\lfloor\frac{N}{p}\rfloor$ to $min((p_i+1)\lfloor\frac{N}{p}\rfloor, N)$, where N is the total number of particles in the file, given in the header of the file, p is the total number of processors, and i ranges from 0 to $p-1$ indexing each process. The user can specify whether to load all attributes defined in the Tipsy binary file or only to load in the positions of the particles. The user can specify whether the data should be distributed spatially after being read in, which as previously reviewed, is optimal for the operation of most filters.

5.2.2 Marked particle files

As part of the Tipsy binary reader, there is an option to read in a simple text format marked particle file to enable the pre-selection of a subset of particles to be read in. This can reduce the size of the data, but it has the disadvantage of being unsupported in parallel. This was a design decision–in parallel runs ParaView's native marking mechanism should be used. The marked particle files are of the format where first line lists the number of particles in the original Tipsy file, the number of gas particles in the original file, and finally the number of star particles in the original file, each separated by a space. The number of dark particles in the original file can then be inferred. The Tipsy binary file must match these numbers or reading will abort. Beginning on the second line of the marked particle file a marked-particle index is listed, one index per-line.

5.2.3 Add additional ASCII attributes

The *Add Additional Attribute* filter offers the user the option to read in an ASCII file which on the first line lists the number of particles in the file, then, beginning with the second line, additional float attributes are listed, one per line. A set of points equal to the number of particles in the file must have already been loaded in. This filter is quite basic, but useful for those who quickly want to integrate data from external processing applications into a small or medium sized file. This filter operates in parallel although it operates on an ASCII file and thus is inefficient in comparison to reading in a binary file. Its parallel implementation is simple: each process sorts the list of the global ids of particles it is responsible for. The sorting operation is $O(N/p)\log(N/p)$ where N is the number of particles in the entire data set and p is the number of processors. Due to the format of the additional attribute file,

the ids should correspond to the position in the file at which the attribute resides plus one. Then each process starts at the beginning of the file and counts until it reaches the first id it is responsible for, an operation which is $O(N)$ worst case, and finally the ids are searched for in order, an operation which is of order $O(N)$ as well.

5.3 Data Analysis

AstroViz adds several analysis capabilities to ParaView. The *Profile* filter calculates a variety of physical quantities, averages, and cumulative values as a function of radius. The *Principle Moments of Inertia* filter calculates the principle moments of inertia of a collection of particles. The *Center of Mass* filter calculates the center of mass of a collection of particles. The *Neighbor Smooth* filter smooths values over neighbor particle quantities, producing an additional estimate of the local density for each particle. The *Virial Radius* filter calculates the radius from a selected center which corresponds to a specific overdensity. Finally, the *Friends-Of-Friends* filter employs a heuristic to quickly identify candidates for groups of objects which are gravitationally bound.

5.3.1 Profile

The output of the *Profile* filter is the form of a table which can be exported for further analysis in an external program or used within ParaView to perform realtime analysis tasks. Plots can also be generated as a post-processing step. The user interface for the *Profile* command allows the user to select a center around which to profile via the GUI. The user can select a single point, in which case the selected point is considered to be the center, can select a line, in which case the center is defined to be the midpoint of the line, or can manually enter the coordinates of the center point. The user can also select the number of bins to be used. The following quantities are produced by the *Profile* filter and included in the table: radius, number in bin, density, cumulative mass $M(< r)$, circular velocity $\sqrt{M(< r)/r}$, average velocity

$$\langle \vec{v} \rangle = \frac{1}{N} \sum_{i=1}^{N} \vec{v}_i, \tag{7}$$

average radial velocity

$$\langle \vec{v}_{rad} \rangle = \frac{1}{N} \sum_{i=1}^{N} \vec{v}_{rad,i} \tag{8}$$

where

$$\vec{v}_{rad,i} = \frac{\vec{v}_i \cdot \vec{r}_i}{|\vec{r}_i|^2} r_i, \tag{9}$$

average tangential velocity

$$\langle \vec{v}_{tan} \rangle = \frac{1}{N} \sum_{i=1}^{N} (\vec{v}_i - \vec{v}_{rad,i}), \tag{10}$$

average velocity dispersion, radial velocity dispersion, and tangential velocity dispersion, where the dispersion is defined as

$$\vec{\sigma}_v = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\vec{v}_i - \langle\vec{v}\rangle)^2} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\vec{v}_i^2 - \langle\vec{v}\rangle^2}, \tag{11}$$

and average angular momentum

$$\vec{j} = \frac{1}{N}\sum_{i=1}^{N}\vec{r}_i \times \vec{v}_i. \tag{12}$$

The parallel implementation of the *Profile* command is straightforward. Initially a calculation is done in parallel to discover an accurate estimate of the upper-bound for the maximum radius from the selected center within the dataset. This is done by having each process compute its maximum radius and then having a single process compare them all to discover the true global maximum. The computation of the maximum radius is a strict upper bound, as the bounding box of the data on each process is used, rather than computing the radius of each particle which would be $O(N)$, where N would be the maximum number of particles residing on a process over all processes. The bounding box of the data on each process is kept track of as part of the reading process so it is already computed and is $O(1)$ to access. This result will be slightly different than that obtained if run serially, as the bounding boxes may be tighter on subprocesses than an overall bounding box. But, if there is any change, the parallel version is a tighter upper bound for the maximum radius than the serial version and is always guaranteed to be greater than or equal to the true maximum radius over all points within the dataset to the selected center. From the maximum radius upper-bound and the number of bins requested by the user, the extent of each bin is computed and is equal to

$$\text{bin spacing} = \frac{\text{maximum radius estimate}}{\text{total number of bins}}. \tag{13}$$

After the initial setup, the calculation is ready to proceed. If $\vec{c}$ denotes the selected center and $\vec{x}$ the coordinates of a particle, we can compute which bin to which the particle belongs to via the equation

$$\text{bin number} = \left\lfloor \frac{|\vec{x} - \vec{c}|}{\text{bin spacing}} \right\rfloor. \tag{14}$$

Referring again to the equations for profiled quantities above we see they invariably include a summation of a certain quantity over all particles. As we can calculate which bin a given particle belongs to independently of the other particles and the summation operator is commutative and associative, each processor can independently calculate the sum and we can in the end combine the sum from each processor before performing any final computations on these sums. More precisely, each processor loops over all of its particles, and for each particle updates the appropriate bin and bin attributes by adding the particle's attributes to those in the bin. At the very end, the root process combines the sums from all processes

for all bins and then for each bin computes the final value of the profiled quantities which is a function of these summations alone. Thus this last operation is $O(p)$ and the entire *Profile* algorithm is itself $O(N/p)$.

