Extreme Scaling of Production Visualization Software on Diverse Architectures

- Hank Childs, David Pugmire, Sean Ahern,Brad Whitlock, Mark Howison, Prabhat, Gunther H. Weber, and E. Wes Bethel

Presented by -

Akshay Toshniwal
Devang Agrawal
Prakhar Mandloi

Course Instructors -

Dr. Soumya Dutta
Dr. Preeti Malakar

Content

Introduction

- We need to have visualisation software that can keep up with scientific simulations that have enormous datasets otherwise it will potentially jeopardise the value of simulations.

- Most of the visualization software uses Bruteforce pure parallelism.

- Showing how pure parallelism paradigm scales to massive datasets

- The findings on scaling characteristics and bottlenecks will help us to understand how pure parallelism would perform in the future

- This research seeks to better understand how pure parallelism will perform on more cores with large data sets

Questions to ponder upon

- How does pure-parallelism scale?
- What are the bottlenecks?
- What are the pitfalls of running production software at a massive scale?
- Will pure parallelism be effective for next generation of datasets?

Important Terminologies

- **Collective I/O** : It is a parallel I/O strategy where all processes work together to perform I/O operations more efficiently. It can help balance the I/O workload across the available resources. Eg. MPI_File_read_all and MPI_File_write_all are used for reading and writing data collectively.

- **Non Collective I/O** : each process or core independently performs its own I/O operations without any coordination with other processes. Each process may read or write different parts of the data at different times. Eg. MPI_File_read and MPI_File_write are used for reading and writing data independently.

- **Embarrassingly parallel and Non-embarassingly parallel -** Some visualization algorithms require no inter process communication and can operate on portion of the data set without coordination with the other cores these processes are called embarrassingly parallel. Eg : Slicing and Contouring. However some important algorithms do require interprocess communications and therefore non embarrassingly parallel Eg. Volume Rendering

- **Upsampled Data –** The data is interpolated to fill in the discrete values by mathematical calculations resulting in smoother dataset.

- **Replicated Data –** The data is replicated within the space if any value within the cells is required.

- **Stripe Count –** Number of simultaneous connections from a file.

Pure Parallelism

- Parallelization technique with no optimizations to reduce the amount of data being read.
- Simulation writes data to disk
- Read by visualization software at full resolution
- Visualization software store it in primary memory
- Because of large data we parallelize the processing by partitioning the data over processors
- Each processor works on each piece of process
- Combines the data through rendering.

- VisIt visualization software was used in these experiments

- The pure parallelism paradigm accommodates both type of algorithms .For embarrassingly parallel algorithms, each core can directly apply the serial algorithms to its portion of the data set also Pure parallelism is often the simplest environment to implement Non-embarrassingly parallel algorithms as because every piece of data is available at any time, at full resolution.

- While operating rendering algorithms we combines all the cores as if all the data was rendered on a single core.

- This research uses pure parallelism on hardware scenario where processing occurs on the supercomputer that generated the data.

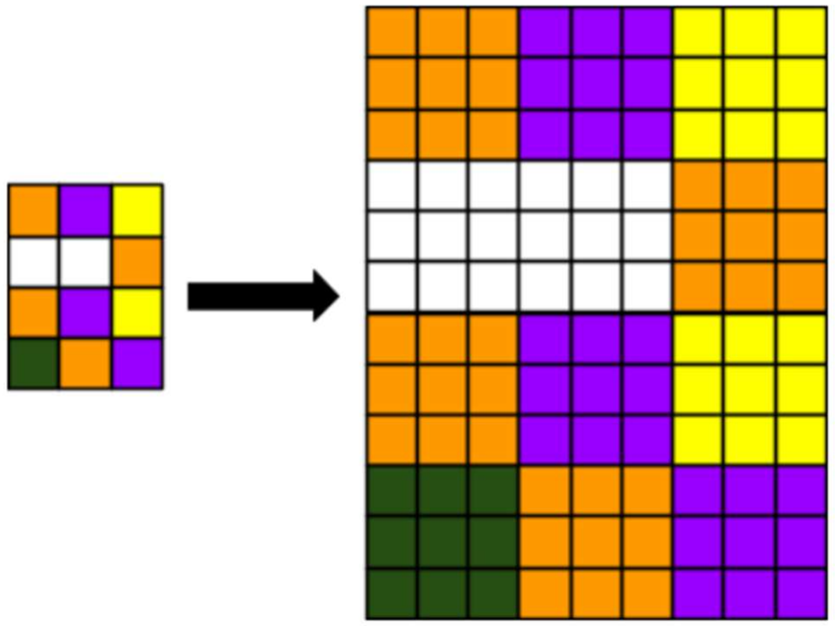
Massive-Data Experiments

- Basic experiment used in this research paper is a parallel program with high concurrency to read a very large data set, apply Marching cubes algorithm for contouring and at last render this surface as 1024 x 1024 image.

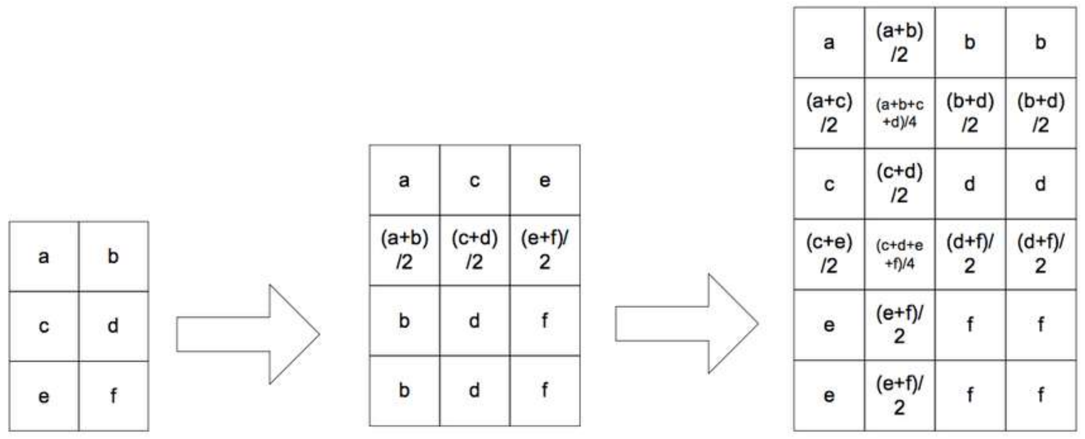
- They also tried to implement volume rendering but due to requirement of O(n^2) buffer caused Visit to run out of memory at scale.

