In Situ Visualization for Large-Scale Combustion Simulations

Hongfeng Yu ■ *Sandia National Laboratories*

Chaoli Wang ■ *Michigan Technological University*

Ray W. Grout and Jacqueline H. Chen ■ *Sandia National Laboratories*

Kwan-Liu Ma ■ *University of California, Davis*

Supercomputers' massive processing power drives scientific discovery in many areas, and their computing power and storage capacity grow rapidly every year. As scientists gain access to more powerful machines, they attempt to solve larger, more complex problems. Consequently, the amount of data their scientific simulations generate is increasing at an astounding pace. Large-scale scientific simulations typically output time-varying multivariate volumetric data. Such data can now easily occupy petabytes of storage space. To maximize simulation output's use, scientists need efficient, effective solutions to manage and study the ever-growing data volume.

Supercomputer time is a prized commodity. Scientists from various fields compete for this time at national laboratories and universities and must use it efficiently. So, many scientists employ a batch-oriented, sequential process to analyze large data sets. A common strategy is to dump as much raw data during the simulation run as the storage capacity allows. Subsequent data analysis and visualization occur offline. This offline postprocessing can involve reducing petabytes of simulation-generated data to a more manageable size. Additional postprocessing might be necessary to prepare the data for visualization.

Occasionally, scientists directly access visualization clusters at supercomputing centers. During the simulation run, they can collect measurements that provide an overview of the simulation results. With help from visualization specialists, the scientists then can visually analyze their data. In this scenario, all visualization calculations take place on the visualization machine through *coprocessing*, which can occur online or offline. The system then delivers the visualization results, such as images or animations, directly to the scientists' desktop. If this option isn't available, the scientists could use a separate parallel computer to prepare their data for visualization.

However, even when coprocessing is available, transferring and storing simulation output can be formidable. Scientists usually don't have the capability to routinely transfer and store petabytes of data from supercomputer storage devices to their laboratories for study. Transferring that much data is time-consuming even if sufficient storage space is available. To reduce the amount of data to store and transfer, the common practice is to store selected time steps and study a limited range or a coarse-grained temporal resolution of the data. Although convenient, examining only a limited subset of large-scale data might hinder subsequent analysis of the overall phenomenon and selected details.

As the petascale-computing era approaches, simply transferring raw simulation data to storage devices or visualization machines is increasingly cumbersome. A better solution is to reduce or transform the data in situ on the same machine as

As scientific supercomputing moves toward petascale and exascale levels, in situ visualization stands out as a scalable way for scientists to view the data their simulations generate. This full picture is crucial particularly for capturing and understanding highly intermittent transient phenomena, such as ignition and extinction events in turbulent combustion.

Published by the IEEE Computer Society 0272-1716/10/$26.00 © 2010 IEEE

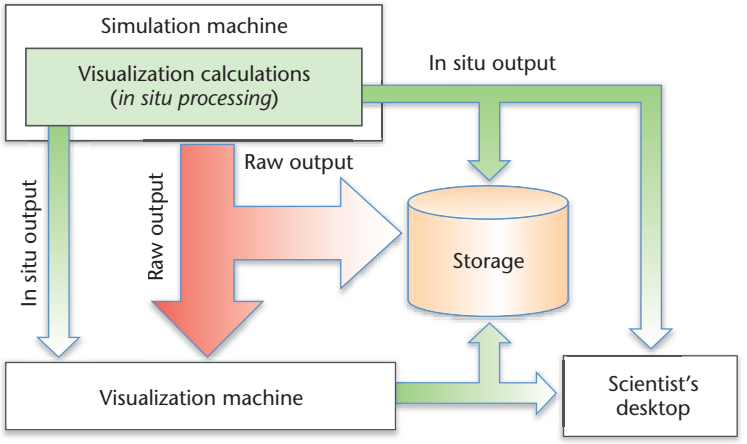


Figure 1. Comparing postprocessing, coprocessing, and in situ processing. Postprocessing and coprocessing (red arrows) involve transferring raw data. In situ processing provides a highly scalable solution for dealing with the extreme-scale data that scientific simulations produce.

