Extreme Scaling of Production Visualization Software on Diverse Architectures

Hank Childs* David Pugmire and Sean Ahern† Brad Whitlock‡
Mark Howison, Prabhat, Gunther Weber, and E. Wes Bethel§

Lawrence Berkeley National Laboratory
Oak Ridge National Laboratory
Lawrence Livermore National Laboratory

Abstract

We present the results of a series of experiments studying how visualization software scales to massive data sets. Although several paradigms exist for processing large data, we focus on pure parallelism, the dominant approach for production software. These experiments utilized multiple visualization algorithms and were run on multiple architectures. Two types of experiments were performed. For the first, we examined performance at massive scale: 16,000 or more cores and one trillion or more cells. For the second, we studied weak scaling performance. These experiments were performed on the largest data set sizes published to date in visualization literature, and the findings on scaling characteristics and bottlenecks contribute to understanding of how pure parallelism will perform at high levels of concurrency and with very large data sets.

1 Introduction

Over the last decade, supercomputers have increased in capability at a staggering rate. Petascale computing has arrived and machines capable of tens of petaflops will be available in a few years. No end to this trend is in sight, with research in exascale computing well underway. These machines are primarily used for scientific simulations that produce extremely large data sets. The value of these simulations is the scientific insights they produce: these insights are often enabled by scientific visualization. However, there is concern whether visualization software can keep pace with the massive data sets simulations will produce in the near future. If this software cannot keep pace, it will potentially jeopardize the value of the simulations and thus the supercomputers themselves.

For large data visualization, the most fundamental question is what paradigm should be used to process this data. The majority of visualization software for large data, including much of the production visualization software that serves large user communities, utilizes the pure parallelism paradigm. Pure parallelism is brute force: data parallelism with no optimizations to reduce the amount of data being read. In this paradigm, the simulation writes data to disk and the visualization software reads this data at full resolution, storing it in primary memory. As the data is so large, it is necessary to parallelize its processing by partitioning the data over processors and having each processor work on a piece of the larger problem. Through parallelization, the visualization software has access to more I/O bandwidth (to read data faster), more memory (to cache more data), and more compute power (to execute its algorithms more quickly).

In this paper, we seek to better understand how pure parallelism will perform on more and more cores with larger and larger data sets. How does this technique scale? What bottlenecks are encountered? What pitfalls are encountered with running production software at massive scale? Will pure parallelism be effective for the next generation of data sets?

*e-mail:hchilds@lbl.gov
†e-mail:pugmire,ahern@ornl.gov
‡e-mail:whitlock2@llnl.gov
§e-mail:mhowison,prabhat,ghweber,ewbethel@lbl.gov

These questions are especially important because pure parallelism is not the only data processing paradigm. And where pure parallelism is heavily dependent on I/O bandwidth and large memory footprints, alternatives de-emphasize these traits. Examples include in situ processing, where visualization algorithms operate during the simulation's run, and multi-resolution techniques, where a hierarchical version of the data set is created and visualized from coarser to finer versions. With this paper, however, we only study how pure parallelism will handle massive data.

We performed our experiments using only a single visualization tool, "VisIt," though we do not believe this limits the impact of our results. This paper's aim is to understand whether pure parallelism will work at extreme scale, not to compare tools. When a program succeeds, it validates the underlying technique. When a program fails, it may indicate a failing in the technique or, alternatively, a poor implementation in the program. The principal finding of this paper is that pure parallelism at extreme scale works, that algorithms such as contouring and rendering performed well, and that I/O times were very long. Therefore the only issue that required further study was I/O performance. One could envision addressing this issue by studying other production visualization tools. But these tools will ultimately employ the same (or similar) low-level I/O calls, like `fread`, that are themselves the key problem. So rather than varying visualization tools, each of which follow the same I/O pattern, we varied I/O patterns themselves (meaning collective and non-collective I/O) and also compared across architectures and filesystems.

This paper is organized as follows: after discussing related work in section 2, we present an overview of pure parallelism in section 3. We then discuss our trillion cell experiments in section 4, followed by the results of a weak scaling study in section 5. The final section describes pitfalls that we encountered in our efforts.

2 Related Work

We are aware of no previous efforts to examine the performance of pure parallelism at extreme scales on diverse architectures. However, other publications provide corroboration in this space, albeit as individual data points. For example, Peterka et al. [9] demonstrate a similar overall balance of I/O and computation time when volume rendering a 90 billion cell data set on a BG/P machine.

There are paradigms besides pure parallelism, such as in situ processing [6, 5], multi-resolution processing [3, 8], out-of-core [12], and data subsetting [1, 10]. Framing the decision of which paradigm should be used to process massive data as a competition between pure parallelism and the others is an oversimplification. These techniques have various strengths and weaknesses and are often complementary. From the perspective of this paper, the issue is whether pure parallelism will scale sufficiently to process massive data sets.