5.3.2 Calculate principle moments of inertia

The *Principle Moments of Inertia* command is implemented according to the standard calculation of the principle moments of inertia of N bodies, reviewed here. The moment of inertia scalar is considered to be

$$I = \sum_{1}^{n} m_i(x_i^2 + y_i^2 + z_i^2) \tag{15}$$

and the moment of inertia tensor is

$$\mathbf{I} = \begin{bmatrix} I_{11} & -I_{12} & -I_{13} \\ -I_{21} & I_{22} & -I_{23} \\ -I_{31} & -I_{32} & I_{33} \end{bmatrix} \tag{16}$$

with the following components

$$I_{11} = I_{xx} = \sum_{k=1}^{N} m_k(y_k^2 + z_k^2), \tag{17}$$

$$I_{22} = I_{yy} = \sum_{k=1}^{N} m_k(x_k^2 + z_k^2), \tag{18}$$

$$I_{33} = I_{zz} = \sum_{k=1}^{N} m_k(x_k^2 + y_k^2), \tag{19}$$

$$I_{12} = I_{xy} = \sum_{k=1}^{N} m_k x_k y_k, \tag{20}$$

$$I_{13} = I_{xz} = \sum_{k=1}^{N} m_k x_k z_k, \tag{21}$$

and

$$I_{23} = I_{yz} = \sum_{k=1}^{N} m_k y_k z_k. \tag{22}$$

To calculate the principle moments of inertia, we utilize the fact that I is real and symmetric, and thus via the finite dimensional spectral theorem, know it can be diagonalized into the form

$$\mathbf{I} = \begin{bmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{bmatrix}. \tag{23}$$

The eigenvalues of I are the principle moments of inertia and the eigenvectors are the principle axes.

The parallel implementation of the computation is straightforward. Referring again to the equations for the moment of inertia tensor above, we see they invariably include, as with the *Profile* filter, a summation of a certain quantity over all particles. As the summation operator is commutative and associative, again as with the *Profile* filter, each processor can independently calculate the sum, and we can in the end combine the sum from each processor before performing any final computations on these sums. More precisely, each processor loops over all of its particles, and for each particle updates its local moment of inertia tensor. At the very end the root process combines the tensors from all processes by summing each component and finally computes the eigenvalues and eigenvectors for this symmetric 3×3 tensor. Thus the *Principle Moments of Inertia* algorithm is $O(N/p)$.

The *Principle Moments of Inertia* filter takes no arguments, other than a specification of which data array contains the mass. The eigenvectors are displayed on screen by displaying three vectors, colored by whether they are the first, the second, or the third moment. These vectors are the eigenvectors scaled by the total maximum distance from the defined center to the edge of the cube to be in a comparable size to the simulation, whatever scale it may be, and thus easily visible in comparison to the simulation. The eigenvectors can also be displayed in the spreadsheet view within ParaView and optionally exported as a comma separated value file for storage and further analysis.

5.3.3 Calculate the center of mass

The center of mass $\vec{c}$ of N particles, where $\vec{x}_j$ is the coordinate vector of the jth particle, is

$$\vec{c} = \frac{\sum_{j=1}^{N} m_j \vec{x}_j}{\sum_{j=1}^{N} m_j}. \tag{24}$$

The parallel implementation of this filter is again straightforward and proceeds in a similar manner to both the *Principle Moments of Inertia* and *Profile* filters. Referring to the equation for the center of mass above, we see that it is computed by dividing the result of two summations of a certain quantity over all particles. As the summation operator is commutative and associative, as with the profile and moment of inertia filters, each processor can independently calculate the sum and we can in the end combine the sum from each processor before performing the division of the result of the two summations to compute the final result. Thus this last operation is $O(p)$ and the entire *Center of Mass* algorithm is itself $O(N/p)$. The *Center of Mass* filter takes no arguments other than a specification of which data array contains the mass. The center of mass computed is displayed on screen by displaying a single point. It can also be displayed in the spreadsheet view and optionally exported as a comma-separated-value file for storage and further analysis.

5.3.4 Smooth particle quantities

The basic idea of the *Neighbor Smooth* filter is to produce an estimate of the local density for particle data or to smooth over a specific attribute with the goal of reducing noise. Locally, the *Neighbor Smooth* filter builds a Kd-Tree, then for each particle it finds the N nearest neighbors, averaging the particle's attribute value with those of the neighbor particles to find smoothed variable value for the particle in question. For those quantities which need a volume to be computed (e.g. density), the volume is considered to be the sphere around a point with radius of the outermost neighbor point.

The parallel implementation of this feature simply makes each process responsible for creating smoothed values for the particles that reside on its process. No smoothing is currently done across processes, which can produce seams in the smoothed values at processor boundaries. In AstroViz 2.0 these problems, as well as issues of efficiency, will be addressed by implementing a new, more efficient algorithm and making use of ghost cells, duplication of particle data across processor boundaries, to handle situations where neighbor particles may be across a particle boundary. An outline of these steps is given in Section 7.

