CS87 Project Report:
Ray Marching 3-Dimensional Fractals on GPU Clusters

Jonah Langlieb, **K**ei Imada, **L**iam Packer
Computer Science Department, Swarthmore College, Swarthmore, PA 19081

December 16, 2018

Abstract

Fractals are self-similar mathematical structures which play an important role in chaos theory and which serve as visually stunning examples of the complexity that can stem from simple mathematical definitions. They are especially amiable for use with parallel computing because they only require the (embarrassingly parallel) iteration of a complex function at each point. However, with the advent of 3D 'mandelbulb' fractals, new techniques are required to more efficiently visualize these structures. Ray marching offers one such technique that combines the traditional approach of 3D ray tracing with mathematically-informed 'jumps' between time steps, allowing for drastically improved performance. In this paper, we explore the application of ray marching 'Mandelbulb' fractals with modern CUDA GPGPU programming, extended to run on the commodity GPU cluster available at Swarthmore College, in order to understand the trade-offs involved in large-scale CUDA programming. Our program, which uses C++ CUDA to interface with the GPU and MPI for inter-node commucation demonstrates robust scaling across resolution and number of nodes. Our hypotheses—that additional GPUs would improve run-time only when the image size was larger than a single GPU's memory and that partitioning which considered the load of each node would improve run-time—were both empirically supported by our data.

1 Introduction

Fractals are mathematical structures defined by self-similarity that serve as a foundational illustration of and model for chaos theory. Unlike calculus-based models which assumes that the closer an object is examined, the more similar it becomes to a smooth euclidean ideal, fractals are designed to stay 'rough' and complex at small scales. While the fundamental mathematical idea can be traced back earlier, the term 'fractal' was coined and popularized by Benoit Mandelbrot in his seminal 1982 work *The Fractal Geometry of Nature* which used fractals as a way of modeling natural processes, which similarly requires the rejection of simplification at small scales. Such modeling has found widespread and productive use in fields such as Biology [10] and Physics [9, 4], along with continued research in Mathematical Chaos Theory. And, because of the elegant and succinct equations which underlie them, they are particularly useful in Computer Graphics as astonishing illustrations of the complexity that can be achieved with computers.

One type fractal which became popular in the Computer Graphics field in the late 2000s is 3-dimensional (3D) fractals, especially the Mandelbulb fractal, a variant of the 2-dimensional (2D) Mandelbrot, which is generated using recursive iterations of an imaginary 'escape-time' function. Unlike 2D fractals, which are relatively straightforward to display because the value of the fractal can simply be calculated for each point on the

plane, 3D fractals require a different approach to efficiently display. The naïve approach would be to use standard graphics techniques, like ray tracing, to generate an image. Using ray tracing, rays of light are extended in incremental constant-time steps from the 'eye's' viewpoint until they hit a part of the object, simulating the way we naturally see. One advantage of this approach is that it does not require rendering any non-visible section of the object. However, for fractals this approach is computationally expensive (though embarrassingly parallel) and memory inefficient because, unlike typical graphical constructs which are simple enough to make it easy to detect if a point is on the objects surface, the calculation for a fractal at each timestep to determine whether a point is on it is *itself* computationally expensive. Therefore, another approach is necessary. One such approach, ray marching, was pioneered as early as 1989 by Hart et al. [2] and extensively refined by Inigo Quilez [7]. In this approach, analytically-derived distance functions are used which provide an estimate for the maximum distance a ray can go *without* reaching the object. Using these functions, instead of ray tracing in constant-time steps, we can ray march in variable-sized 'jumps' for this maximum distance. This dramatically improves performance.

Because ray marching can efficiently take advantage of ray tracing and because of its embarrassingly-parallel nature, we thought it could be applied to graphical processing units (GPUs). The architecture of these external units are designed for (and force) embarrassingly-parallel tasks and fractal generation has a long tradition of taking advantage of them [5]. And, in recent years, GPUs have become commonplace enough that many commodity machines (laptop and desktop) include one. With this prevalence, we wanted to use (commodity) GPU clusters to further take advantage of the parallelism inherent in Mandelbulb generation. Our work explores how to parallelize 3D fractal rendering on a commodity GPU cluster such that we maximize the speedup and image resolution. In this experiment, we used the networked, GPU-equipped computers of the Swarthmore College Computer Science Department which has both heterogeneous nodes and, due to broad student use, widely varying load.