Our study employed the VisIt visualization tool [2], which primarily uses pure parallelism, although some of its algorithms allow for out-of-core processing, data subsetting, or in situ processing[1]. ParaView [7], another viable choice for our study, also relies heavily on pure parallelism, again with options for out-of-core processing, data subsetting, and in situ visualization. The end users of these tools, however, use pure parallelism almost exclusively, with the other paradigms often used only situationally. Both tools rely on the Visualization ToolKit [11]. VTK provides relatively small memory overhead for large data sets, which was crucial for this study (since data sets must fit in memory), and especially important given the trend in petascale computing towards low memory machines. Yet another viable choice to explore pure parallelism would have been the commercial product EnSight [4], with the caveat that measuring performance accurately would have been more difficult with that tool.

3 Pure Parallelism Overview

Pure parallelism works by partitioning the underlying mesh (or points for scattered data) of a large data set among cores/MPI tasks. Each core loads its portion of the data set at full resolution, applies visualization algorithms to its piece, and then combines the results, typically through rendering. In VisIt, the pure parallelism implementation centers around data flow networks. To

[1] The experiments in this paper used pure parallelism exclusively.

satisfy a given request, every core sets up an identical data flow network, differentiated only by the portion of the whole data set that core operates on.

Many visualization algorithms require no interprocess communication and can operate on their own portion of the data set without coordination with the other cores (i.e. "embarassingly parallel"). Examples of these algorithms are slicing and contouring. There are important algorithms that do require interprocess communication (i.e. not "embarassingly parallel"), however, including volume rendering, streamline generation, and ghost data generation[2].

The pure parallelism paradigm accommodates both types of parallel algorithms. For "embarrassingly parallel" algorithms, it is possible to have each core directly apply the serial algorithms to its portion of the data set. And pure parallelism is often the simplest environment to implement non-embarrassingly parallel algorithms as well, since every piece of data is available at any given moment and at full resolution. This property is especially beneficial when the order of operations is data dependent (streamlines) or when coordination between data chunks is necessary (volume rendering, ghost data generation).

After the algorithms are applied, the results of those algorithms are rendered in parallel. This rendering algorithm serves to collect the results of all the cores into a single result, as if all of the data was rendered on a single core. Although this algorithm doesn't scale perfectly, it is known to scale well with an efficient $\mathcal{O}(n \log n)$ reduction algorithm.

Pure parallelism is typically run with one of two hardware scenarios: (i) on a smaller supercomputer dedicated to visualizing and analyzing data sets produced by a larger supercomputer or (ii) on the supercomputer that generated the data itself. In both of these scenarios, visualization and analysis programs often operate with substantially less resources than the simulation code for the same data set. For either hardware scenario, the accepted rule of thumb for pure parallelism is to have on order 10% of the total memory footprint used to generate the data. Although this rule has relaxed some with rising machine costs, examples can be found at supercomputing centers across the country, including dedicated visualization machines at Lawrence Livermore (Gauss, which has eight percent of the memory of the BG/L machine) and Argonne (Eureka, which has nearly five percent of the memory of the Intrepid machine). In this paper, we have demonstrated results using large supercomputers (scenario ii), but our results are applicable to either hardware scenario.

4 Massive Data Experiments

The basic experiment used a parallel program with a high level of concurrency to read in a very large data set, apply a contouring algorithm ("Marching Cubes"), and render this surface as a 1024×1024 image. We had originally set out to perform volume rendering as well, but encountered difficulties (see section 6 on Pitfalls). An unfortunate reality of experiments of this nature is that running large jobs of the largest supercomputers in the world is a difficult and opportunistic undertaking. It was not possible to re-run on all of these machines with the improved volume rendering code. Further, these runs were undoubtedly affected by real world issues like I/O and network contention. That said, loading data, applying an algorithm, and rendering is representative of many visualization operations and this process exercises a significant portion of the code.

Our variations of this experiment fell into three categories:

- **Diverse supercomputing environments**, to test the viability of these techniques with different operating systems, I/O behavior, compute power (e.g. FLOPs), and network characteristics. We performed these tests on two Cray XT machines (OLCF/ORNL's JaguarPF & NERSC/LBNL's Franklin), a Sun Linux machine (TACC's Ranger), a CHAOS Linux machine (LLNL's Juno), an AIX machine (LLNL's Purple), and a BG/P machine (LLNL's Dawn). For each machine but Purple, we ran with 16000 cores and visualized one trillion cells[3]. For machines with more than 16000 cores available, JaguarPF and Franklin, we performed a weak scaling study, maintaining a ratio of one trillion cells for every 16000 cores. More information about the machines can be found in table 1.

[2] When a large data set is decomposed into chunks, "ghost data" refers to a redundant layer of cells around the boundaries of each chunk. These extra cells are sometimes necessary to prevent artifacts, usually due to interpolation inconsistencies.

[3] We ran with only 8000 cores and one half trillion cells on Purple, because the full machine has only 12208 cores, and only 8000 are easily obtainable for large jobs.