the simulation runs, minimizing the data or information requiring storage or transfer.[1,2] As Figure 1 shows, in situ processing doesn't require transferring large quantities of raw data over the network, so it's highly scalable for handling the data petascale scientific simulations produce. (In situ visualization isn't a new concept; see the "Simulation-Time Visualization" sidebar.) In this case study of in situ visualization for turbulent-combustion simulations, we investigate in situ data processing and visualization strategies in a massively parallel environment. Detailed experimental results demonstrate that in situ visualization is a promising direction for accelerating high-performance supercomputing and scientific discovery.

The Challenges of In Situ Visualization

Researchers seldom perform simulation and visualization on the same parallel computer. Supercomputer time is expensive and sometimes difficult to acquire. So, many scientists are reluctant to use their supercomputing time for visualization calculations. Additionally, not all simulation codes can share data seamlessly with the codes that perform visualization calculations. Some visualization calculations can also be computationally expensive and impractical for in situ processing. Moreover, integrating visualization with simulation demands that visualization researchers and domain scientists collaborate closely over an extended period. Such a long-term commitment isn't always guaranteed.

Compared with traditional postprocessing visualization, in situ visualization has unique challenges.[1] First, the visualization code must interact directly with the simulation code. To optimize memory use, the simulation and visualization codes must share the same data structures to avoid data replication.

Second, balancing the visualization workload is more difficult because the visualization must comply and tightly couple with the simulation architecture. For stand-alone processing, researchers can parallelize visualization algorithms by partitioning and distributing data to best suit their visualization needs. In contrast, for in situ visualization, the simulation code dictates data partitioning and distribution. Moving data frequently among processors isn't an option for visualization processing. The visualization workload must be balanced so that the visualization is as scalable as the simulation.

Finally, visualization calculations must not incur excessive costs, with decoupled I/O delivering rendering results while the simulation is running. Researchers can't hardware-accelerate visualization calculations on the supercomputer because no graphics hardware is available. To keep costs down, scientists need an alternative that simplifies the calculations such that visualization accounts for only a small fraction of the simulation time. Through our experiment with a large-scale turbulent-combustion simulation using S3D, a Sandia DNS (direct numerical simulation) solver, we show that in situ visualization is exactly such an alternative. (For more on DNS, see the related sidebar.)

Parallel Rendering

The combustion simulation produces volume and particle data, so our in situ visualization solution integrates both volume and particle rendering.

Volume Rendering

Data partitioning for volume rendering comes directly from the domain decomposition for simulation. Achieving seamless rendering along data partition boundaries requires duplicating the data along data region boundaries (two voxels wide).

To propagate boundary information in 3D, a processor assigned with a data region must communicate with its 26 neighboring processors. We use diagonal communication elimination[3] and follow the communication in the x, y, and z directions, in turn. With this method, a processor need communicate with only six neighbors for exchanging boundary information.

After boundary exchange, a processor renders its data region using software ray casting and generates the corresponding partial image for compositing. Because the complete simulation data are available for in situ visualization, one ray-casting pass can render multiple variables.

Particle Rendering

For particle rendering, we implement a software-based point sprite technique. With point sprites,

Simulation-Time Visualization

Researchers have developed several efficient methods and guidelines for data processing in massively parallel computing environments. One viable approach is simulation-time visualization: studying data as simulations generate it.

In this scheme, researchers conduct simulation and visualization calculations on the same parallel supercomputer so that the two processes can share the data. Such visualization can render images or extract features directly from the raw data. These results are generally much smaller than the raw data and storable for later examination. Reducing both data transfer and storage costs early in the data analysis pipeline optimizes scientific discovery's overall process. Many researchers have experimented with this approach. Although they demonstrated runtime simulation visualization on parallel computers, the systems and problems were fairly small.[1–3]

Simulation-time visualization also lets scientists visually monitor the simulation while it runs. For example, SCIRun provided a computational-steering environment that supported runtime simulation tracking.[4] An object-oriented data-flow approach let users control scientific simulations interactively by varying boundary conditions, model geometries, and computational parameters.