We used a custom CUDA C++ program to interface with the GPUs and MPI to distribute the sub-tasks and communicate between nodes, In order to measure different aspects of the parallelized runtime, we modified the resolution of the output image, the number of nodes, and the partitioning scheme with which we split up the nodes work. We hypothesized that, due to the tradeoff between communication costs and limited memory, the GPU cluster would have a faster runtime only when the resulting image was larger than a single GPUs memory. Additionally, due to the heterogeneity of nodes, we expected a partitioning scheme that respected such differences would out-perform a simple 'even' partitioning. Supporting our hypothesis, we found that our solution scales well with additional GPUs, especially when the resolution exceeds the capacity of the GPUs memory. Additionally, the load-respecting partition was markedly faster than the even partition.

2 Related Work

We were particularly inspired by Hart's 1989 paper *Ray Tracing Determinstic 3-D Fractals* which ray marched the Julia Set (another family of 3D fractals) in order to render highly detailed images, even on constrained hardware [3]. This is this one of the first papers to apply ray marching (which they call 'unbounding volumes') to fractals and is an especially good primer to understanding more-complicated contemporary ray marching algorithms. Additionally, analogous to our work, they used the *AT&T Pixel Machine*, a predecessor of modern GPUs, which consists of 64 parallel processors dedicated to graphics processing [6]. Even though the processing speed is quite different ($\approx$ 1 hour for a 1280 x 1024

image) and their memory much more physically constrained, we found their techniques helpful in optimizing our own algorithm for speed and memory consumption. Additionally, it is always inspiring and humbling to read a paper almost three decades old which remains exceptionally relevant to modern computer science.

Additionally, the work of Inigo Quilez, who championed the use of ray marching in graphics, lays the foundation of this work, not only for ray marching in general, but also for the Mandelbulb. He has many blog posts about how to render the Mandelbulb [7] and includes a fully-functioning web-based version of his code [8]. This online version was especially invaluable in implementing our code. He also demonstrates extra optimizations, such as color and rotations.

Additionally in *Fractal Art Generation Using GPUs*, Mayfield et al. help motivate much of our experiments into the speed-up possible by ray marching fractals with their analysis of 2D fractals. Their analysis of the GPU vs CPU speedup fractal generation was helpful in understanding our own speed-up tradeoffs, even though they only used 1 GPU [5].

3 The Problem and Solution

3.1 The Problem

The problem that we have explored is that of 3D fractal generation through ray marching for large-scaling resolutions. Ray marching is an extension of ray tracing, where instead of checking whether or not a vector has hit the surface at some arbitrary set of distances, it uses a distance function $d(\vec{x})$, where $\vec{x}$ is a vector that indicates the current position of the ray. This distance function then calculates the largest possible sphere around the point $\vec{x}$ such that the surface that is not within this sphere. Based on a predefined ε that is an arbitrarily small value, the distance function "hits" the surface when an unbounding sphere of size $\leq \varepsilon$ is found. That is to say, when the distance function tells us that we are very close to the surface, we consider the ray on the surface. In the case of the "Mandelbulb" fractal, the details of the distance function are beyond the scope of this paper but has been previously derived by other posts and papers [2].

While ray marching is an embarrassingly parallel problem that can be implemented on a single GPU for fast ray marching on relatively small resolutions, the problem of computing large resolutions (On the scale of $\geq 2^{16} \times 2^{16}$) is not fit for a single GPU. This is due to the limited memory of a single GPU, along with the limited number of possible threads to assign to different resolution indices. We propose a scalable solution to this problem.

3.2 Our Solution

Our solution to this problem of large resolution ray marching is to implement a CUDA/MPI program that utilizes the messaging capabilities of MPI to assign indices of a large resolution to various nodes in the network with usable GPUs. This remedies the problem of limited GPU memory, since a cluster scales linearly with the number of nodes.

More specifically, we are leveraging the Open-MPI abilities of the Swarthmore Computer Science labs to use various connected lab machines with GPUs to perform these large computations. Note that the GPUs provided by Swarthmore have drastically different computational power and, due to general student use, constantly fluctuating use. Due to the large-scale nature of our computation, we used the higher-end GPUs for the majority of our computation. For a certain run of the computation for an image size of $n \times n$, we create a one-dimensional array of length $n \cdot n$ of integers. We can then partition this in two different ways.