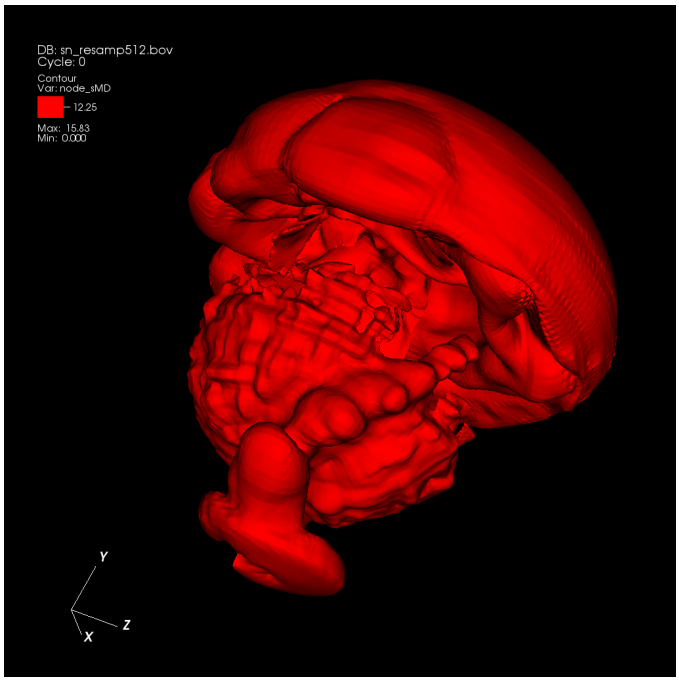

Figure 1: Contouring of two trillion cells, visualized with VisIt on Franklin using 32000 cores.

Machine Name	Machine Type/OS	Total # of Cores	Memory Per Core	CPU Type	Clock Speed	Peak FLOPS	Top 500 Rank (11/2009)
JaguarPF	Cray	224,162	2GB	XT5	2.6GHz	2.33PFLOPs	#1
Ranger	Sun Linux	62,976	2GB	Opteron Quad	2.0GHz	503.8 TFLOPs	#9
Dawn	BG/P	147,456	1GB	PowerPC	850MHz	415.7 TFLOPs	#11
Franklin	Cray	38,128	1GB	XT4	2.6GHz	352 TFLOPs	#15
Juno	Commodity (Linux)	18402	2GB	Opteron Quad	2.2GHz	131.6 TFLOPs	#27
Purple	AIX	12,208	3.5GB	POWER5	1.9GHz	92.8 TFLOPs	#66

Table 1: Characteristics of supercomputers used in this study.

- **I/O pattern**, to see if certain patterns (collective versus non-collective) exhibit better performance at scale. For the non-collective tests, the data was stored as compressed binary data (gzipped). We used ten files for every core, and every file contained 6.25 million data points, for a total of 62.5 million data points per core. Since simulation codes often write out one file per core, and following the rule of thumb that visualization codes receive one tenth of the cores of the simulation code, using multiple files per core was our best approximation at emulating common real world conditions. As this pattern may not be optimal for I/O access, we also performed a separate test where all cores use collective access on a single, large file via MPI-IO.

- **Data generation**. Our primary mechanism was to upsample data by interpolating a scalar field for a smaller mesh onto a high resolution rectilinear mesh. However, to offset concerns that upsampled data may be unrepresentatively smooth, we ran a second experiment, where the large data set was a many times over replication of a small data set. The source data set was a core-collapse supernova simulation from the CHIMERA code on a curvilinear mesh of more than three and one half million cells[4]. We applied these upsampling and replication approaches since we are not aware of any existing data sets containing trillions of cells. Moreover, the primary objective for our study is to better understand the performance and functional limits of parallel visualization software. This objective can be achieved even when using synthetic data.

[4]Sample data courtesy of Tony Mezzacappa and Bronson Messer (Oak Ridge Lab), Steve Bruenn (Florida Atlantic University) and Reuben Budjiara (University of Tennessee)

4.1 Varying Over Supercomputing Environment

For these runs, we ran on different supercomputers and kept the I/O pattern and data generation fixed, using non-collective I/O and upsampled data generation.