5.3.5 Finding the virial radius

Following[7], the tensor virial theorem relates the total kinetic energy, the thermal energy, the gravitational potential energy and the magnetic energy of a system. In its tensor form, it states that

$$\frac{1}{2}\frac{dI_{jk}^2}{dt^2} = 2T_{jk} + \Pi_{jk} + W_{jk} \tag{25}$$

where the inertia tensor $\mathbf{I}$ (note this is distinct from the tensor we previously referred to as the *moment of inertia tensor*), contributions to the kinetic energy tensor from ordered motion $\mathbf{T}$, contributions to the kinetic energy tensor from random motion $\mathbf{\Pi}$, and potential energy tensor $\mathbf{W}$ are defined as

$$I_{jk} \equiv \int d^3\mathbf{x}\rho x_j x_k, \tag{26}$$

where ρ is the density and the integral is to be taken over all space,

$$T_{jk} \equiv \frac{1}{2}\int d^3\mathbf{x}\rho\bar{v_j}\bar{v_k}, \tag{27}$$

$$\Pi_{jk} \equiv \int d^3\mathbf{x}\rho\sigma_{jk}^2, \tag{28}$$

where σ_{jk}^2 is the velocity dispersion tensor, and

$$W_{jk} \equiv -\int d^3\mathbf{x}\rho(\mathbf{x})x_j\frac{\partial\Phi}{\partial x_k} \tag{29}$$

where Φ is the potential. The tensor statement of the virial theorem can be derived by solving the integral form of the Euler equation considering Ampere's law, the continuity equation and the Poisson equation. Refer to [7] for full details of the derivation.

Since we often approximate e.g. galaxies as time independent, the left side is zero in many situations. We can further simplify the equation by considering the virial theorem in its scalar form, obtained by taking the trace of the potential-energy tensor, to obtain the total potential energy W. Furthermore, it can be shown that $K \equiv \text{trace}(T) + \frac{1}{2}\text{trace}(\Pi)$ represents the kinetic energy. If we consider the system to be in steady state then $\ddot{I} = 0$ and thus the trace of the statement of the tensor virial theorem becomes

$$2K + W = 0, \tag{30}$$

a vast simplification over the tensor virial theorem. The virial radius is formally the radius at which virial equilibrium holds. In practice this radius can be hard to compute so is often approximated by the radius at which the density is equal to a constant α times the critical density $\rho_{\text{crit}} = \frac{3H^2}{8\pi G}$. The critical density is the density of matter required for the universe to be flat, a model which is observationally favored. Typically, $\alpha = 200$ is chosen in the literature

The *Virial Radius* finder filter in AstroViz allows the user to specify $\alpha\rho_{\text{crit}}$ as well as the center from which to search specified in a similar manner to the profile filter. It then runs a root finding algorithm, the Illinois root-finding method[13], to search for the radius from the specified center at which the density is equal to the specified density with an additional heuristic: that the virial radius is located on a portion of the density curve with a negative slope. This second heuristic gives us a best guess approach to bracketing the root. Bracketing the root is important as the density as a function of radius is not a monotonic function and thus can have multiple roots; if the initial points we feed the Illinois root finding method bracket multiple roots then the method may fail.

Since the Illinois method is used to find the virial radius we first need a bracketing pair. As the function is non-monotonic, we do not know this *a priori* and must search for it as well. To search for a good bracketing pair, yet another heuristic is used, namely the softening radius, which can be specified by the user but is in principle a property inherit to the simulation being visualized. This is a good heuristic as it gives us a radius below which it is physically meaningless to probe for a given location. The search for a pair of points which bracket the virial radius is then done using the Fibonacci search algorithm, that is to say it is done via a divide-and-conquer algorithm by searching for a section in which our heuristic that the virial radius is located on a portion of the density curve with a negative slope is satisfied. Pseudocode for this algorithm is provided in **Algorithm 3**. We continue the search until we reach a point where the density decreases for three successive sequences and in this case run the root finder with the first and last point in the sequence as the initial brackets to find the virial radius. If the bracketing heuristic is not satisfied or if the virial radius is not found with the bracket guess, then the algorithm simply prints a warning message and returns. Psuedocode for the virial radius algorithm is provided in **Algorithm 4**.

This algorithm functions in parallel by computing the density within a given radius in parallel. Each process first builds a Kd-Tree representation of its data. Then the process uses the Kd-Tree to find all points which reside within the sphere of the proposed radius

and sums their mass. The density is found by combining the result of this summation for all processors and finally dividing by the volume of the sphere given by the proposed radius.

Algorithm 3: Algorithm to bracket the root of the density function.

Input: A softening value, Center from which to search, a Kd-Tree representing the dataset which is queryable in parallel

Output: A pair of points bracketing the root of the density function

`BracketRoot(`*softening,center,tree*`);`

```
/* Bounds(tree) returns the geometric 3d-bounding box of the tree.
   MaxDistance(center,bounds) returns the maximum distance from the
   center to a corner of the bounding box.                          */
```

$r_{max} \longleftarrow$`MaxDistance(`*center,*`Bounds(`*tree*`))`;
$r[3] \longleftarrow \{softening, softening, softening\}$;
$\rho[3] \longleftarrow \{0, 0, 0\}$;
$fib[2] = \{1, 1\}$;
while $r_2 \leq r_{max}$ **do**
  if $\rho[0] < \rho[1] < \rho[2]$ **then**
    return $\{r[0], r[2]\}$;
  end
  $fib_{next} \longleftarrow fib[0] + fib[1]$;
  $r_{next} \longleftarrow fib_{next} \cdot softening$;
  $\rho_{next} \longleftarrow$`Density(`$r_{next}$*,center,tree*`)`;
  `/* ShiftLeftUpdate(`*array,element*`) moves all elements in` *array* `one index to the left, throwing out the first, and adds` *element* `to the end.  */`
  `ShiftLeftUpdate(`fib*,*fib_{next}`)`;
  `ShiftLeftUpdate(`r*,*r_{next}`)`;
  `ShiftLeftUpdate(`ρ*,*ρ_{next}`)`;
end
return $(0, r_{max})$; `// Unable to find a tight bracket, returning entire interval`

Algorithm 4: Algorithm to find the virial radius.