The simpler of the two is partitioning the n^2 length array into m partitions, with the partition $\frac{n}{m} \cdot m_0$ to $\frac{n}{m} \cdot (m_0 + 1) - 1$ being assigned to machine m_0, where each machine is arbitrarily assigned a number to establish an ordering in the set of machines. We call this the 'even

partitioning' scheme.

The more complex of the two assumes that GPU load is correlated to the amount of data stored in the GPU itself. Let n be the number of GPUs there are and f_i be the free memory of the ith GPU. For each node, we calculate the *partition percentage*, denoted x_i, which is calculated by

$$x_i = 100\frac{f_i}{\sum_{i=1}^{n} f_i}.$$

We then partition the computation such that the ith node gets $x_i\%$ of the total data. We call this the 'free memory load partitioning' scheme.

After the computation of the resulting image is finished, we have an array of length n^2 with each index $A[i]$ being assigned a value based on the iteration that the computation found an unbounding sphere with radius less than some arbitrarily small ε, as was previously talked about in the general ray marching algorithm. The array is then fed into a library to write to a `.png` file with a name that identifies which machine in the set of machines wrote that image so as not to create duplicates.

Since the lab machine file system is NFS, we then wait for the file system to process the creation of these files in the respective sub-directory to the program. After these files are written across the network, we use ImageMagick's `convert` command-line tool to append these files together to create the final image for the sake of pretty fractals and verification of our results as can be seen in 1.

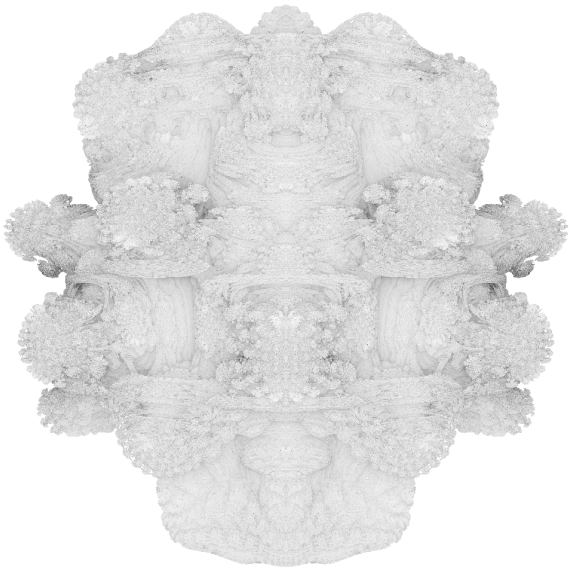

Figure 1: 1024x1024 fractal from our program's computation. This image used the 10th iterations of the fractal

4 Results

After finishing our implementation of ray marching 3D Fractals, specifically the Mandelbulb, through an MPI/CUDA interface, we gathered results based on general relationships between resolution size and computation time, even partitioning and load-based partitioning, and of homogeneous clusters and heterogeneous clusters.

The gathering of data was done by timing each run of computation on a certain sized homogeneous or heterogeneous cluster. The sizes of each cluster ranged from 1 up to 16 in increments of powers of two. We refrained going above 16 nodes to prevent large disruption of the Computer Science lab machines. We also decided to limit total computation time to an upper-bound of 600 seconds. This was to minimize disruption across the most widely used lab (Clothier) among students. There was also the issue of finding many GPUs of the same architecture. In the heterogeneous cluster, we decided to use a hostfile comprised of 16 Quadro M1000M, 8 Quadro 1000M, 4 GeForce GTX 1080, and 4 GeForce GTX 750. We included over 16 hosts in the case that some hosts spontaneously fail.

These times were then saved to a `.csv` file with rows containing information regarding resolution, nodes, partitioning method (0 for even, 1 for balancing), homogeneity (0 for homogeneous, 1 for heterogeneous), trial number of a specific run, time taken to partition the data, and time

taken to compute the fractal. For the purpose of consistency, we decided to do 3 trials for each parameter list to obtain a number of computation times. We averaged these results if the run did not fail or exceed 600 seconds, which would result in a timeout and computation time of -1. This table was then analyzed to produce the following results.

4.1 Resolution vs. Computation

We found that based on our results, a second-order regression found that $C \propto R^2$, where C is the computation time for a run, and R is the resolution of that run. This makes perfect sense, since in order to compute the desired pixel value for each pixel in the resolution, there are R^2 pixels in the resolution, resulting in a computation time relating to R^2. This result was consistent with both heterogeneous and homogeneous cluster runs.