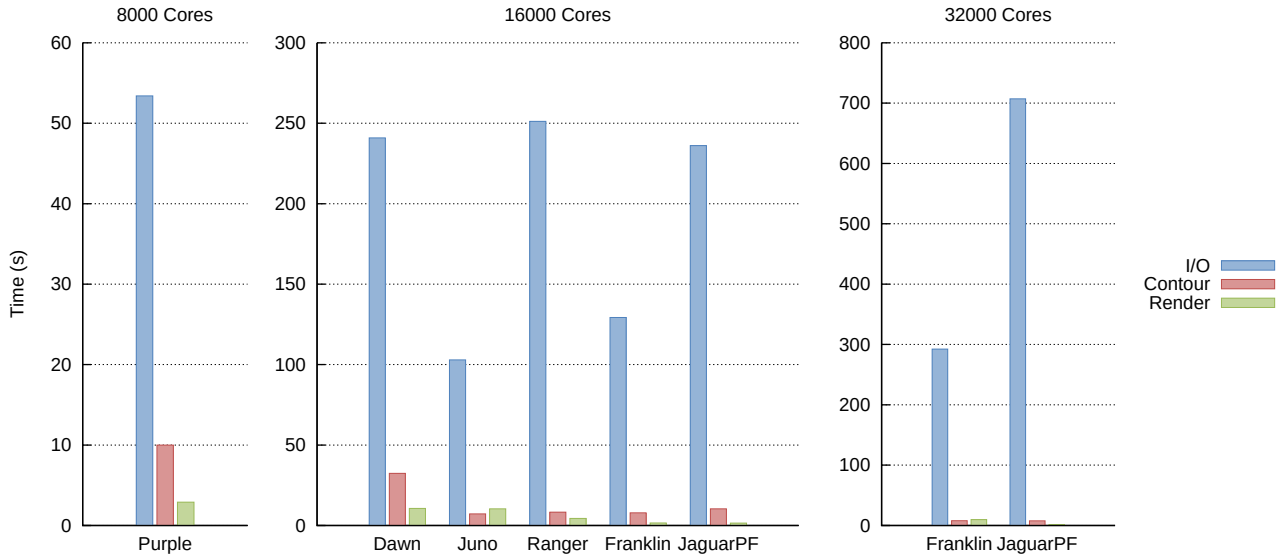

Figure 2: Runtimes for the I/O, contouring, and rendering phases.

Machine	Cores	Data set size	Total I/O time	Contour time	Total pipeline execution (time to produce surface)	Rendering time
Purple	8000	0.5 TCells	53.4s	10.0s	63.7s	2.9s
Dawn	16384	1 TCells	240.9s	32.4s	277.6s	10.6s
Juno	16000	1 TCells	102.9s	7.2s	110.4s	10.4s
Ranger	16000	1 TCells	251.2s	8.3s	259.7s	4.4s
Franklin	16000	1 TCells	129.3s	7.9s	137.3s	1.6s
JaguarPF	16000	1 TCells	236.1s	10.4s	246.7s	1.5s
Franklin	32000	2 TCells	292.4s	8.0s	300.6s	9.7s
JaguarPF	32000	2 TCells	707.2s	7.7s	715.2s	1.5s

Table 2: Performance across diverse architectures. Dawn requires that the number of cores be a power of two.

There are several noteworthy observations:

- Careful consideration of I/O striping parameters is necessary for optimal I/O performance on Lustre filesystems (Franklin, JaguarPF, Ranger, Juno, & Dawn). Even though JaguarPF has more I/O resources than Franklin, its I/O performance did not perform as well because its default stripe count is 4. In contrast, the default stripe count of 2 on Franklin was better suited for the I/O pattern, which read ten separate gzipped files per core. Smaller stripe counts often benefit file-per-core I/O because the files are usually small enough (tens of MB) that they won't contain many stripes, and spreading them thinly over many I/O servers increases contention.

- Because the data was gzipped, there was an unequal I/O load across cores. The reported I/O times measure the elapsed time between file open and a barrier after all cores are finished reading. Because of this load imbalance, I/O time did not scale linearly from 16000 to 32000 cores on Franklin and JaguarPF.

- The Dawn machine has the slowest clock speed (850MHz), which is reflected in its contouring and rendering times.

- Although many of the variations we observed were expected, for example due to slow clock speeds, interconnects, or I/O servers, some variation was not:

  – For Franklin's increase in rendering time from 16000 to 32000 cores, seven to ten network links failed that day and had to be statically re-routed, resulting in suboptimal network performance. Rendering algorithms are "all reduce" type operations that are very sensitive to bisectional bandwidth, which was affected by this issue.

  – For Juno's slow rendering time, we suspect a similar network problem.

  We have have been unable to schedule time on either machine to follow up on these issues.

4.2 Varying Over I/O Pattern

For these runs, we compared collective and non-collective I/O patterns on Franklin for a one trillion cell upsampled data set. In the non-collective test, each core performed ten pairs of fopen and fread calls on independent gzipped files without any coordination among cores. In the collective test, all cores synchronously called MPI_File_open once then MPI_File_read_at_all ten times on a shared file (each read call corresponded to a different domain in the data set). An underlying collective buffering or "two phase" algorithm in Cray's MPI-IO implementation aggregated read requests onto a subset of 48 nodes (matching the 48 stripe count of the file) that coordinated the low-level I/O workload, dividing it into 4MB stripe-aligned fread calls. As the 48 aggregator nodes filled their read buffers, they shipped the data through MPI to their final destination among the 16016 cores. We used a different number of cores (16000 versus 16016) to make data layout more convenient for each scheme.

Machine	I/O pattern	Cores	Data set size	Total I/O time	Data read	Read bandwidth
Franklin	Collective	16016	1 TCells	478.3s	3725.3GB	7.8GB/s
Franklin	Non-collective	16000	1 TCells	129.3s	954.2GB	7.4GB/s

Table 3: Performance with different I/O patterns. The data set size for collective I/O corresponds to 4 bytes for one trillion cells. The data read is not 4000GB, because one gigabyte is 1,073,741,824 bytes. The data set size for non-collective I/O is much smaller because it was gzipped.