Input: The critical density, a softening, a center from which to search, a Kd-Tree representing dataset which is queryable in parallel

Output: The radius from the center at which the density equals the critical density, NULL if no such radius is found

`FindVirialRadius(`ρ_{crit}*,softening,center,tree*`)`;
$(b_l, b_r) \longleftarrow$ `BracketRoot(`*softening,center,tree*`)`;
$r \longleftarrow$`Illinois(DensityMinusCriticalDensity,`ρ_{crit}*,center,tree,*b_l*,*b_r`)`;
return r; `// ` r `is NULL if not found`

5.3.6 Friends-Of-Friends halo finder

A halo can be considered as a complete group of objects gravitationally bound to each other. Identifying halos is an important component of many analysis tasks in astrophysics as analyzing their properties can give insight into a wide variety of topics including galaxy formation, star formation, and gravitational lensing, and provides one of many connections between astrophysical simulations and observations[7]. There are a wide variety of different algorithms for identifying halos. Spherical Overdensity[12], AHF[11], SKID[20], and Friends-of-Friends[8] are some of the most frequently used. The variety in halo-finding algorithms stems from computational complexity incurred by using the simple definition above. In principle, calculating the gravitational forces at play for N particles is $O(N^2)$ and is computationally prohibitive for large N. Because of this, a halo-finding algorithm typically makes some approximations, meaning internally it often uses a modified definition of what a halo is considered to be for efficiency considerations.

The Friends-of-Friends halo finding algorithm[8] (hereafter FOF) as mentioned above is the most frequently used halo finding algorithm at present because of its simplicity and efficiency. It changes, for efficiency considerations, the definition of a halo to be the set of objects for which every object in the set is within a certain length scale b, called the *linking length*, from at least one other member in the set. While this definition is purely geometrical, it is conceptually simple and has only one free parameter. If an efficient method is used to discover if there are objects located within sphere of radius b of a given object, the algorithm is itself efficient. Disadvantages of this algorithm are that it is un-satisfyingly non-physical, it is impossible to find substructure i.e. sub-halos of a halo, and it is prone to link two halos together via a *linking bridge*[11].

The *Friends-of-Friends Halo Finder* filter in AstroViz is an FOF halo finder. It functions in parallel by constructing a Kd-Tree on each process. Using this, it recursively maps all points which reside within a linking length of another point to a single unique global id, one unique global id per set of objects for which every object in the set is within a linking length from at least one other member in the set. This algorithm is efficient as we need only inspect the particles at a particular node in the Kd-Tree and those which reside at nodes within a sphere of distance b of that node.

5.3.7 Dependencies of AstroViz filters on data distribution

AstroViz Filters which are independent of data distribution are the Tipsy binary reader, *Center of Mass* filter, *Profile* filter, and the *Principle Moments of Inertia* filter. There are a few AstroViz Filters where data distribution will change result. Currently the *Friends-of-Friends Halo Finder* and the *Neighbor Smooth* filter outputs will be nonsensical in parallel if data is not first spatially distributed as they rely on information about neighbor particles but only have capability to consider particles residing on the same process as neighbors. AstroViz 2.0 will fix this, but for now the filters may be used, but only with caution. These steps are detailed in Section 7. Finally, the marked particle file option of Tipsy binary reader feature does not function in parallel under any circumstances.

6 Performance

In this section I address the performance of AstroViz filters, providing quantitative numbers obtained on the Horus visualization cluster at the Swiss National Computing Center and the ZBox3 cluster at the University of Zurich to illustrate the scaling of AstroViz algorithms across a variety of problem sizes, number of processors available, and available cluster hardware.

6.1 Machines

The ZBox3 supercomputer at the University of Zurich consists of 144 nodes, each with quad core 2.4 GHZ Intel CPUs, Dolphin SCI highspeed interconnects, and 8GB main memory per node. Benchmarks for AstroViz on ZBox3 were done with ParaView 3.6 with software rendering via the OSMesa library.

The Horus HP-XC SVN Visualization cluster at the Swiss National Supercomputing Center (CSCS) has 16 nodes, each with dual core 2.4 GHz AMD Opteron Processor 250 and two 4x Infiniband interconnects. Ten nodes have 8GB main memory, five have 16GB, and one has 32GB for a total of 192GB. Each node has two NVIDIA Quadro FX 4500 GPUs. Benchmarks were done with ParaView 3.6 with GPU rendering enabled. When a number n of processors is quoted for Horus this means that n CPUs and n GPUs are used.

6.2 GHALO simulation

GHALO is a high resolution cosmological simulation calculating the structure of the Galactic dark matter halo.[19] At full resolution, it evolves over three billion dark matter particles from the initial conditions given by the cosmic microwave background radiation to the present. Benchmarks of AstroViz data analysis features were done on GHALO at three different resolutions, the highest with 3,057,221,615 particles (hereafter **GHALO B3**), the second highest with 141,232,694 particles (hereafter **GHALO B2**), and the lowest with 11,254,149 particles (hereafter **GHALO B1**). AstroViz analysis tasks were benchmarked on the ZBox3 and Horus supercomputers.

6.3 Results

The following timings were made: first I examined the performance of each AstroViz filter versus the number of processors used for the two lower GHALO resolutions. Next I compared the performance on a CPU cluster (ZBox3) to a combined CPU and GPU cluster (Horus) on the two lower resolutions. Finally, I examined the performance analyzing the highest resolution GHALO B3 simulation on 512 processors of the ZBox3.