- Variation of these experiments falls into three categories :
  - Varying over the supercomputing environments.
  - Varying over the I/O patterns.
  - Varying over data generation.

Upsampled Data

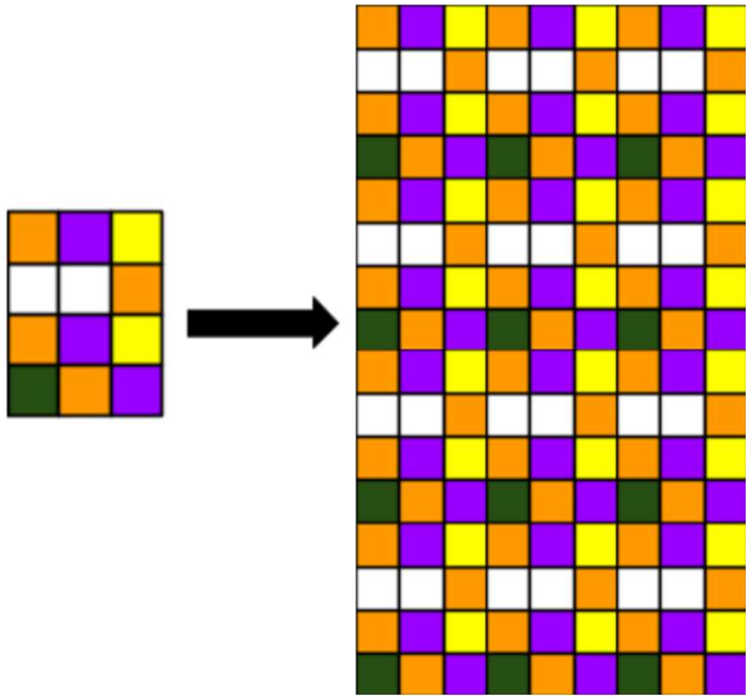


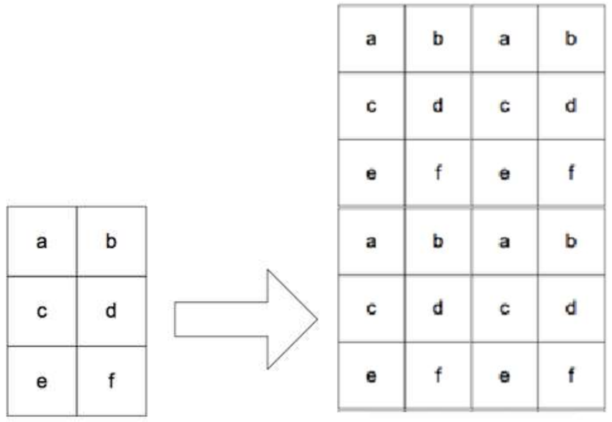
https://shinyprints.com/blog/upsample-your-images-for-print/



https://researchgate.net/figure/An-example-in-Image-Upsample-To-implement-image-upsample-in-GPU-is-also-straightforward_fig3_311429265