Figure 2 shows this exact relationship on a $2-16$ node heterogeneous cluster with even partitioning up to resolution $16,384$. The curve was fit to a second-order regression with an average p-value on the order of 10^{-11}.

One interesting result that can be seen from figure 2 is that on the heterogeneous cluster, the 4-node and 2-node clusters outperformed the 16-node and 8-node clusters. This is likely due to the architecture of these 2 and 4 node clusters. This can most likely be attributed to the fact that they are comprised of only NVIDIA GeForce GTX 1080 GPUs, known to be considerably faster than the other GPUs running on the lab machines. The clusters that are comprised of more than 4 nodes use lower powered GPUs for the partitioned data. These results show that even a 16-node cluster with 4 GTX 1080s and 12 non-GTX 1080s performs worse on the same resolution image than a 4-node cluster of GTX 1080s. On average, this 4-node cluster performed the same job in 59.77% of the computation time for that of a 16 node cluster, and 67.99% faster than that of an 8 node cluster. The results of this are shown in 1, where the runtimes of the 8

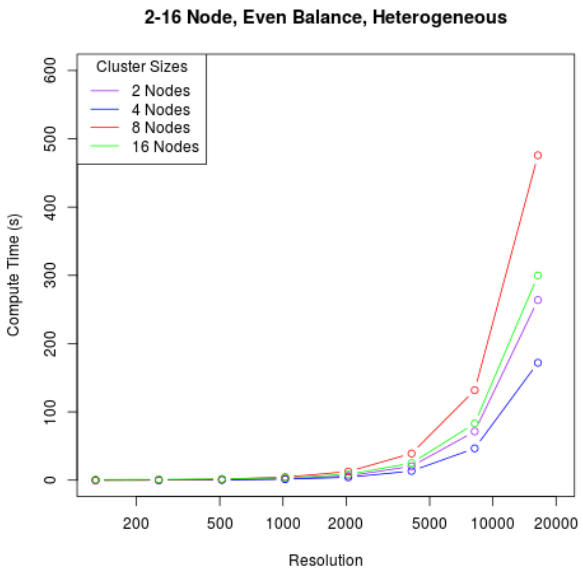

Figure 2: Computation Time (s) vs Resolution for heterogeneous, evenly partitioned nodes. *The relationship shown fits that of $C \propto R^2$, as is expected of this algorithm*

node and 16 node clusters are used to determine the speedup of the 4 node cluster, beginning at resolution 1024.

4.2 Speedup

We also measured the speedup from a 1 node run to each of the multiple-node runs on a heterogeneous cluster with even partitioning. Figure 5 shows the speedup of the various cluster sizes ranging from 2 nodes to 16 nodes with resolutions up to 4096. The reason larger resolutions are not used to measure speedup is due to the initial condition we set, where the longest computation cannot exceed 600 seconds, and due to how slow a single node is compared to multiple nodes, 4096 was the highest resolution a single node could compute in under 600 seconds. As with before, the heterogeneous 4-node cluster outperformed each of the other node clusters due to the concentration of high-powered GPUs on the cluster.

To confirm our intuition, we also provide the

Resolutions	16 Node	8 Nodes	4 Nodes	16 Node Speedup	8 Node Speedup
1024	2.93 s	3.75 s	1.19 s	2.44	3.13
2048	7.59 s	11.09 s	3.58 s	2.11	3.09
4096	23.34 s	36.97 s	12.74 s	1.83	2.90
8192	84.55 s	133.88 s	47.23 s	1.79	2.83
16384	299.50 s	475.39 s	171.82 s	1.74	2.76

Table 1: Comparison of Speedups of 4 Node Cluster. *4 Node GTX 1080 clusters run significantly faster due to the power of the smaller cluster. The speedup for each resolution starting at resolution* 1024 *is shown above.*

speedup of the homogeneous cluster runs in figure 6, which is the same setup as before except the cluster is of homogeneous makeup. Here there is no difference in any of the nodes computation power due to it being a homogeneous cluster, therefore the results reflect our intuition that the 16-node cluster will have the largest speedup. Adding more nodes to perform computations clearly has a positive effect on the runtime as the image size increases to unmanageable sizes for a single-node GPU.

4.3 Partitioning

In comparing the two different partitioning schemes, we hypothesized that the *free memory load partitioning* scheme would perform better in a heterogeneous cluster of higher node values. This is due to the fact that this partitioning scheme partitions more of the image to compute to nodes with more free memory. As stronger GPUs tend to have more free memory, this allows more computation to be performed on more powerful GPUs, significantly speeding up the computation time.