Tiankai Tu and his colleagues demonstrated how to effectively monitor a terascale earthquake simulation running on a supercomputer's thousands of processors.[5] They performed rendering in situ using the same data-partitioning scheme they adopted for the simulation. So, data movement among processors was unnecessary. Over a wide-area network, they could interactively change view angles, adjust sampling steps, edit color and opacity transfer functions, and zoom in and out to visually monitor the simulation runs.[6] These visualization calculations' time and storage overhead were almost negligible, making this approach attractive.

References

1. A. Globus, *A Software Model for Visualization of Time Dependent 3-D Computational Fluid Dynamics Results*, tech. report RNR 92-031, NASA Ames Research Center, 1992.
2. K.-L. Ma, "Runtime Volume Visualization of Parallel CFD," *Proc. Int'l Parallel Computational Fluid Dynamics Conf.* (ParCFD 95), Inst. Computer Applications in Science and Eng., 1995, pp. 307–314.
3. J. Rowlan et al., "A Distributed, Parallel, Interactive Volume Rendering Package," *Proc. IEEE Visualization Conf.* (VIS 94), IEEE Press, 1994, pp. 21–30.
4. S.G. Parker and C.R. Johnson, "SCIRun: A Scientific Programming Environment for Computational Steering," *Proc. ACM/IEEE Supercomputing Conf.* (SC 95), IEEE CS Press, 1995, article 52, doi:10.1145/224170.224354.
5. T. Tu et al., "From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing," *Proc. 2006 ACM/IEEE Conf. Supercomputing* (SC 06), IEEE CS Press, 2006, article 91, doi:10.1145/1188455.1188551.
6. H. Yu et al., "Remote Runtime Steering of Integrated Terascale Simulation and Visualization," *Proc. 2006 ACM/IEEE Conf. Supercomputing* (SC 06), ACM Press, 2006, article 297, doi:10.1145/1188455.1188767.

we simply render a screen-aligned 2D quad centered at a particle's vertex.

The quad's scale is based on the distance between the eye and the particle in 3D under perspective projection. We use procedural texturing to shade each quad. More specifically, we look up a precalculated sphere normal map to obtain the normal for each pixel in the quad. Then, we use the normal to calculate the lighting using the Phong model and the eye and light directions.

We interpolate each pixel's depth between the particle's depth (the sphere's center) and the sphere boundary's depth, which we use to accurately depict the spheres' intersection in 3D. Because the system performs all calculations in the image space, we can render tens or hundreds of thousands of particles more efficiently using point sprites rather than rendering 3D sphere geometry.

Integrating Volume and Particle Rendering

Our approach draws each particle as a sphere with a given radius. So, some particles lie across the data region boundary. As Figure 2 shows, we must handle these particles and ensure that they blend correctly along the boundaries when integrated with volume rendering. Similar to volume

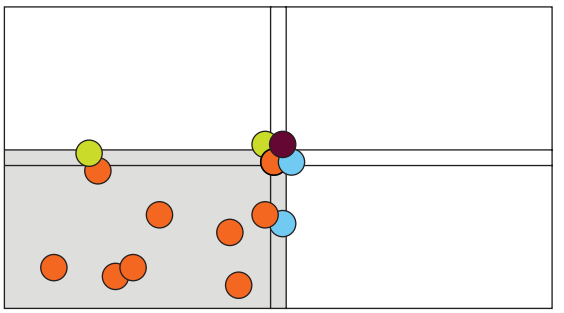


Figure. 2. Handling particles along neighboring data region boundaries. The processor assigned to the gray data region must exchange particles along the boundary with neighboring processors so that the particles can blend correctly when integrated with volume rendering. The particles from neighboring data regions are blue, green, and brown.