Replicated Data

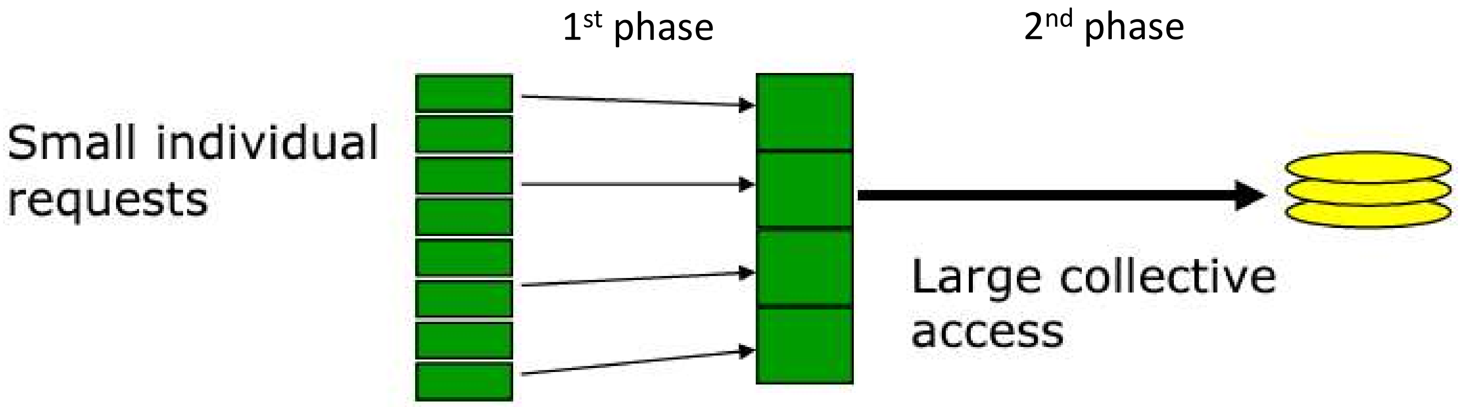




a	b
c	d
e	f

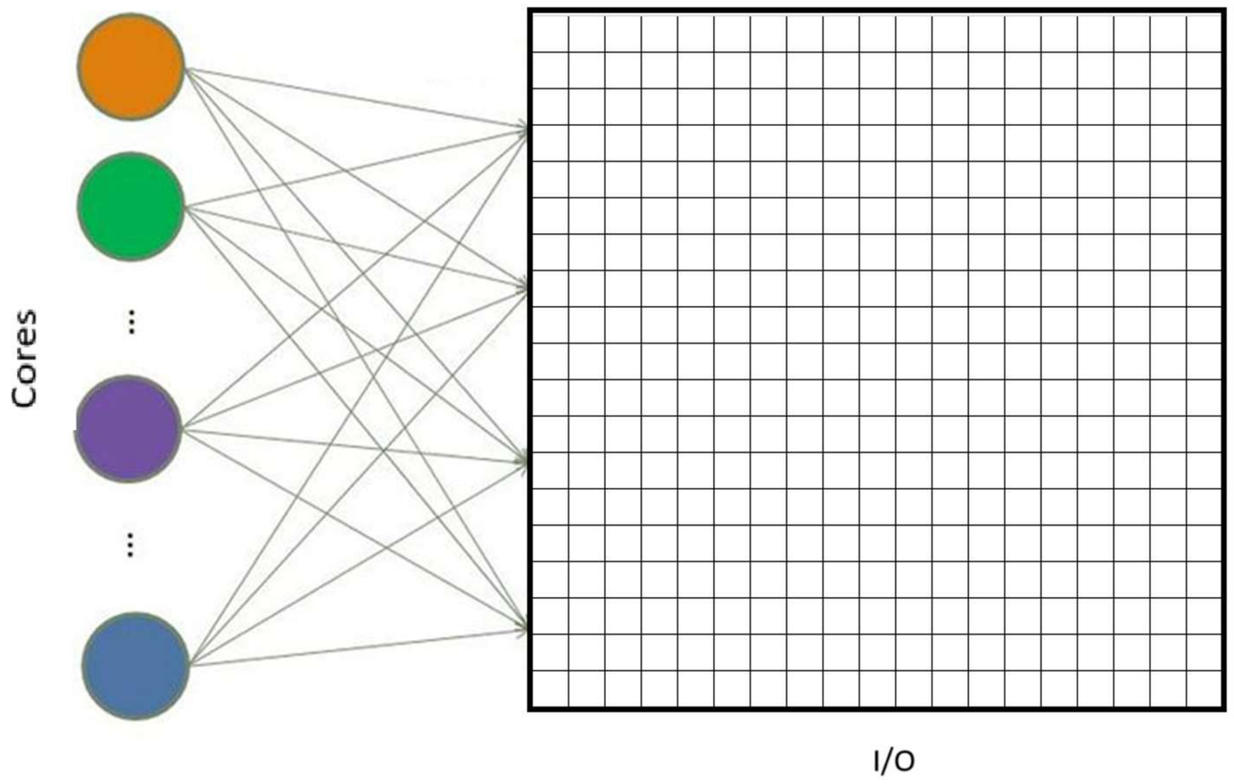
a	b	a	b
c	d	c	d
e	f	e	f
a	b	a	b
c	d	c	d
e	f	e	f

Collective I/O



Lecture 32 Introduction to MPI I/O (University of Illinois Urbana-Champaign)
https://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf

Non-collective I/O (Independent I/O)



Weak Scaling

- Weak scaling refers to the practice of increasing the number of processors or cores in a parallel computing system while simultaneously increasing the size of the problem (e.g., data set size) proportionally.

- The objective is to maintain a constant workload per processor as the system scales up. In other words, weak scaling measures how effectively a system can handle larger problems as more computational resources are added.

- Weak scaling helps in scaling number of processors with a fixed amount of data per processor.

- This maintains constant workload per processor providing us with real world scenario.

Different supercomputing environments used-

- Details of the supercomputer environment used in this research are –

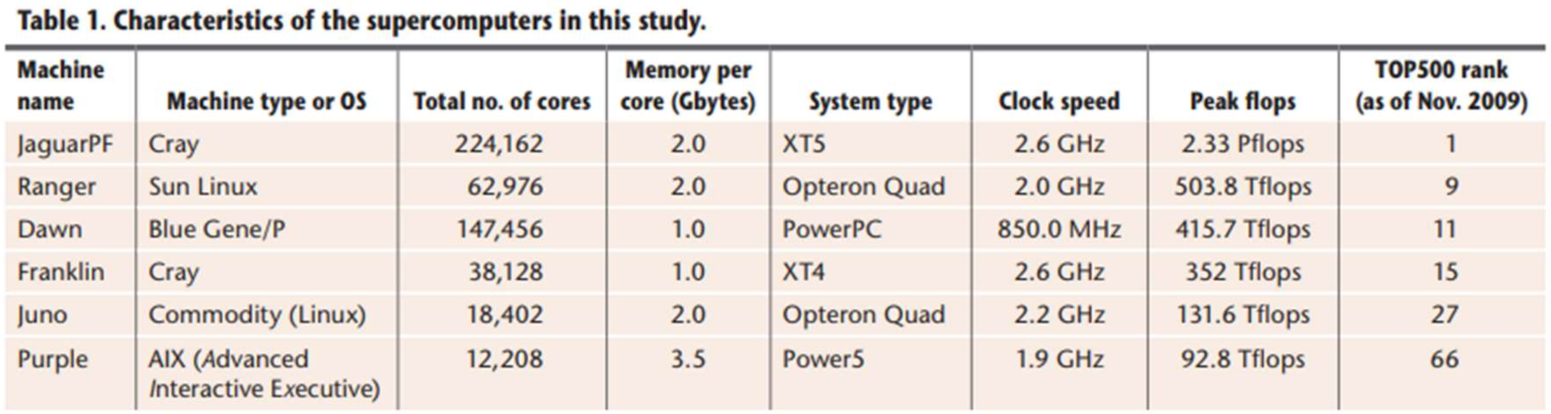


Table 1. Characteristics of the supercomputers in this study.

Machine name	Machine type or OS	Total no. of cores	Memory per core (Gbytes)	System type	Clock speed	Peak flops	TOP500 rank (as of Nov. 2009)
JaguarPF	Cray	224,162	2.0	XT5	2.6 GHz	2.33 Pflops	1
Ranger	Sun Linux	62,976	2.0	Opteron Quad	2.0 GHz	503.8 Tflops	9
Dawn	Blue Gene/P	147,456	1.0	PowerPC	850.0 MHz	415.7 Tflops	11
Franklin	Cray	38,128	1.0	XT4	2.6 GHz	352 Tflops	15
Juno	Commodity (Linux)	18,402	2.0	Opteron Quad	2.2 GHz	131.6 Tflops	27
Purple	AIX (Advanced Interactive Executive)	12,208	3.5	Power5	1.9 GHz	92.8 Tflops	66

- The authors had 32000 cores available on JaguarPF and Franklin (both using Cray OS). They also performed the weak-scaling study maintaining the ratio of 1 trillion cells for every 16000 cores