This hypothesis was reflected in our results when comparing the two partitioning schemes with computation runtime as the metric. Figure 7 shows the box plots of each computation time spread for resolutions 256 up to 16384 iterating in powers of two for both even partitioning on a heterogeneous cluster, and load partitioning on a heterogeneous cluster. It is clear that load partitioning has reduced the overall runtime of the 16 node cluster, which is especially prevalent in larger resolutions. This is also reflected in figure 4, where the lineup of computation time roughly matches what we would expect thanks to the free memory load partitioning scheme. As we have stated in previous sections, this partitioning scheme in practice greatly helped the runtime of larger heterogeneous clusters, which shows the promise of providing a more clever partitioning algorithm to minimize computation time.

5 Conclusions and Future Directions

We have implemented a 3D fractal rendering software that uses ray marching technology and runs on GPU clusters. We chose fractals due to the large amounts of computation power necessary to perform ray marching on such a surface, the beautiful detail that results from these computations, and the embarrassingly parallel nature of the computation.

We found our implementation to successfully capture our goals for this project. We saw promising results in the power of ray marching through a GPU cluster, allowing for much larger resolutions of images to be calculated in lower increments of time due to the sheer abundance of resources in clusters. This is especially reflected in figure 6 and 5, where speedup only increases as the resolution sizes scale to very large values. To further optimize the implementation, the load partitioning scheme proved very useful

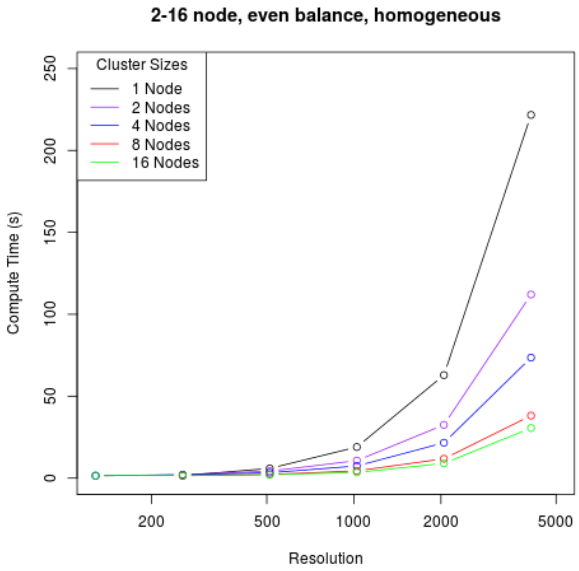

Figure 3: Graph of Computation Time (s) vs Resolution for Homogeneous Clusters. *The Homogeneous cluster displays the same relationship, but the overall cluster power is lower due to the less powerful GPUs, resulting in longer computation overall*

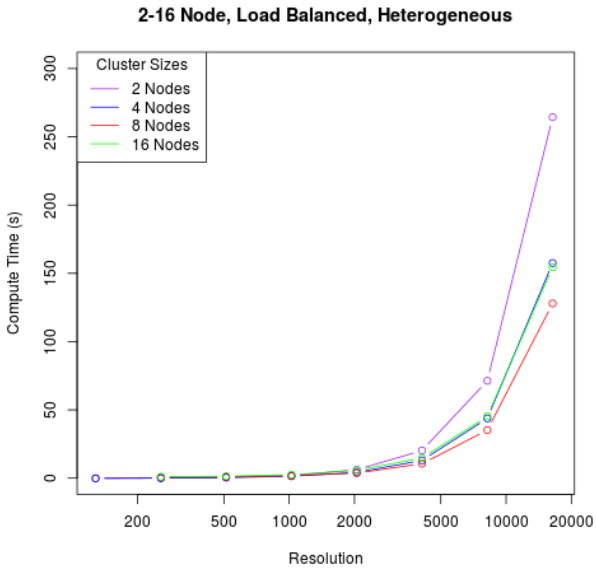

Figure 4: Graph of Computation Time (s) vs Resolution for Heterogeneous, load partition clusters. *Here we see again that higher-node clusters see a speedup when compared to smaller clusters in terms of runtime. This is thanks to a more efficient usage of resources thanks to a partitioning scheme that takes into account the power of the varied GPUs in the cluster.*

as shown from figure 7, where the downsides of a heterogeneous cluster, such as having wasted resources on powerful nodes, are diminished by taking into account the resources available at each node. While this may take $O(n)$ time with a naïve implementation where n is the number of nodes, this will pale in comparison to the runtime of a very large resolution image, showing the importance of accounting for valuable resources.