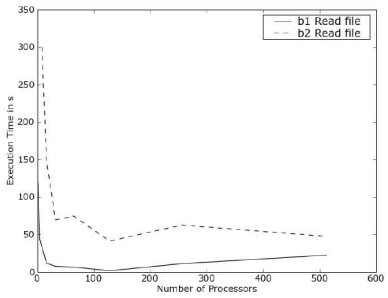
6.3.1 Performance scaling with number of processes

The performance of various AstroViz filters scaling with number of processors available for the two lower GHALO resolutions is depicted in **Figures 13(a)-13(g)**. For the Tipsy binary reader, the performance is depicted in **Figure 13(a)**. We see that an order of magnitude more particles for the same number of processors results in less than an order of magnitude more time to read in the file for the lowest number of processors used. There is a critical processor number, 100, for which at both resolutions reading time is minimized. After this critical point reading time increases, presumably because of communication overhead in preparing the final image for display.

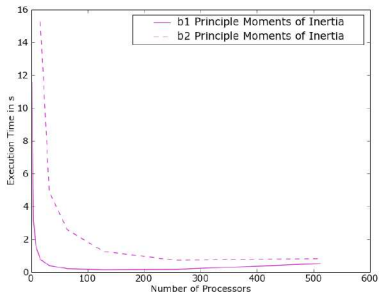
For the trivially data parallel AstroViz algorithms, namely the *Principle Moments of Inertia*, whose performance is depicted in **Figure 13(b)**, *Center of Mass*, whose performance is depicted in **Figure 13(c)**, and *Profile*, whose performance is depicted in **Figure 13(d)**, filters, there is a general decrease in an already low running speed as the number of processors is increased, up to a value of 200. Then there is a gradual increase, presumably due to communication overhead, as the number of processors increases to 500, but this increase in running time is small. It appears in both cases that the curves for both resolutions are asymptoting to the same value. We see that an order of magnitude more particles for the same number of processors results in nowhere near an order of magnitude more time to run.

For the algorithms which rely on Kd-Trees, namely the *Neighbor Smooth*, *Virial Radius* finder, and the *Friends-of-Friends Halo Finder* filters, we get a variety of speedups depending on whether the filter needs to communicate with other processes or operates only on its local Kd-Tree. The *Friends-of-Friends Halo Finder* and *Neighbor Smooth* filters operate only on their local Kd-Tree. Thus, as the number of processors increases there is no communication overhead to contend with and the Kd-Tree each process is responsible for simply gets smaller providing faster searching time and a faster running time for the algorithm. This matches what we see in the running times of these algorithms, depicted in **Figure 13(e)** and **Figure 13(f)** respectively.

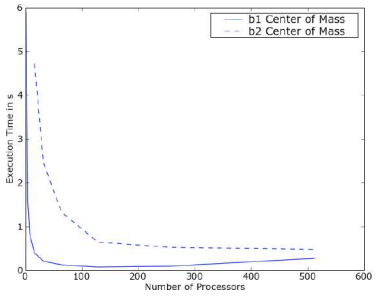
On the other hand, the *Virial Radius* filter relies on being able to find quantities inherent to particles across all processors which reside within a certain radius from a given center. This means that the processors must communicate and if we were to scale the number of processors up indefinitely we would reach the limit of one particle per processor. In this limit there is a great deal of communication required each time the algorithm needs to find all particles within a given radius; one piece of data for every point within the radius would have to be sent. So with this algorithm we expect a critical number of processors to be optimal for a given problem size. This optimal number is the one which has enough processors to reduce the size of the Kd-Trees on each process and speed up search time yet still has enough particles per processor to take advantage of locality and being able to collect results for many particles local to a process reducing communications overhead. This matches what we see in the running time of this algorithm, depicted in **Figure 13(g)**. The optimal number of processors for the lower GHALO resolution is around 50, after which the running time slowly but steadily increases with number of processors. The optimal number of processors for the higher GHALO resolution is around 150, after which it exhibits the same behavior as the



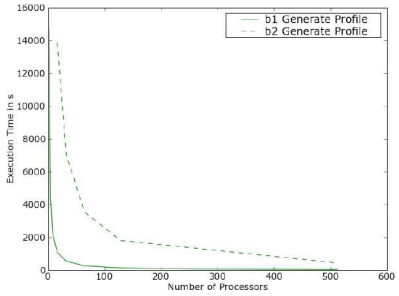
(a) Tipsy binary reader.



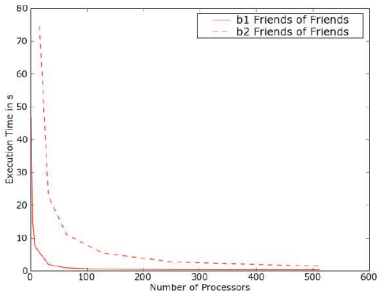
(b) *Principle Moments of Inertia* filter.



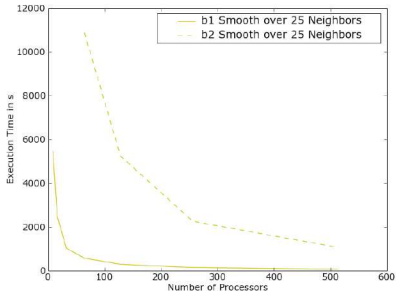
(c) *Center of Mass* filter.



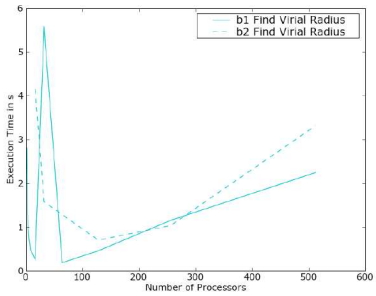
(d) *Profile* filter.



(e) *Friends-of-Friends Halo Finder* filter.



(f) *Neighbor Smooth* filter.