Varying over the supercomputing environment

- I/O patterns and data generation techniques were fixed
- Experiment used non-collective I/O and upsampled data
- The source data set is a core-collapse supernova simulation from the Chimera code. (The sample data was courtesy of Tony Mezzacappa and Bronson Messer from Oak Ridge, Steve Bruenn from Florida Atlantic University and Reuben Budjiara from University of Tennessee)

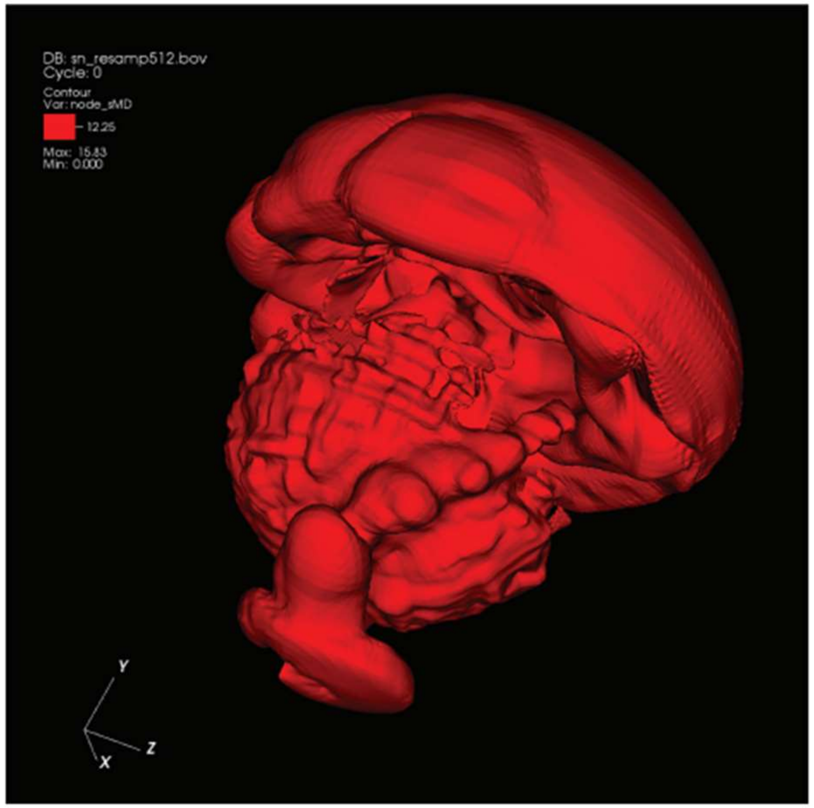


Figure 1. Our first category of experiments varied over supercomputing environment. This image is from the Franklin run, showing a contour of a 32,000-core VisIt visualization of a two-trillion-cell data set.

Ultrascale Visualization

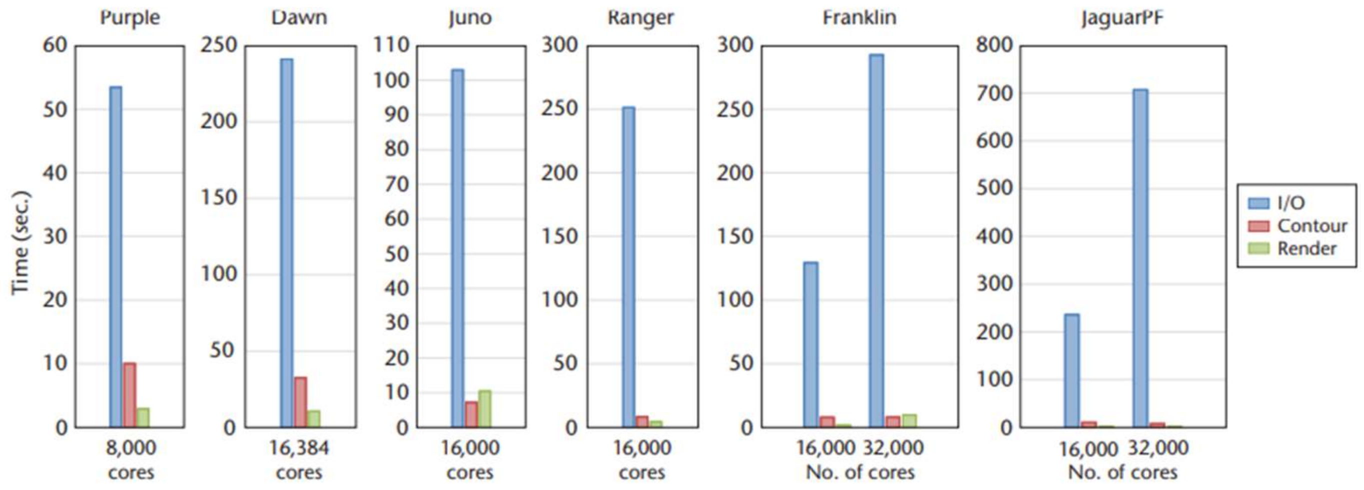


Figure 2. Runtimes for I/O, contouring, and rendering. These results show that, although there is variation across the supercomputers, I/O is the slowest phase.

Table 2. Performance across diverse architectures.

Machine	No. of cores	Data set size (TCells)	Total I/O time (sec.)	Contour time (sec.)	Total pipeline execution time (sec.)†	Rendering time (sec.)
Purple	8,000	0.5	53.4	10.0	63.7	2.9
Dawn	16,384*	1.0	240.9	32.4	277.6	10.6
Juno	16,000	1.0	102.9	7.2	110.4	10.4
Ranger	16,000	1.0	251.2	8.3	259.7	4.4
Franklin	16,000	1.0	129.3	7.9	137.3	1.6
JaguarPF	16,000	1.0	236.1	10.4	246.7	1.5
Franklin	32,000	2.0	292.4	8.0	300.6	9.7
JaguarPF	32,000	2.0	707.2	7.7	715.2	1.5

** Dawn requires that the number of cores be a power of two.*
† This measure indicates the time to produce the surface.

Observations (Varying environment) :

1. The size of cell data was relatively very small hence Franklin's(default stripe count = 2) performed better than JaguarPF(default stripe count = 4) even though JaguarPF had more I/O resources.