5.1 Extensions

There are various ways our project could be extended—we have come up with five possible extensions for this project: support for nodes with multiple GPUs, development of large scale distributed image stream processing software, shaders based on other fractal computation algorithms, explorations of other partitioning schemes, and creation of 3D fractal animations.

5.1.1 Multi-GPU Nodes

Our program assumes that the cluster has one GPU per per node. Many nodes on XSEDE have more than one GPUs. If we can leverage these multi-GPU nodes by using the `cudaGetDeviceCount` command in CUDA, we could potentially observe a considerable amount of speedup. As a result, the method of partitioning based on simply node ordering in the set of nodes would need to be modified.

This could potentially turn into a hashtable of different GPUs to different nodes that then get ordered and the method should not change much. If the number of GPUs per node is not uniform, then the communication of the locations of all GPUs would need to happen in at most linear time, potentially in logarithmic time with a map-reduce implementation.

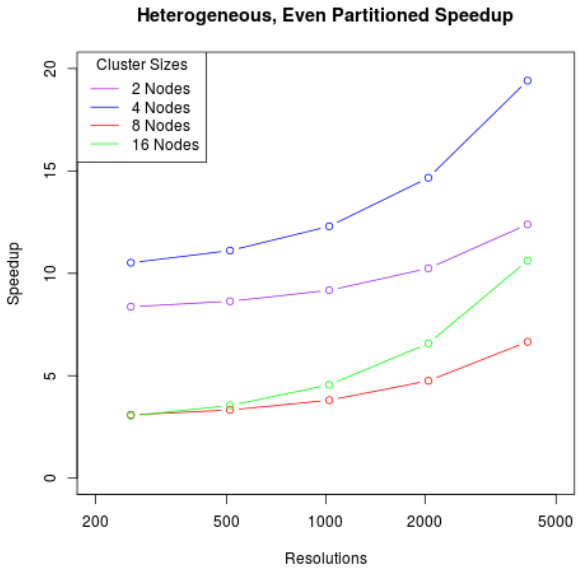

Figure 5: Speedup of multiple-node runs for heterogeneous, evenly partitioned resolution. *Note in this case, since the 4-node cluster is comprised of very powerful GPUs, the especially fast computation time observed in section* 4.1.

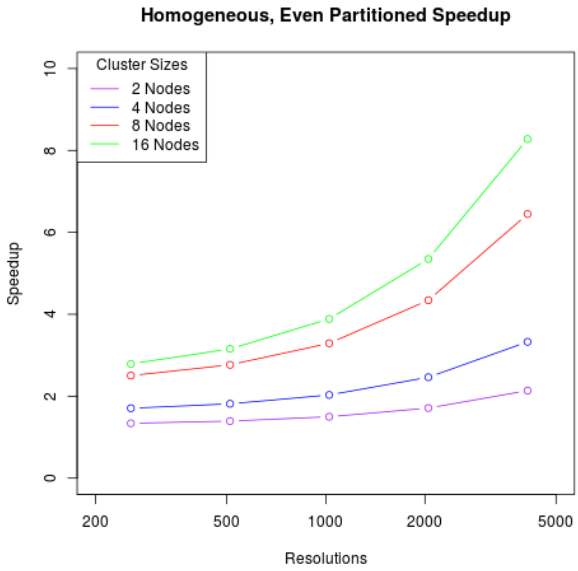

Figure 6: Speedup of multiple-node runs for homogeneous, evenly partitioned resolution. Results are compared to a single-node cluster to measure speedup.

5.1.2 Large Scale Distributed Image Stream Processing Software

We only measured the partitioning time and the compute time in our software because the main time bottleneck of the whole software was the write to the NFS and the appending of the large images. Furthermore, our `convert` command to append all the images gave errors when the images were too large. To address these issues, we suggest developing or utilizing existing large scale image stream processing softwares to join large images. To remedy the issues brought by NFS, this program could be run on a tightly-coupled architecture with a powerful parallel file system such as `lustre` that XSEDE runs [1] to reduce the network penalties.