(g) *Virial Radius* filter.

Figure 13: Running time of various AstroViz filters versus processor number for GHALO B1 and GHALO B2.

lower resolution, slowly increasing with the number of processors.

6.3.2 Realtime rendering performance

Figure 14 depicts the performance of the ZBox3 CPU cluster and that of the Horus combined CPU and GPU cluster on a rendering benchmark that consists of a scripted realtime interaction sequence, meant to mimic typical tasks the user might do involving rotations of the geometry and zooming in scale. For the lower B1 resolution, peak rendering performance is at 28 CPUs or 28 CPUs and 28 GPUs for the respective clusters. For the higher B2 resolution, peak performance is achieved at 250 CPUs on ZBox3 or 28 CPUs/GPUs (the maximum available on Horus). While Horus shows a steady decrease in rendering time as CPUs/GPUs are added, ZBox3 shows there is a turnaround, after which adding more CPUs while keeping the problem size fixed is actually detrimental to the rendering speed. This turnaround would likely be seen in a CPU/GPU cluster as well, were enough nodes available. Overall, a modest increase in the number of CPUs or CPU/GPU pairs available over a single CPU or CPU/GPU pair shows a vastly improved performance. After this initial improvement, the returns are diminishing and in some cases nonexistent even as processing power is added.

Figure 15 depicts a close-up of the performance on the Horus cluster compared to the performance on the ZBox3 cluster. From this we see that when enough GPUs are available, namely more than 4 for the B1 resolution and more than 8 for the B2 resolution, performance of the CPU/GPU cluster is generally better than that of the CPU only cluster, as one might expect. This gain in only seen in a window however, after which the performance of the two becomes comparable as the number of CPUs or number of CPUs/GPUs is increased, respectively, and appear to asymptote to the same line for both resolutions. The increased rendering speed in the window in which the CPU/GPU cluster wins is modest. However, it can be up to 15% of the total rendering time which is significant for large rendering times.

6.3.3 Performance on a large number of particles

Comprehensive performance tests were not carried out for the highest resolution of GHALO on Horus or ZBox3 because the memory requirements inherent in visualizing and analyzing a 3 billion particle file made it prohibitive to reduce the number of nodes of ZBox3 used below 64. However, here I present a feasibility demonstration of usage of AstroViz features on the full resolution GHALO B3 given enough resources, namely 512 processors of ZBox3. The B3 file took a full 38 minutes to read in, a significant increase over the lower resolution files, but the AstroViz algorithms as well as the rendering benchmark did not show the same significant increase in running time. **Figure 16** depicts the histogram of read time versus number of processes of the B3 file being read across 512 processors. We see about half the processors took 38 minutes to read the file but the other half took less time. Further work would have to be done to investigate this disparity and its origins whether in hardware or in communication lag. But it is interesting to note that some processors managed to read their potion of the data a full four times faster despite the fact that each process is assigned

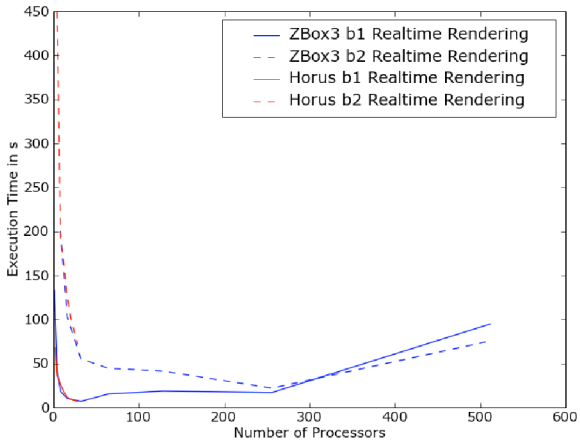


Figure 14: Comparison of performance on a CPU cluster (ZBox3) to a combined CPU and GPU cluster (Horus).

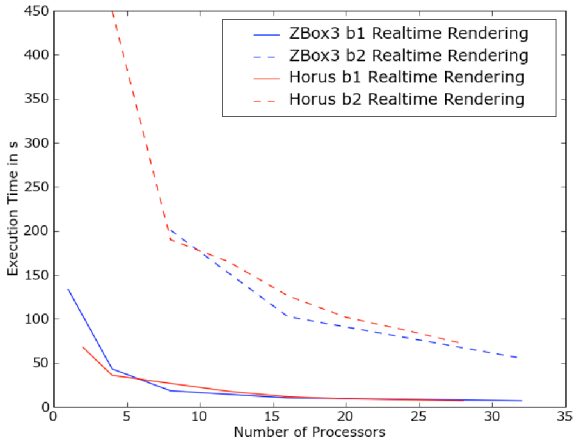


Figure 15: Close-up of the benchmark rendering performance on the combined CPU and GPU cluster Horus as compared to the CPU cluster ZBox3.

the same amount of data.

Figure 17 depicts the histogram of the time execution of of the *Principle Moments of Inertia* filter vs. number of processes executed on the B3 resolution across 512 processors. Here we can see there is not much variation–all processors finish between 8 and 9 seconds. For this number of processors, the *Principle Moments of Inertia* filter finished in .83 seconds for the intermediate GHALO resolution. Thus we see that increasing the problem size by an order of magnitude in this situation has increased the running time of this filter by an order of magnitude as well.