2. I/O load was not load balanced resulting in non-linear scaling in I/O time from 16000 to 30000 cores on Franklin and JaguarPF.

3. Dawn had the slowest clock speed which is reflected in its contouring and rendering time

4. When authors ran Franklin from 16000 to 32000 cores, 7 to 10 network links failed, and were statically rerouted resulting in decreased performance, Authors suspected same errors in Juno's slow rendering time.

Varying over the I/O Pattern

- Compared Collective and non-collective I/O patterns on Franklin for one trillion cell upsampled data set
- In the non-collective test, 10 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 and then MPI_File_read_at_all 10 times on a shared file (each read call correspond to a different domain in the data set).
- An underlying collective buffer works in 2 phase and 48 nodes are dedicated to low level I/O workload, dividing it into 4 Mbyte stripe-aligned fread cell.
- Once aggregator nodes have filled their read buffer they ship their data through MPI to final destination among 16016 cores.

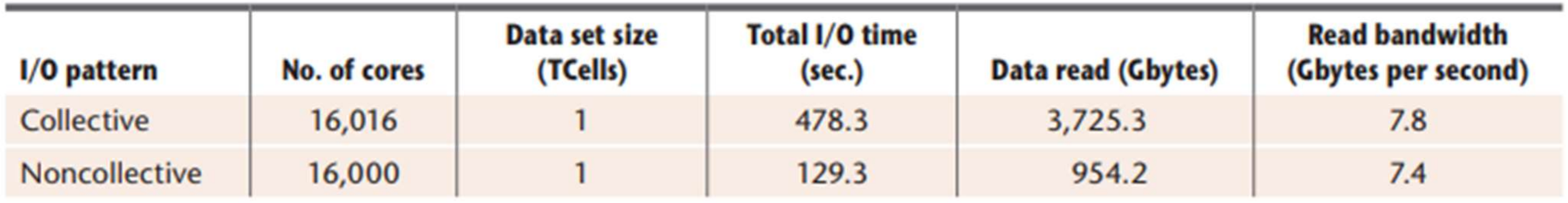


Table 3. The performance of different I/O patterns on Franklin.

I/O pattern	No. of cores	Data set size (TCells)	Total I/O time (sec.)	Data read (Gbytes)	Read bandwidth (Gbytes per second)
Collective	16,016	1	478.3	3,725.3	7.8
Noncollective	16,000	1	129.3	954.2	7.4

Observations (Varying I/O Methods) -

1. The data set for collective I/O corresponds to 4 bytes for one trillion cells and data read is 3725.3 GBs as 1 GB is 1,073,741,824 bytes

2. The data read in non collective I/O is much smaller as it is gzipped.

3. In both cases maximum available bandwidth is 12GBps but only approximately 60%(7.8 in collective and 7.4 in non-collective) of maximum available was used. Authors blamed this inefficiency on Cordination overhead between the MPI Tasks and Gzip compression factors.

Varying over Data Generation

- Compared both upsampled and replicated data sets with one trillion on 16,016 cores of franklin using collective I/O.

- Environment(Franklin) and I/O(collective) were fixed in this method.

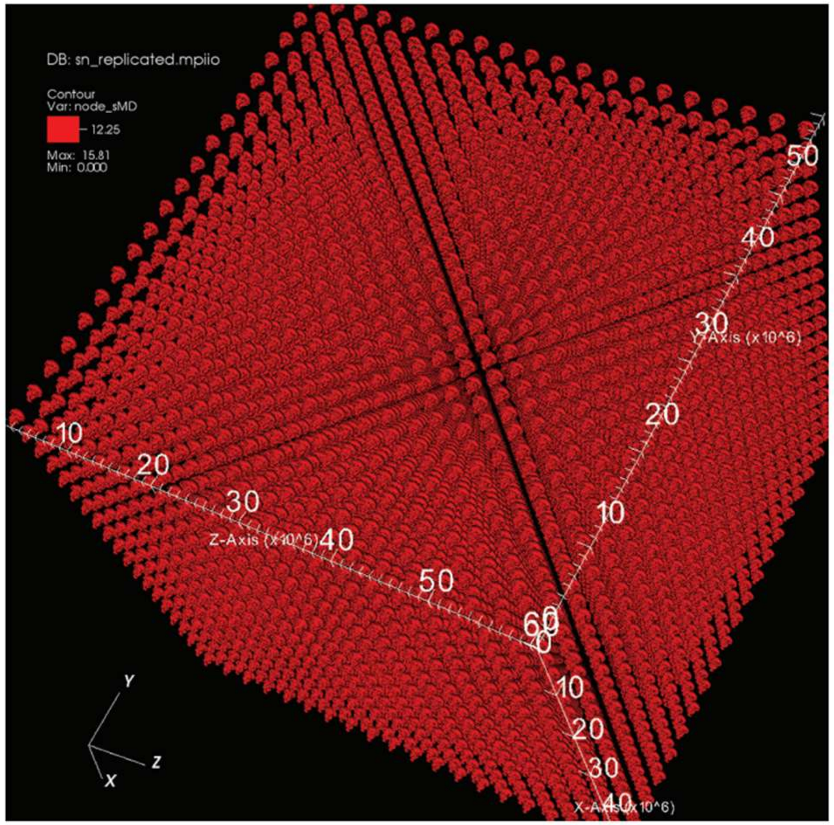


Figure 3. Our third category of experiments varied over data generation, to ensure we weren't studying data that was unrepresentatively smooth. This image shows a contouring of replicated data (one trillion cells total), visualized with VisIt on Franklin using 16,016 cores.

Observations (Varying Data Generation) -

1. The contouring times were identical because 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 in which a cell contains a contribution to the isosurface.

2. The rendering time nearby doubled because the contouring algorithm produced more triangles with the replicated data set.