5.1.3 Shading Using Other Fractal Computation Algorithms

In regards to estimating 3D fractals, ray marching is superior to ray tracing and the escape-time algorithm in various ways—including space usage and time consumption. One major downside of ray marching, however, is the loss of wondrous colors that escape-time algorithms give. Our program only colors the final image in monochrome. We could instead enjoy vivid shadings of 3D fractals. Thus, in order to give life to these fractals, for future work, we suggest streaming the computed points and the neighboring points around the computed points (perhaps the points within distance of ε away) and feeding them into the escape-time algorithm. This would not only alleviate the space consumption problem given by the escape-time algorithm, but would also give us the exact location of the surface points on the 3D fractal. We have also come across alternative ways of coloring the 3D fractals. Orbit traps is one way of coloring frac-

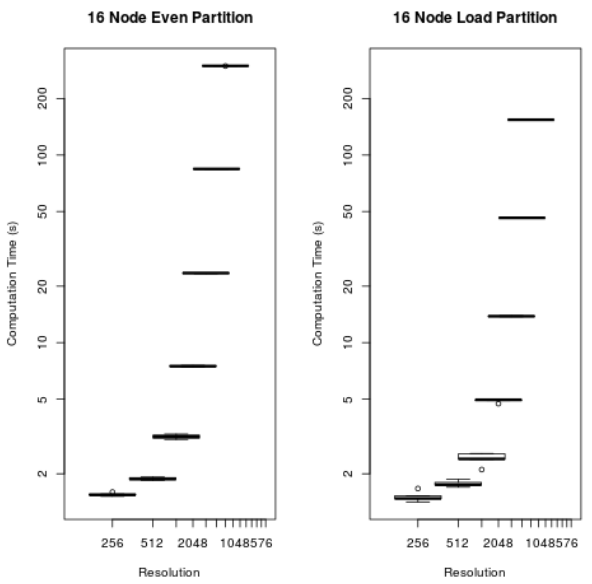

Figure 7: Box plots of the two partitioning methods used on a 16-node heterogeneous cluster. X-axis scaled logarithmically to reflect iteration steps. *The computation time for the even partitioning at higher resolution values increases compared to the free memory load partitioning scheme as resolution increases.*

tals where one chooses a static geometric object and then keeps track of how close the orbit comes to the object. Implementing orbit traps in our program is in our plans for future projects.

5.1.4 Partitioning

We have explored two different types of data partitioning in this paper: even distribution of data and memory load distribution. In the future, we intend to implement other types of partitioning schemes. One that we have in our minds is the *spill-over* partitioning scheme where we first sort the nodes based on the amount of free GPU memory, and then use the smallest number of nodes. If communication costs between nodes is expensive, we believe the spill-over partitioning scheme will perform faster than the two partitioning schemes we investigated in this paper, because the spill-over partitioning scheme uses the smallest number of nodes by definition, and thus will have coarser granularity of communication.

5.1.5 3D Fractal Animations

There exists transformation distance-estimation functions that we could use for transforming our 3D fractal in many ways, which include but not limited to: rotations, translations, scaling, twisting, and bending. We also know that rendering consecutive frames of a video is embarrassingly parallel. Thus, for another future project, we can render multiple frames of an animation in parallel, perhaps distributed across multiple nodes with GPUs.

6 Meta-discussion

6.1 Makefiles

The compilation process for this project was unfamiliar to us from the outset due to most labs up to this point being written in C/C++ also came with a predefined Makefile. We were put to the task of mapping out the dependencies between different files and classes, and as a result made sure that functions were defined properly in the appropriate scope and no re-definitions were possible in the compilation process. Finally, we needed to ensure that `make clean` removed the respective `.o` and `.png` files. We found that an effective way of doing this was by searching for the files with those extensions at clean-time and removing them.

When moving from simply CUDA to a combination of CUDA/MPI, the task was then to construct a makefile that properly links the two different compilers together with regards to the respective files that included any CUDA or MPI defintions. To accomplish this, we decided on changing the backend compiler of `nvcc` through the `-ccbin=` flag.

6.2 Asynchronous CUDA

This was a very subtle problem we had. Initially, we timed the computation of the fractal by wrapping the call in `gettimeofday()` calls which worked fine in previous labs that did not utilize CUDA. The issue with this is that CUDA kernel calls are asynchronous, therefore this `gettimeofday()` block wrapping the kernal call simply measures the runtime of setting up the kernel, rather than the actual runtime of the kernel. To remedy this, we used the function `cudaDeviceSynchronize()` to force the system to wait until the kernel call was complete, finally yielding correct timings. Before finding these correct timings, every resolution and node number yielded approximately the same runtimes for the kernel call which greatly confused us.