The rendering benchmark performance on 512 processors on the high resolution B3 file was 141.33 seconds, that of the B2 file was 75.64 seconds, and that of the B1 file was 94.94 seconds, not an order of magnitude from one resolution to the next in either case. However, if we compare to peak performance on the rendering benchmark on the B1 and B2 resolutions, we compare to 32 processors at 6.95 seconds and 256 processors at 22.08 seconds respectively. Here we see that by increasing the number of processors by an order of magnitude from the peak performance on B1 as we increase the file size by an order of magnitude, we again achieve optimal performance, which is approximately 3 times the optimal performance of the lower resolution file. From this we can hazard a guess that optimal performance on the rendering benchmark for B3 resolution would be achieved for approximately 2560 processors and would be approximately 68 seconds. This ballpark estimate agrees well with official ParaView recommendations to, in the case of unstructured data, try to have enough processors available such that there are 250,000-500,000 cells (particles in our case as there is one particle per cell) per processor with a maximum of 1,000,000 cells per processor. For 3 billion particles, as in the B3 resolution, this gives us the estimate that a minimum of 3000 processors should be used for optimal performance.

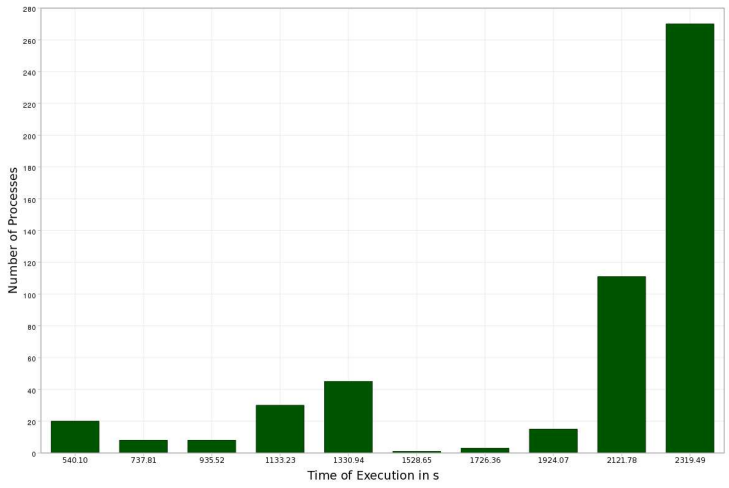


Figure 16: Read Time GHALO B3 on 512 Processors.

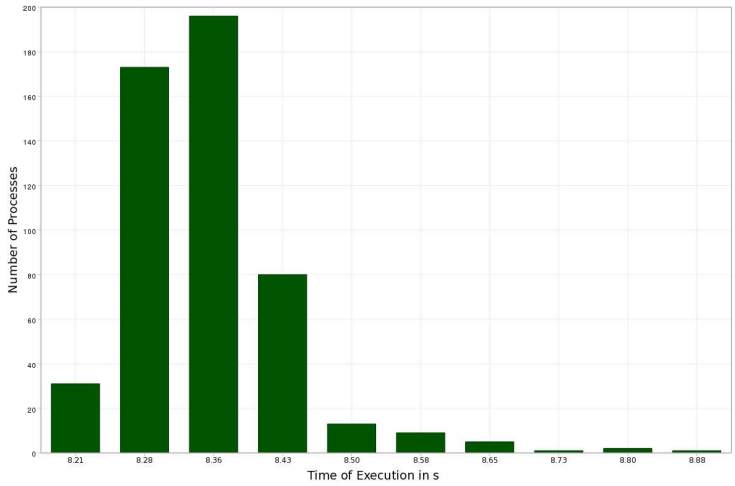


Figure 17: *Principle Moments of Inertia* filter executed on GHALO B3 on 512 Processors.

7 Further work

There is much potential for AstroViz to expand and refine its functionality. First and foremost is the incorporation of support for additional file formats to make AstroViz of use to a wider community. For the observational astronomy community, support for the FITS format is crucial, for the N-Body community, support for Gadget binaries is an oft requested feature, for the SPH and AMR communities a tighter integration with simulation output is necessary, and with the introduction of new simulation codes further formats may need to be added. Additionally, while ParaView supports reading in HDF5 files natively, it requires the user to first create an XML file describing the contents of the HDF5 file. This can be a barrier to entry for some users and can be done away with if something is known about which code produced the HDF5 file by automatically wrapping the HDF5 output of that code as part of a custom reader. Each AstroViz feature currently available will continue to function on the data read in by additional or external data and the need for additional functions will expand as the types of data available to be loaded increases and in turn increases the number of domains for which AstroViz is a useful analysis tool.

Secondly, AstroViz analysis tasks which are currently geared toward those involved in analyzing N-body simulations will be improved upon and expanded to enfranchise the SPH, theoretical cosmology, and observational communities. For the SPH community additional smoothing capabilities and techniques for volume rendering are important to include. For the theoretical cosmology communities, support for more advanced statistical operations such as generating correlation functions is a requested feature. For the observational astrophysics communities, support for pixel-by-pixel operations such as finding the summation of pixel values in a given area selected by the user, combining images taken at different wavelengths, correctly handling and utilizing metadata associated with the images, and support for finding connected regions of pixels which satisfy a certain functional distribution will be added. This area of development is especially exciting as the observational community, not accustomed

to dealing with issues in high performance computing on a daily basis, currently uses a smattering of tools which do not scale to larger images and more complex tasks, meaning that there is great potential for AstroViz to work with this community to quickly improve the observational astrophysics analysis workflow.

Thirdly, the analysis tasks which currently do not use ghost cells but formally require some handling of border process cells, namely the *Neighbor Smooth* and the *Friends-of-Friends Halo Finder* filters will be improved either by algorithmically doing away with the necessity to use ghost cells in the first place or by using ghost cells in the end where necessary.

Finally, efficiency is a primary consideration in high performance computing, both in time and in space. For the AstroViz *Profile* and *Neighbor Smooth* filters, running time can be long for a low number of processors as shown by the performance analysis graphs. This likely stems from some communications inefficiencies in the former case and an inefficient algorithm in the latter. Optimizations will be made in both cases. For the Tipsy binary reader and other filters which create additional data arrays, space is a consideration and giving the user more control over which variables to read in and which to compute during analysis tasks can mitigate wasting space with data arrays which the user knows he or she will not utilize during an analysis workflow. Each of these exciting developments in AstroViz–increasing formats handled, accuracy, the number of analysis functions available, and efficiency–is already underway as a component of AstroViz 2.0 which will be released in March 2010.