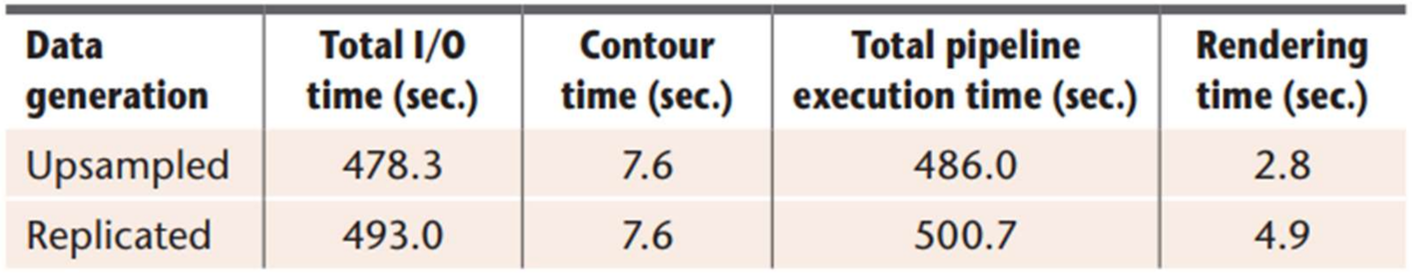


Table 4. Performance across different data generation methods.

Data generation	Total I/O time (sec.)	Contour time (sec.)	Total pipeline execution time (sec.)	Rendering time (sec.)
Upsampled	478.3	7.6	486.0	2.8
Replicated	493.0	7.6	500.7	4.9

Scaling Experiments

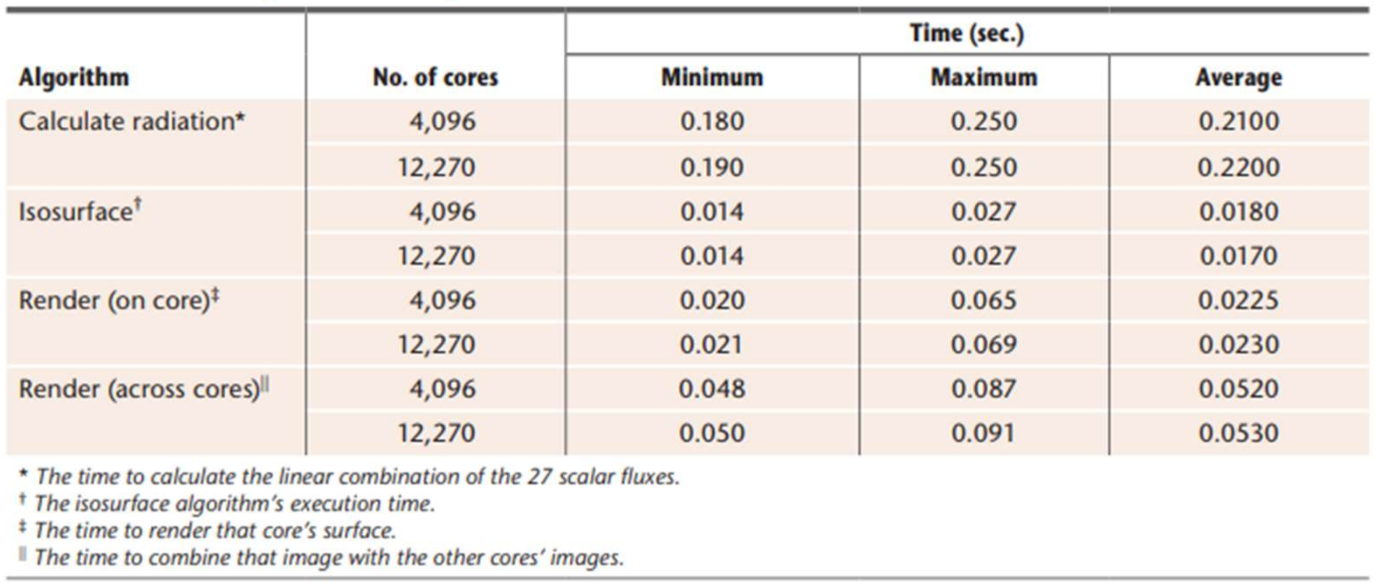
- Performed weak scaling for demonstrating scaling properties of pure parallelism for both isosurface generation and volume rendering.
- Dataset taken for study was output from Denovo(Oak Ridge National Laboratory's 3D Radiation Transport Code).It models nuclear reactor core radiation dose levels and surface rounding areas.
- Visit read and combined 27 scalar fluxes at runtime to get single scalar field representing radiation dose levels.
- The isocontouring extracted six evenly spaced isocontour values of radiation dose level and rendered a 1024 x 1024 pixel image.
- The volume rendering test consisted of raycasting with 1000, 2000 and 4000 samples per ray of radiation dose level on a 1024 x 1024 pixel image.

Scaling Study (Cont.)

- There were two types of simulations on which they run the tests:
  - First one, the baseline run consists of 103,716,288 cells on 4096 spatial domains with a 83.5-Gbyte disk.
  - Second one, this run was nearly three times the size of baseline run, with 321,117,360 cells on 12720 spatial domains and a 258.4-Gbyte disk.

Results of Scaling Study

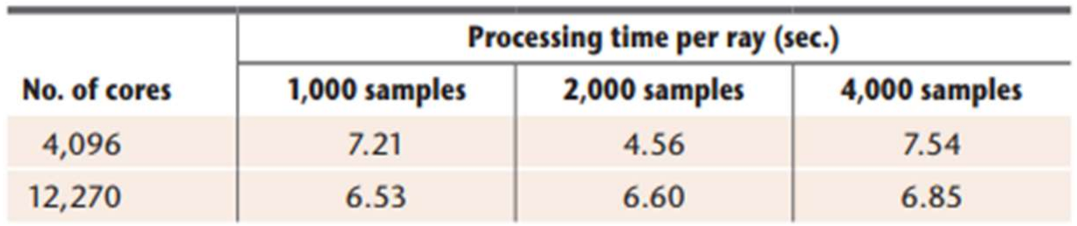
- Weak scaling of Isosurfacing.