Direct Numerical Simulation

Numerical simulation plays a valuable role in analyzing and assessing turbulent flames. As long as the fluids in question can fall on a continuum, Navier-Stokes equations provide a useful description of the flow. When species transport and energy evolution equations augment the flow, the resulting system of differential equations is theoretically solvable (assuming a suitable reaction mechanism is available). This solution would let researchers predict the performance and emissions of myriad energy conversion devices involving combustion, such as gas turbines and automotive engines. Frustratingly, this tantalizing possibility is beyond the scope of computational capabilities now and for the foreseeable future, Moore's law notwithstanding.

Solutions to the Navier-Stokes equations are notoriously chaotic. The smallest scale that affects the overall solution determines the maximum grid spacing when solving equations numerically. This scale is typically three to five orders of magnitude smaller than the overall domain size. When the domain size or the velocity of the flow's largest structure increases, the smallest scales' size decreases further. Chemical reactions add yet smaller scales requiring resolution. The equations' direct numerical solution (DNS) is tractable only for the simplest problems: the Sandia DNS solver S3D does exactly this, solving the governing equations for canonical small-laboratory-scale problems.

To solve real-world combustion problems, some engineering models filter the governing equations in time or space to give a set of equations for large-scale behavior. Denis Veynante and Luc Vervisch provided a comprehensive review.[1] These methods require a model for the physical phenomena that are essential to the broader solution. However, researchers typically resolve on the coarser grids they normally use. Unique DNS benchmark simulations provide validation data useful for exploring the reasons for existing models' shortcomings and developing of new approaches.

S3D and Combustion Simulation

S3D is a Runga-Kutta integrator that advances the balance equations for conserving mass, momentum, energy, and chemically reactive species.[2] For example, the conservation equation for the mass fraction of chemically reactive species α, Y_α, is

$$\frac{\partial \rho Y_\alpha}{\partial t} = -\frac{\partial \rho u_i Y_\alpha}{\partial x_i} - \frac{\partial \rho V_{i\alpha} Y_\alpha}{\partial x_i} + \omega_\alpha .$$

S3D evaluates the chemical-source term ω_α from a reaction mechanism that couples the equations describing the thermodynamic state.

Using an eighth-order approximation, S3D approximates the spatial derivatives at each point on a finite-difference grid. Figure A shows the solution loop. The most rapidly varying quantities limit the maximum time step. Because S3D solves a fully compressible form of the Navier-Stokes equations, the acoustic timescale forms an upper bound on the time step. The practical implication is that the major species don't change significantly over a timescale of 10 to 100 time steps.

However, traditionally, simulation codes save restart files even less frequently than this because of storage requirements and I/O time restrictions. Native restart files contain the primary solution variables—pressure, temperature, velocity, and composition. The species mass fractions for each molecular species in the chemical mechanism describe these variables.

In a recent simulation of a lifted flame that is autoignition[3] stabilized, the necessary resolution is 15 μm over an overall domain of approximately 2 cm. (Autoignition

rendering, data partitioning for particle rendering follows the domain decomposition for the simulation. Particles exchange similarly for voxels lying on the boundary.

Our data partitioning for volume and particle rendering follows the scheme the simulation adopted. Data exchange happens for only boundary regions, minimizing data communication. To integrate volume rendering with particle rendering, we pay careful attention to their drawing order to preserve proper blending and depth order. We use this integration algorithm:

1. Each processor renders the particles in its data region.
2. Each processors exchanges particles along its region's boundaries with particles along the boundaries of its neighbors' regions.
3. Each processor renders the particles along its and its neighbors' boundaries.
4. We read out the particle rendering's red, green, blue, and alpha (RGBA) and depth channels.
5. We perform volume ray casting with depth lookup of the particle image for correct blending.