8 Conclusions

In this thesis I introduced AstroViz, a fully parallel open-source visualization and analysis tool. I reviewed the need for AstroViz in the astrophysical community, introduced the challenges faced in scientific visualization, and presented some theoretical background in data structures, root-finding algorithms, and parallel computing necessary to follow the details of the implementation of AstroViz features. I outlined the toolkits VTK and ParaView, upon which AstroViz is built, and described the theoretical and practical details of each AstroViz feature. Finally, I presented a collection of performance benchmarks and showed AstroViz analysis tasks performing well across a variety of problem sizes, number of processors available, and available cluster hardware.

By enabling the astrophysical community to have ready access to a visualization and analysis tool which is fully functional in parallel, the processing of higher resolution simulations than ever before is possible. As the resolution of simulations increases so does the ability of the simulation to capture the physics actually at play in reality. Thus the scientist visualizing and analyzing these simulations will have unprecedented insight into the physical phenomena. With the advent of a standard and scalable analysis and visualization tool freely available to the astrophysics community, the scientist will be able to fully immerse him- or her-self in the data concentrating on developing an understanding of the physical processes at play in the universe rather than on the technical machinery enabling said immersion.

For the full source code and documentation of AstroViz 1.0, details of the installation

process, and a comprehensive set of tutorials addressing usage of each AstroViz feature and setup on a client or a server machine, the reader is referred to the AstroViz website[4].

9 Acknowledgments

I would like to acknowledge Dr. Joachim Stadel and Doug Potter for many useful conversations which contributed to AstroViz's algorithmic design, Jean Favre and the Swiss National Supercomputing Center (CSCS) for providing use of the CPU/GPU cluster Horus, Lucy Moran and Jonathan Coles for providing editorial feedback, and finally Berk Geveci, the team at Kitware Inc., and the ParaView mailing list for help related to ParaView and VTK development and integration of AstroViz into the ParaView distribution.

References

[1] `http://www-hpcc.astro.washington.edu/tools/tipsy/tipsy.html`.

[2] `https://wci.llnl.gov/codes/visit/home.html`.

[3] `http://www.kitware.com/`.

[4] `http://www.itp.uzh.ch/~corbett/astroviz/astroviz.html`.

[5] Andrew Moore. A tutorial on kd-trees. Extract from PhD Thesis, 1991. Available from `http://www.cs.cmu.edu/simawm/papers.html`.

[6] L. S. Avila, S. Barré, R. Blue, D. Cole, B. Geveci, W. Hoffman, B. King, C. C. Law, K. M. Martin, W. J.Schroeder, and A. H. Squillacote. *The VTK User's Guide*. Kitware, Inc., Columbia, 2006.

[7] J. Binney and S. Tremaine. *Galactic Dynamics*. Princeton University Press, Princeton, NJ, 2008.

[8] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White. The evolution of large-scale structure in a universe dominated by cold dark matter. *Astrophysical Journal*, 292:371–394, May 1985.

[9] Mark de Berg. *Computational Geometry: algorithms and applications*. Springer, 2000.

[10] K. Heitmann, Z. Lukić, P. Fasel, S. Habib, M. S. Warren, M. White, J. Ahrens, L. Ankeny, R. Armstrong, B. O'Shea, P. M. Ricker, V. Springel, J. Stadel, and H. Trac. The cosmic code comparison project. *Computational Science and Discovery*, 2008.

[11] Steffen R. Knollmann and Alexander Knebe. AHF: AMIGA'S Halo Finder. *The Astrophysical Journal Supplement Series*, 182(2):608–624, 2009.

[12] C. Lacey and S. Cole. Merger Rates in Hierarchical Models of Galaxy Formation - Part Two - Comparison with N-Body Simulations. *R.A.S. MONTHLY NOTICES*, 271:676, December 1994.

[13] John F. Monahan. *Numerical Methods of Statistics*. Cambridge University Press, Cambridge, UK, 2001.

[14] Kenneth Moreland. *IceT Users' Guide and Reference*. Sandia National Laboratories, 2000.

[15] Brian W. O'Shea, Greg Bryan, James Bordner, Michael L. Norman, Tom Abel, Robert Harkness, and Alexei Kritsuk. Introducing Enzo, an AMR Cosmology Application. 2004.

[16] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit, 4th Edition*. Kitware, Inc., Columbia, 2006.

[17] Volker Springel, Naoki Yoshida, and Simon D. M. White. GADGET: A code for collisionless and gasdynamical cosmological simulations. 2000.

[18] Amy Henderson Squillacote. *The ParaView Guide*. Kitware, Inc., Columbia, 2007.

[19] J. Stadel, D. Potter, B. Moore, J. Diemand, P. Madau, M. Zemp, M. Kuhlen, and V. Quilis. Quantifying the heart of darkness with GHALO–a multi-billion particle simulation of our galactic halo. 2008.

[20] Joachim Stadel. *Cosmological N-Body Simulations and Their Analysis*. PhD thesis, University of Washington, 2001.

[21] P. J. Teuben, J. Stone, and T. Gardiner. Athena: a Grid-Based Code for Astrophysical Gas Dynamics. In *Astronomical Data Analysis Software and Systems XVI*, volume 376 of *Astronomical Society of the Pacific Conference Series*, page 93, October 2007.

[22] Romain Teyssier. Cosmological Hydrodynamics with Adaptive Mesh Refinement: a new high resolution code called RAMSES. 2001.

[23] Matthew Turk. Analysis and Visualization of Multi-Scale Astrophysical Simulations Using Python and NumPy. In *Proceedings of the 7th Python in Science Conference*, pages 46 – 50, Pasadena, CA USA, 2008.