Algorithm	No. of cores	Time (sec.)		
		Minimum	Maximum	Average
Calculate radiation*	4,096	0.180	0.250	0.2100
	12,270	0.190	0.250	0.2200
Isosurface†	4,096	0.014	0.027	0.0180
	12,270	0.014	0.027	0.0170
Render (on core)‡	4,096	0.020	0.065	0.0225
	12,270	0.021	0.069	0.0230
Render (across cores)‖	4,096	0.048	0.087	0.0520
	12,270	0.050	0.091	0.0530

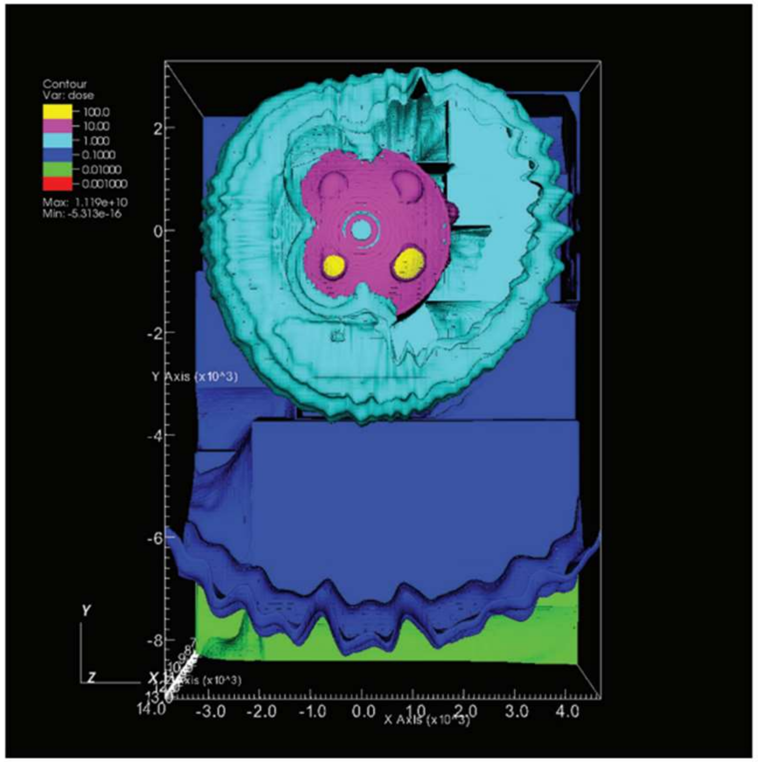
* *The time to calculate the linear combination of the 27 scalar fluxes.*
† *The isosurface algorithm's execution time.*
‡ *The time to render that core's surface.*
‖ *The time to combine that image with the other cores' images.*

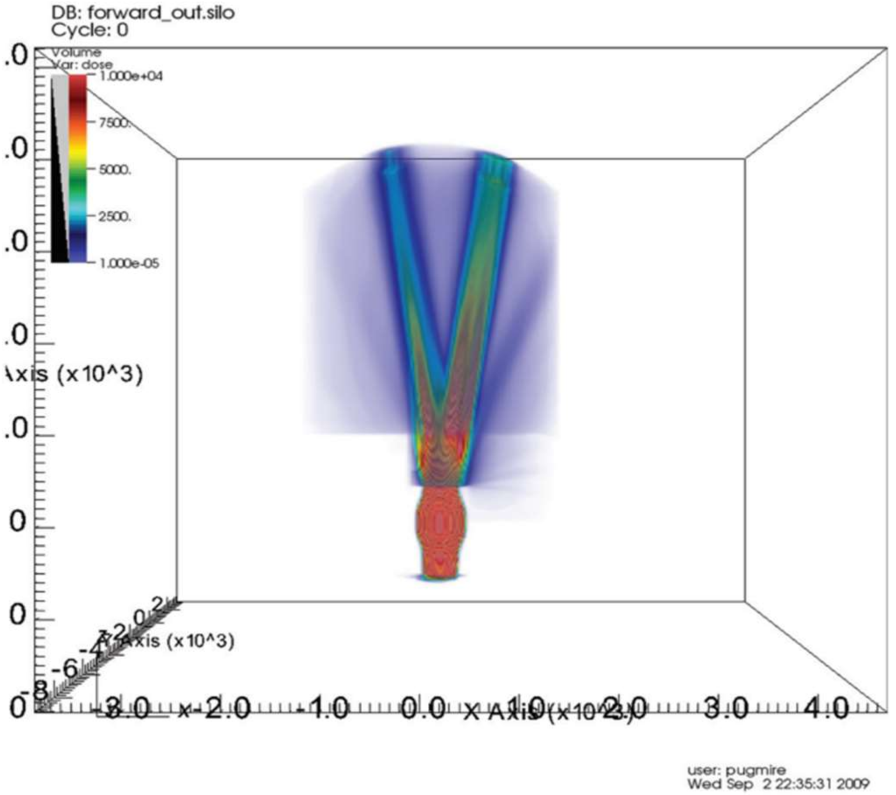
- Weak scaling of Volume Rendering.



No. of cores	Processing time per ray (sec.)		
	1,000 samples	2,000 samples	4,000 samples
4,096	7.21	4.56	7.54
12,270	6.53	6.60	6.85

Visualization Results





Visualization results for the Denovo calculation, produced by VisIt using 12,270 cores of JaguarPF: (a) a rendering of an isosurface and (b) a volume rendering of the data.

Pitfalls at Scale

- The inefficient code existed at various levels of software, from core algorithms (volume rendering), to code supporting algorithms (status updates), to foundational code (plug-in loading).

- They discuss pitfalls at various levels that are metioned above:

  - Volume Rendering

  - All-to-One Communication

  - Shared libraries and Start-Up Time

Pitfalls at Volume Rendering

- The volume-rendering code initially used a method that involved an $O(n^2)$ buffer, where "n" is the number of cores.
- This buffer was used during an all-to-all communication phase, which redistributed data samples along rays according to a dynamic partitioning strategy.
- The buffer became contained mostly empty spaces (zeroes), became inefficient and problematic as the number of cores increased.
- This approach caused significant memory usage, leading to the system running out of memory at large scales.

Solution to Volume Rendering Pitfall

- Eliminate the optimization that was intended to minimize the number of samples needing communication.
- Instead, they chose to assign pixels to cores without worrying about where individual samples lay.

Results after Correction

- Ray casting performance was approximately five seconds per frame for a 1024x1024 image.