The algorithm employs our software-based frame buffer object (FBO) implementation. First, each processor creates an FBO consisting of a color buffer (RGBA, 32-bit, floating-point channels) and a depth buffer (32-bit, floating-point). It then binds the FBO as the rendering target. The processor renders particles in its data region into the FBO, in which user-specified transfer functions define a particle's color and visibility (opaque or transparent). During rendering, the algorithm selects the visible particles along the boundaries and buffers them for exchange.

is useful for studying flame anchoring behavior in a hot environment.) So, the simulation requires more than 1.3 × 10^9 grid points. This simulation's chemical mechanism includes 22 species to describe ethylene-air combustion: the restart file size is more than 140 Gbytes.

Incorporating Visualization

Three main reasons exist for incorporating visualization into the solution algorithm. To limit the total I/O time and data size, the simulation outputs restart files every 1 μm (200 time steps). Especially for minor radical species, interesting effects related to stabilization might occur more rapidly than this.

Also, when a new simulation case is being set up, subtle interactions between the boundary conditions and the interior solution during initial start-up transients can produce errors. Diagnosing these errors can be difficult because the next save of the restart file can create a large-scale instability that obscures the event that triggered the original instability.

Once a solution is complete, the workflow for generating volume renderings includes reloading each restart file into S3D and writing a subsampled representation of the data to visualize. These steps incur significant I/O overhead; time series or animations can exacerbate the situation.

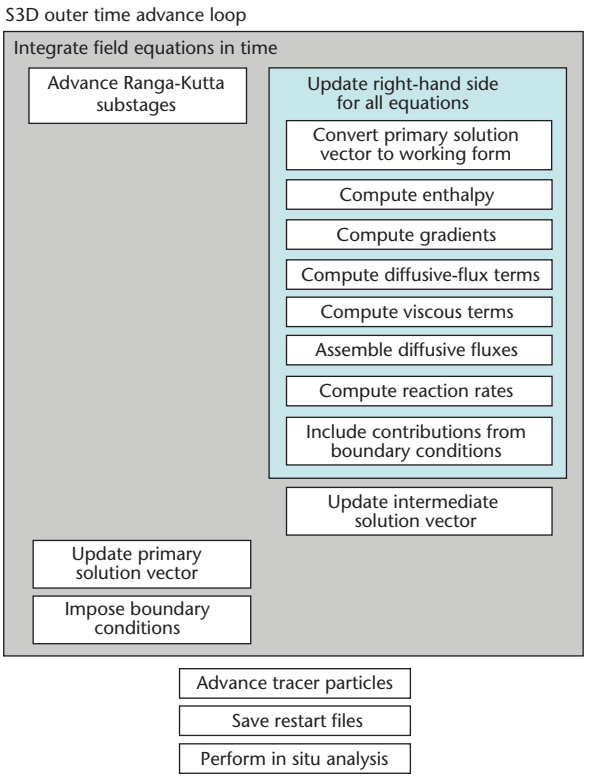


Figure A. The S3D time advance loop, showing all the major tasks. This figure provides an overview of the S3D execution pipeline for combustion simulation.

References

1. D. Veynante and L. Vervisch, "Turbulent Combustion Modeling," *Progress in Energy and Combustion Science*, vol. 28, nos. 11–12, 2002, pp. 193–266.
2. J.H. Chen et al., "Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D," *Computational Science & Discovery*, vol. 2, 2009, article 015001, doi:10.1088/1749-4699/2/1/015001.
3. C.S. Yoo et al., "DNS of a Turbulent Lifted Ethylene/Air Jet Flame in an Autoignitive Coflow—Stabilization and Flame Structure," *Proc. Int'l Workshop Measurement and Computation of Turbulent Nonpremixed Flames* (TNF 09), 2008, pp. 296–297.

Next, processors exchange particles along the boundaries and ensure that particles from neighbors render and blend correctly using the depth buffer. We then save the FBO's color buffer and depth buffer for integrating with volume rendering.