Both patterns led to similar read bandwidths, 7.4 and 7.8 GB/s, which are about 60% of the maximum available bandwidth of 12 GB/s on Franklin. In the non-collective case, load imbalances caused by different gzip compression factors may account for this discrepancy. For the collective I/O, we speculate that coordination overhead between the MPI tasks may be limiting efficiency. Further, we note that achieving one hundred percent efficiency would not substantially change the balance between I/O and computation.

4.3 Varying Over Data Generation

For these runs, we processed both upsampled and replicated data sets with one trillion cells on 16016 cores of Franklin using collective I/O.

Data generation	Total I/O time	Contour time	Total pipeline execution	Rendering time
Upsampling	478.3s	7.6s	486.0s	2.8s
Replicated	493.0s	7.6s	500.7s	4.9s

Table 4: Performance across different data generation methods.

The contouring times were identical, since this operation is dominated by the movement of data through the memory hierarchy (L2 cache to L1 cache to registers), rather than the relatively rare case where a cell contains a contribution to the isosurface. The rendering time nearly doubled, because the contouring algorithm produced more triangles with the replicated data set.

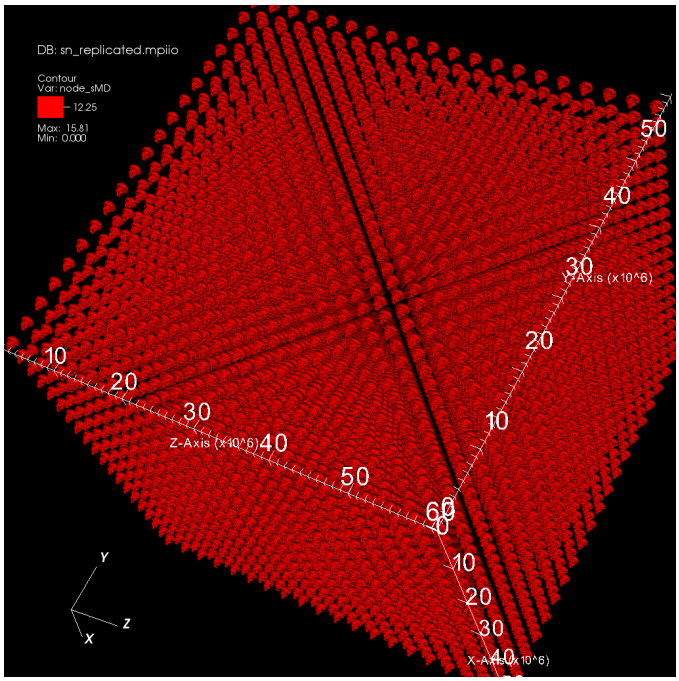

Figure 3: Contouring of replicated data (one trillion cells total), visualized with VisIt on Franklin using 16016 cores.

5 Scaling Experiments

To further demonstrate the scaling properties of pure parallelism, we present results that demonstrate weak scaling (scaling up the number of processors with a fixed amount of data per processor) for both isosurface generation and volume rendering[5]. Once again, these algorithms test a large portion of the underlying pure parallelism infrastructure and indicate strong likelihood of weak scaling for other algorithms in this setting. Demonstrating weak scaling properties on high performance computing systems meets the accepted standards of "Joule certification," which is a program within the U.S. Office of Management and Budget to evaluate the effectiveness of agency programs, policies and procedures.

5.1 Study Overview

The scaling studies were performed on output from Denovo, a three-dimensional radiation transport code from Oak Ridge National Laboratory, which modeled radiation dose levels for a nuclear reactor core and surrounding areas. The Denovo simulation code does not directly output a scalar field representing effective dose. Instead, we calculated this dose at runtime through a linear combination of 27 scalar fluxes. For both the isosurface and volume rendering tests, VisIt read in 27 scalar fluxes and combined them to form a single scalar field representing radiation dose levels. The isosurface extraction test consisted of extracting six evenly spaced iso-contour values of the radiation dose levels and rendering an 1024×1024 pixel image. The volume rendering test consisted of ray-casting with 1000, 2000 and 4000 samples per ray of the radiation dose level on a 1024×1024 pixel image.

These visualization algorithms were run on a baseline Denovo simulation consisting of 103,716,288 cells on 4096 spatial domains with a total size on disk of 83.5 GB. The second test was run on a Denovo simulation nearly three times the size of the baseline run, with 321,117,360 zones on 12720 spatial domains and a total size on disk of 258.4 GB.

[5]This study was run in July of 2009, after the volume rendering algorithm was fixed.

5.2 Results

The baseline calculation used 4096 cores and the larger calculation used 12270 cores. These core counts are large relative to the problem size and were chosen because they represent the number of cores used by Denovo. This matching core count was important for the Joule study and is also indicative of performance for an in situ approach. Note that I/O was not included in these tests.