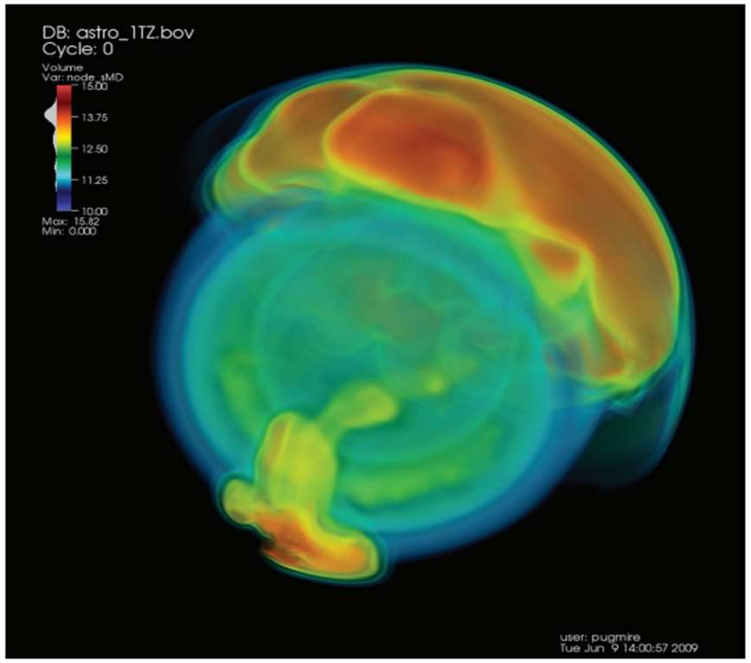
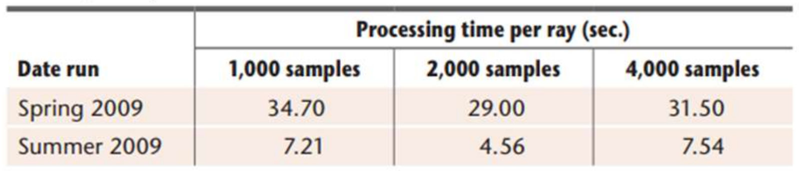


Figure 5. Volume rendering of one trillion cells, visualized by VisIt on JaguarPF. As expected, we ran into many pitfalls when running at high levels of concurrency. In this case, VisIt's volume-rendering algorithm had to be modified to remove an $O(n^2)$ algorithm.

- For weak scaling study Denovo data , running with 4096 cores, the speedup was approximately a factor of five.

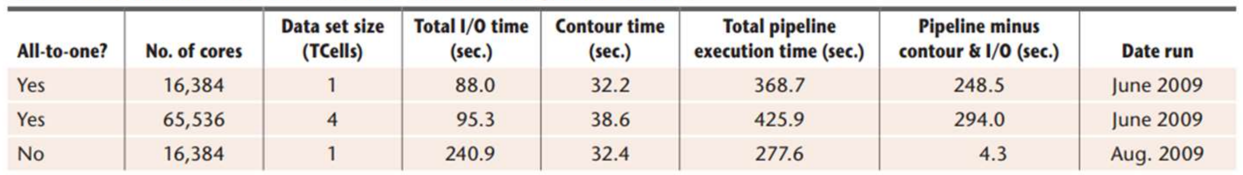


	Processing time per ray (sec.)		
Date run	**1,000 samples**	**2,000 samples**	**4,000 samples**
Spring 2009	34.70	29.00	31.50
Summer 2009	7.21	4.56	7.54

All-to-One Communication

- Pitfall 1 : Slowdown due to point to point communication
  - Delays caused by each core reporting its status and some metadata (such as extents) to a single MPI task (task 0) through point-to-point communication after pipeline execution.

Table 8. Performance with old versus new status-checking code, on Dawn.



All-to-one?	No. of cores	Data set size (TCells)	Total I/O time (sec.)	Contour time (sec.)	Total pipeline execution time (sec.)	Pipeline minus contour & I/O (sec.)	Date run
Yes	16,384	1	88.0	32.2	368.7	248.5	June 2009
Yes	65,536	4	95.3	38.6	425.9	294.0	June 2009
No	16,384	1	240.9	32.4	277.6	4.3	Aug. 2009

- Solution to pitfall 1:
  - The solution to this problem was to switch from point-to-point communication to tree communication.
  - This change helped reduce the time spent waiting for status and extents updates, leading to more efficient execution.

All-to-One Communication

- Pitfall 2 : Slowdown in I/O times
  - The I/O servers backing the file system became unbalanced in their disk usage, which caused the algorithm that assigns files to servers to switch from a round-robin scheme to a statistical scheme(Poisson Distribution).
  - This switch resulted in files no longer being assigned uniformly across I/O servers, leading to some servers being overloaded with more(three or four times) files than others.
- Solutions to pitfall 2:
  - There is no solution given in the paper, but the broader context suggests that addressing such issues would require better balancing of I/O resource, improved file assignment algorithms. Eg. Dynamic Load Balancing, Weighted Scheduling etc.

Pitfalls in Start-Up Time and Solution

- As the number of cores increased, each core attempted to read plug-in information from the file system simultaneously, creating contention for I/O resources.
- This contention resulted in long start-up times, which worsened as more cores were involved, taking as long as five minutes in some cases.
- They modified VisIt's plug-in infrastructure so that plug-in information could be loaded on MPI task 0 (a single core) and then broadcast to all other cores, rather than each core loading the information independently. This modification made plug-in loading nine times faster, significantly reducing the start-up time.

Pitfalls due to Shared Libraries and Solution

- Startup time was still slow due to the use of shared libraries in VisIt which allowed new plug-ins to access symbols not used by the current VisIt routines.
- A solution suggested by them would be to compile static versions of VisIt for high-concurrency cases, eliminating the need for shared libraries and further reducing start-up time.
- This approach would work because new plugins are frequently developed at lower levels of concurrency.

Conclusions

- Pure Parallelism does scale but is only as good as supporting I/O infrastructure.

- I/O was major focus of our study because slow I/O prevented interactive rates when loading data.

- Software and Hardware solution that might address this problem are:
  - Multiresolution techniques and data subsetting limit how much data is read, whereas in situ visualization avoids I/O altogether.

  - An increased focus on balanced machines that have I/O bandwidth in proportion with computing power will reduce I/O time.

QnA