6.3 Threading Issues

In the implementation process of our project, we found in the initial CUDA-only implementation stages of development that the GPUs will silently fail if too many threads or resources are requested for the execution of the program. This problem would arise on a single GPU for resolutions higher than 8192×8192. The form that the failure took on was seemingly instantaneous calculation time, a clear indicator that something has gone wrong. Upon calling `cudaGetLastError()`, which forces the last error found on the GPU to be described, we found that the reason for failure was a combination of too many threads requested to compute and numerous out-of-bounds errors on calculating a thread's ID in the grid-block-thread architecture of the GPU. After fixing these errors, things went smoothly thereon.

6.4 Experiment Design

Originally, we also intended to look at the time it takes for the array to be processed into a `.png` file and then the time taken to convert each of the individual `.png` files produced by separate nodes. We found this to be out of line with the goals of this paper since it is not a measure of our own process, it is merely timing other people's work for their processes. If we had the time to also produce a large scale distributed image processing software or program, then this would be a fine measure, but in this case we scrapped that dimension of testing.

6.5 Changes from Proposal

In our original proposal, we anticipated focusing largely on implementing an escape-time algorithm that would allow is to sparsely generate 3D fractals based purely on analysis. Instead, we decided to approach the computation and visualization of fractals through ray marching based on known distance functions. We decided this because we felt that this approach would be more efficient and simpler in a number of ways. For one, ray marching only generates points which would be visible by a certain viewpoint rather than generating the entire surface. This approach also easily coupled the two difficulties of computation and visualization far better than pure calculation on a point-by-point basis. If an escape-time algorithm were used for a huge number of points in 3D space, there would also be the problem of verification of the algorithm, which would presumably be done by further analysis, as well as the visualization of the fractal.

We also decided to scrap the 2D implementation of the Mandelbrot fractal. This was due to the fact that this has been rigorously researched and implemented in countless architectures and programming paradigms, including CUDA.

Since we decided on the ray marching approach, the idea of writing a number of encoded values to represent points that are "on" the fractal surface was also scrapped since ray marching itself produces a visualization that must simply be converted to a readable format. This is reminiscent of the Fire Simulator lab, where all that needed to be done was to feed an array of numbers to a kernel or library that would be able to write this array to an appropriate format or produce a visualization on the fly.

References

[1] Stephanie Donovan, Linux Symposium, Andi Kleen, Matthew Wilcox Hewlett-packard, Gerrit Huizenga Ibm, Andrew J. Hutton, Martin K. Petersen, Wild Open Source, and Philip Schwan. Lustre: Building a file system for 1,000-node clusters. 08 2003.

[2] John C Hart, Daniel J Sandin, and Louis H Kauffman. Ray tracing deterministic 3-d fractals. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 289–296. ACM, 1989.

[3] John C Hart, Daniel J Sandin, and Louis H Kauffman. Ray tracing deterministic 3-d fractals. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 289–296. ACM, 1989.

[4] Magdalena Malankowska, Stefan Schlautmann, Erwin J. W. Berenschot, Roald M. Tiggelaar, Maria Pilar Pina, Reyes Mallada, Niels R. Tas, and Han Gardeniers. Three-dimensional fractal geometry for gas permeation in microchannels. *Micromachines*, 9(2), 2018.

[5] Will Mayfield, Justin Eiland, Taylor Hutyra, Matt Paulsen, Bryant Wyatt, et al. Fractal art generation using gpus. *arXiv preprint arXiv:1611.03079*, 2016.

[6] Michael Potmesil and Eric M Hoffert. The pixel machine: a parallel image computer. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 69–78. ACM, 1989.

[7] Inigo Quilez. Mandelbulb. `http://www.iquilezles.org/www/articles/mandelbulb/mandelbulb.htm`.

[8] Inigo Quilez. Mandelbulb. `https://www.shadertoy.com/view/ltfSWn`.

[9] Yu-Te Wu, Kuo-Kai Shyu, Tzong-Rong Chen, and Wan-Yuo Guo. Using three-dimensional fractal dimension to analyze thecomplexity of fetal cortical surface from magnetic resonance images. *Nonlinear Dynamics*, 58(4):745, Apr 2009.

[10] Luduan Zhang, Jing Z. Liu, David Dean, Vinod Sahgal, and Guang H. Yue. A three-dimensional fractal analysis method for quantifying white matter structure in human brain. *Journal of Neuroscience Methods*, 150(2):242 – 253, 2006.