Algorithm	Cores	Minimum Time	Maximum Time	Average Time
Calculate radiation	4,096	0.18s	0.25s	0.21s
Calculate radiation	12,270	0.19s	0.25s	0.22s
Isosurface	4,096	0.014s	0.027s	0.018s
Isosurface	12,270	0.014s	0.027s	0.017s
Render (on core)	4,096	0.020s	0.065s	0.0225s
Render (on core)	12,270	0.021s	0.069s	0.023s
Render (across cores)	4,096	0.048s	0.087s	0.052s
Render (across cores)	12,270	0.050s	0.091s	0.053s

Table 5: Weak scaling of isosurfacing. **Isosurface** refers to the execution time of the isosurface algorithm, **Render (on core)** indicates the time to render that cores's surface, while **Render (across cores)** indicates the time to combine that image with the images of other cores. **Calculate radiation** refers to the time to calculate the linear combination of the 27 scalar fluxes.

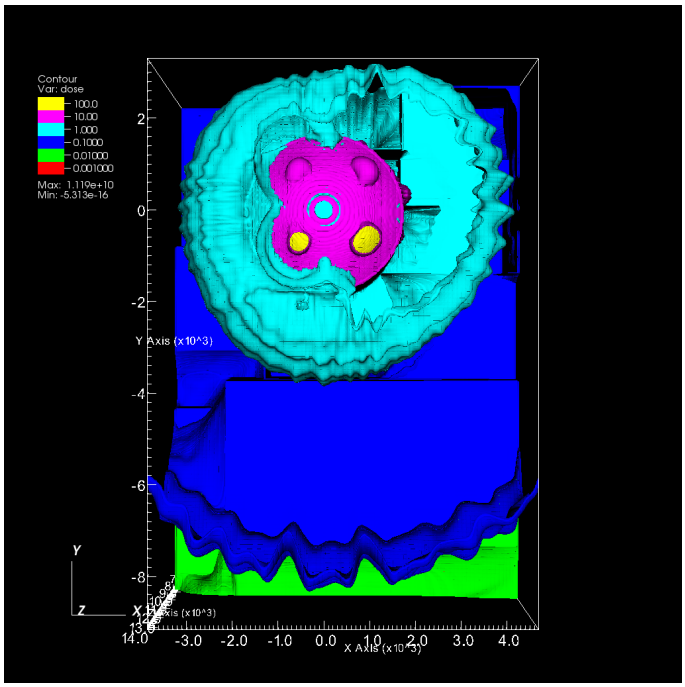

Figure 4: Rendering of an isosurface from the Denovo calculation, produced by VisIt using 12,270 cores of JaguarPF.

6 Pitfalls at Scale

The common theme of this section is how decisions that were appropriate on the order of hundreds of cores become serious impediments at higher levels of concurrency. The offending code existed at various levels of the software, from core algorithms (volume rendering) to code that supports the algorithms (status updates) to foundational code (plugin loading). The volume rendering and status update problems were easily correctable and the fixes will be in the next public version of VisIt. The plugin

Cores	Samples Per Ray: 1000	2000	4000
4,096	7.21s	4.56s	7.54s
12,270	6.53s	6.60s	6.85s

Table 6: Weak scaling of volume rendering. 1000, 2000, and 4000 represent the number of samples per ray. The algorithm demonstrates super-linear performance, because the number of samples per core (which directly affects work performed) is smaller at 12,270 cores, while the number of cells per core is constant. The anomaly where performance increases at 2000 samples per ray requires further study.

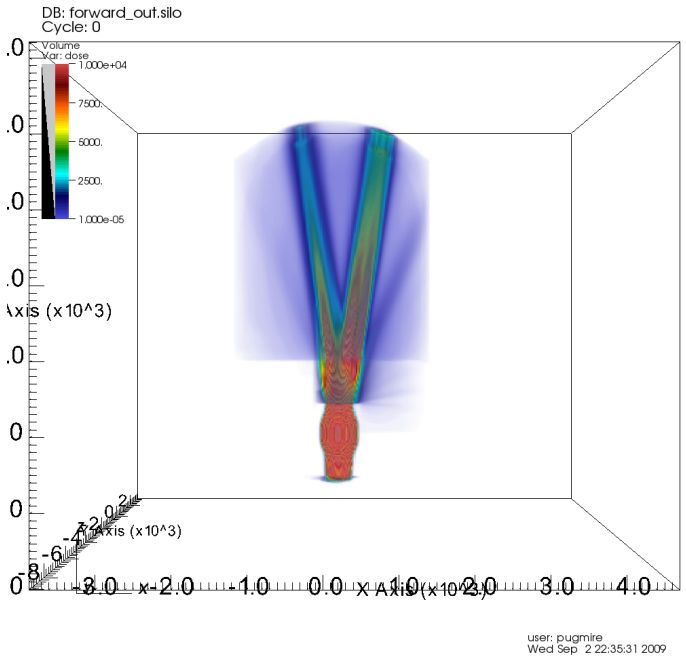

Figure 5: Volume rendering of data from the Denovo calculation, produced by VisIt using 12,270 cores on JaguarPF.

loading problem was partially addressed, but a total fix may require removing shared libraries altogether.