Finally, we perform regular ray casting on the volume. However, we also calculate along a ray each sample's depth and check it with the depth buffer. We assume all visible particles are opaque, so we can stop ray casting for each pixel when the current depth value exceeds the value in the depth buffer. We then accumulate the color value recorded in the color buffer. This way, we perform a depth-accurate integration of volume and particle rendering (see Figure 3).

Image Compositing

In choosing an algorithm for parallel image compositing for in situ visualization, we narrowed our choice to 2-3 swap, a generalization of the popular binary-swap algorithm. (For more on image compositing in parallel, see the "Parallel Image Compositing" sidebar.)

Binary Swap

Over the past 15 years, binary swap has become the de facto algorithm for parallel image compositing. Unlike direct send (see the "Parallel Image Compositing" sidebar), which requires n participating processors to exchange $O(n^2)$ messages, binary swap requires only $O(n \log n)$ messages. This algorithm involves a multistage process and pairs up the processors using a binary compositing tree. At any image-compositing stage, a processor communicates only with its counterpart in the pair.

Parallel Image Compositing

Parallel rendering comprises five stages: data partitioning, data distribution, rendering, image compositing, and image delivery. Steven Molnar and his colleagues treated it as a sorting problem and introduced the sorting classification.[1] Depending on where the sorting happens in the graphics pipeline, they defined three classes: sort-first, sort-middle, and sort-last.

Over the years, researchers have more widely used sort-last parallel rendering because of its simple task decomposition for achieving load balancing.[2–4] Image compositing (blending partial images in the correct depth order) in sort-last rendering demands interprocessor communication. This requirement can become expensive when using more than hundreds of processors because of the potentially large quantity of messages the processors exchange. So, image compositing often becomes a bottleneck affecting sort-last rendering's efficiency.

Since the early 1990s, researchers have developed various image-compositing methods for parallel visualization applications. For sort-last volume rendering, the most popular and representative solutions are *direct send* and *binary swap*. Direct send is the simplest technique.[5] It's flexible with network interconnection and the number of processors it uses. However, it introduces link contention because it requires all-to-all communication. Aleksander Stompel and his team optimized direct send for small to midsize CPU clusters with a precomputed compositing schedule, but the schedule's computation doesn't scale to thousands of processors.[6]

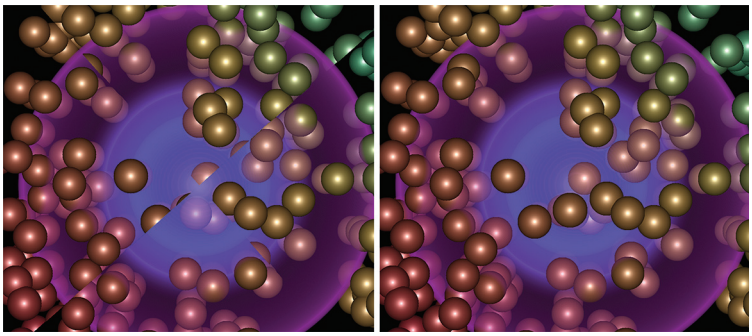
Binary swap uses all processors at all compositing stages.[7] Although it perfectly balances the compositing workload among processors and reduces the number of messages they exchange, it suffers from restrictions regarding the number of processors.

Recently, Hongfeng Yu and his colleagues presented the *2-3 swap* compositing algorithm, which combines direct send's flexibility and binary swap's optimality.[8] They demonstrated 2-3 swap's scalability on a supercomputer with thousands of processors. (For more on binary swap and 2-3 swap, see the section "Image Compositing" in the main article.)

Researchers have also developed hardware-based image-compositing equipment and solutions, such as Lightning-2[9] and Sepia-2.[10] Although these solutions achieve impressive performance with high scalability, building a large-scale visualization system using such hardware can be prohibitively expensive. In response, Jorji Nonaka's team presented a hybrid image-compositing method for sort-last rendering on graphics clusters with the MPC (Mitsubishi Precision Co.) image compositor.[11,12] This method offers high performance at a reasonable cost.