6.1 Volume Rendering

The volume rendering code used an $\mathcal{O}(n^2)$ buffer, where n is the number of cores. An all-to-all communication phase re-distributed samples along rays according to a partition with dynamic assignments. An "optimization" for this phase was to minimize the number of samples that needed to be communicated by favoring assignments that kept samples on their originating core. This "optimization" required an $\mathcal{O}(n^2)$ buffer that contained mostly zeroes. Although this "optimization" was indeed effective for small core counts, the coordination overhead caused VisIt to run out of memory at scale. Our solution was to eschew the optimization, simply assigning pixels to cores without concern of where individual samples lay. As the number of samples gets smaller with large core counts, ignoring this optimization altogether at high concurrency is probably the best course of action.

We do not have comprehensive volume rendering data to present for the one trillion cell data sets. However, we observed that after our changes, ray casting performance was approximately five seconds per frame for a 1024x1024 image.

For the weak scaling study on Denovo data, running with 4,096 cores, the speedup was approximately a factor of five (see table 7).

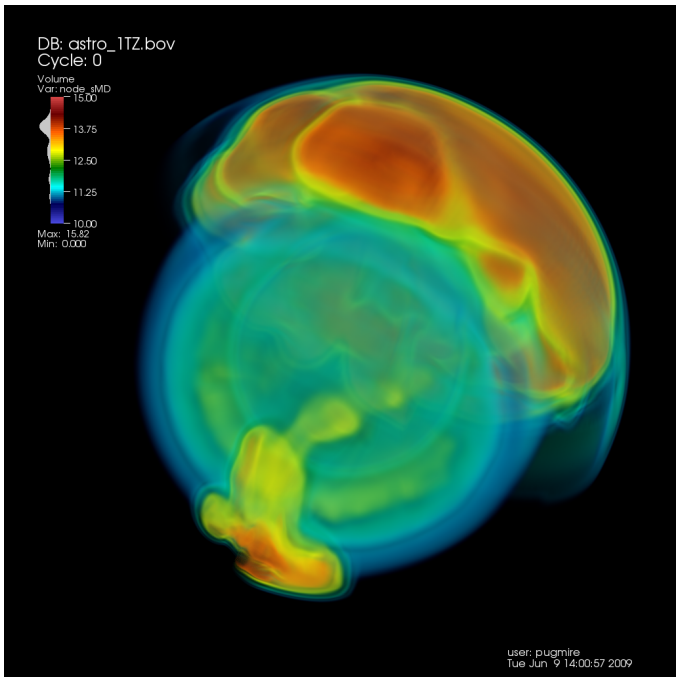

Figure 6: Volume rendering of one trillion cells, visualized by VisIt on JaguarPF.

Cores	Date run	Samples Per Ray: 1000	2000	4000
4,096	Spring 2009	34.7s	29.0s	31.5s
4,096	Summer 2009	7.21s	4.56s	7.54s

Table 7: Volume rendering of Denovo data at 4,096 cores before and after speedup.

6.2 All-To-One Communication

At the end of every pipeline execution, each core reports its status (success or failure) as well as some meta-data (extents, etc). These status and extents were being communicated from each MPI task to MPI task 0 through point to point communication. However, the delay in having every MPI task send a message to MPI task 0 was significant, as shown in the table below. This was corrected subsequently with a tree communication scheme[6].

Machine	All-to-one status	Cores	Data set size	Total I/O time	Contour time	Total pipeline execution	Pipeline minus contour & I/O	Date run
Dawn	yes	16384	1 TCells	88.0s	32.2s	368.7s	248.5s	June 2009
Dawn	yes	65536	4 TCells	95.3s	38.6s	425.9s	294.0s	June 2009
Dawn	no	16384	1 TCells	240.9s	32.4s	277.6s	4.3s	August 2009

Table 8: Performance with old status checking code versus new status checking code. Taking the pipeline time and subtracting contour and I/O time approximates how much time was spent waiting for status and extents updates. Note that the other runs reported in this article had status checking code disabled and that the last Dawn run is the only reported run with new status code.

Another "pitfall" is the difficulty in getting consistent results. Looking at the I/O times from the Dawn runs, there is a dramatic slowdown from June to August. This is because, in July, the I/O servers backing the file system became unbalanced in their disk usage. This caused the algorithm that assigns files to servers to switch from a "round robin" scheme to a statistical scheme, meaning files were no longer assigned uniformly across I/O servers. While this scheme makes sense from an operating

[6]After our initial round of experiments, our colleague Mark Miller of Lawrence Livermore Lab independently observed the same problem and made this enhancement.

system perspective by leveling out the storage imbalance, it hampers access times for end users. With the new scheme, the number of files assigned to each I/O server followed a Poisson distribution, with some servers assigned three or four more times more files than others. Since each I/O server has a fixed bandwidth, those with more files will take longer to serve up data, resulting in I/O performance degradation of factors of three or four for the cores trying to fetch data from the overloaded I/O servers.

6.3 Shared Libraries and Startup Time

During our first runs on dawn, using only 4096 cores, we observed lags in startup time that worsened as the core count increased. Each core was reading plugin information from the filesystem, creating contention for I/O resources. We addressed this problem by modifying VisIt's plugin infrastructure so that plugin information could be loaded on MPI task 0 and broadcast to other cores. This change made plugin loading nine times faster.

That said, startup time was still quite slow, taking as long as five minutes. VisIt uses shared libraries in many instances to allow new plugins to access symbols not used by any current VisIt routines; compiling statically would remove these symbols. The likely path forward is to compile static versions of VisIt for the high concurrency case. This approach will likely be palatable because new plugins are frequently developed at lower levels of concurrency.

7 Conclusion

Our results contribute to the understanding of pure parallelism, the dominant paradigm for production visualization tools, at extreme scale. The experiments were designed to answer two questions: (1) will pure parallelism be successful on extreme data sets at extreme concurrency on a variety of architectures? And (2) can we demonstrate that pure parallelism scales from a weak scaling perspective? We believe our results demonstrate that pure parallelism does scale, though it is only as good as its supporting I/O infrastructure.

We were successful in visualizing up to four trillion cells on diverse architectures with production visualization software. The supercomputers we used were "underpowered," in that the current simulation codes on these machines produce meshes far smaller than a trillion cells. They were appropriately sized, however, when considering the rule of thumb that the visualization task should get ten percent of the resources of the simulation task, and assuming our trillion cell mesh represents the simulation of a hypothetical 160,000 core machine.

I/O performance became a major focus of our study, as slow I/O prevented interactive rates when loading data. Most supercomputers are configured for I/O bandwidth to scale with number of cores, so the bandwidths we observed in our experiments are commensurate with what we should expect when using ten percent of a future supercomputer. Thus the inability to read data sets quickly presents a real concern going forward. Worse, the latest trends in supercomputing show diminishing I/O rates relative to increasing memory and FLOPs, meaning that the I/O bottleneck we observed may potentially constrict further as the next generation of supercomputers arrives.

There are potential hardware and software solutions that can help address this problem, however. From the software side, multi-resolution techniques and data subsetting (such as query-driven visualization) limit how much data is read, while in situ visualization avoids I/O altogether. From the hardware side, an increased focus on balanced machines that have I/O bandwidth commensurate with compute power would reduce I/O time. Further, emerging I/O technologies, such as FLASH drives, have the possibility to make significant impacts. From this study, we conclude that some combination of these solutions will be necessary to overcome the I/O problem and obtain good performance.

8 Acknowledgments

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Com-

puting (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET). Further, we thank Mark Miller for the status update improvements referenced in 6.2 and the anonymous reviewers, whose suggestions greatly improved this paper. Finally, the authors acknowledge the resources that contributed to the research results reported in this paper:

- The National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
- The Livermore Computing Center at Lawrence Livermore National Laboratory, which is supported by the National Nuclear Security Administration of the U.S. Department of Energy under Contract DE-AC52-07NA27344.
- The Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. De-AC05-00OR22725.
- The Texas Advanced Computing Center (TACC) at The University of Texas at Austin for use of their HPC resources.
- We thank the personnel at the computing centers that helped us to perform our runs, specifically Katie Antypas, Kathy Yelick, Francesca Verdier, and Howard Walter of LBNL's NERSC, Paul Navratil, Kelly Gaither, and Karl Schulz of UT's TACC, James Hack, Doug Kothe, Arthur Bland, and Ricky Kendall of ORNL's LCF, and David Fox, Debbie Santa Maria, and Brian Carnes of LLNL's LC.

REFERENCES

[1] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *Proceedings of IEEE Visualization*, 2005.

[2] Hank Childs and Mark Miller. Beyond meat grinders: An analysis framework addressing the scale and complexity of large data sets. In *Proceedings of HPC2006*, 2006.

[3] J. Clyne, P. Mininni, A. Norton, and M. Rast. Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics*, 9:301, August 2007.

[4] Computational Engineering International, Inc. *EnSight User Manual*.

[5] Robert Haimes. pV3: A Distributed System for Large-Scale Unsteady CFD Visualization. In *AIAA paper*, pages 94–0321, 1994.

[6] C.R. Johnson, S. Parker, and D. Weinstein. Large-scale computational science applications using the SCIRun problem solving environment. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[7] C. Charles Law, Amy Henderson, and James Ahrens. An application architecture for large data visualization: a case study. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 125–128. IEEE Press, 2001.

[8] Valerio Pascucci and Randall J. Frank. Global static indexing for real-time exploration of very large regular grids. In *SC*, page 2, 2001.

[9] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R Latham. End-to-End Study of Parallel Volume Rendering on the IBM Blue Gene/P. In *Proceedings of ICPP'09 Conference*, 2009.

[10] Oliver Ruebel, Prabhat, Kesheng Wu, Hank Childs, Jeremy Meredith, Cameron G. R. Geddes, Estelle Cormier-Michel, Sean Ahern, Gunther H. Weber, Peter Messmer, Hans Hagen, Bernd Hamann, and E. Wes Bethel. High performance multivariate visual data exploration for extemely large data. In *SuperComputing 2008 (SC08)*, 2008.

[11] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 93–ff. IEEE Computer Society Press, 1996.

[12] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization 2002 Course Notes*, 2002.