More recently, Nonaka and Kenji Ono proposed a decomposition approach for optimizing large-scale parallel image composition on multicore MPP (massively parallel processing) systems. This approach uses multicore processors for direct send, leveraging the fast communication speed within a CPU.[13] David Pugmire and his colleagues described using a network processing unit for hardware-based image compositing in a distributed rendering system.[14] With commodity graphics hardware's popularity, researchers also explored using GPU clusters for hardware-accelerated parallel visualization.[15] Such a cluster, however, is typically limited to a small scale.

Software image compositing remains the most widely used solution in many applications. In a supercomput-



(a) **(b)**

Figure. 3. Integrating volume and particle rendering. The synthesized volume data set comprises multiple-layer spheres. (a) Not exchanging and handling the particles along the boundary results in incorrect rendering along the diagonals. (b) Exchanging those particles results in correct rendering.

However, the number of processors that binary swap uses isn't flexible. It works best when the number of processors is an exact power of two. In a real network environment, processors are often down for various reasons, meaning that the optimal processor count can deviate from a power of two. To apply binary swap in this situation (for example, $2^{k-1} < n < 2^k$), we send the image data from $n - 2^{k-1}$ processors to the remainder of 2^{k-1} processors. We then perform binary swap directly on the 2^{k-1} processors.

Another solution is to use the binary compositing tree with $k + 1$ levels. In this scenario, the binary tree is complete (with 2^k leaf nodes), but only valid leaf nodes participate in the compositing. The compositing partner computation requires additional complexity. However, neither solution provides optimal parallelism or compositing efficiency.[4]

2-3 Swap

Hongfeng Yu and his colleagues solved binary swap's non-power-of-two cases by introducing the 2-3 swap algorithm.[4] It also involves multistage

ing environment with thousands of processors, dedicated graphics hardware for image compositing isn't practical because of the hardware costs and effort for integration, upgrades, and maintenance.

References

1. S. Molnar et al., "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, 1994, pp. 23–32.
2. J.P. Ahrens and J.S. Painter, "Efficient Sort-Last Rendering Using Compression-Based Image Compositing," *Proc. Eurographics Workshop Parallel Graphics and Visualization* (EGPGV 98), Eurographics, 1998, pp. 145–151.
3. K. Moreland, B. Wylie, and C. Pavlakos, "Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* (PVG 01), IEEE Press, 2001, pp. 85–92.
4. C. Wang, J. Gao, and H.-W. Shen, "Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing," *Proc. Eurographics Workshop Parallel Graphics and Visualization* (EGPGV 04), Eurographics, 2004, pp. 23–30.
5. W.M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," *Proc. IEEE Symp. Parallel Rendering* (PRS 93), ACM Press, 1993, pp. 7–14.
6. A. Stompel et al., "SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* (PVG 03), IEEE CS Press, 2003, pp. 33–40.
7. K.-L. Ma et al., "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," *Proc. 1993 IEEE Symp. Parallel Rendering* (PRS 93), IEEE Press, 1993, pp. 15–22.
8. H. Yu, C. Wang, and K.-L. Ma, "Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing," *Proc. 2008 Conf ACM/IEEE Supercomputing* (SC 08), IEEE Press, 2008, article 48, doi:10.1145/1413370.1413419.
9. G. Stoll et al., "Lightning-2: A High-Performance Display Subsystem for PC Clusters," *Proc. Siggraph*, ACM Press, 2001, pp. 141–148.
10. S. Lombeyda et al., "Scalable Interactive Volume Rendering Using Off-the-Shelf Components," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics* (PVG 01), IEEE Press, 2001, pp. 115–121.
11. J. Nonaka et al., "Hybrid Hardware-Accelerated Image Composition for Sort-Last Parallel Rendering on Graphics Clusters with Commodity Image Compositor," *Proc. IEEE Symp. Volume Visualization and Graphics* (VolVis 04), IEEE CS Press, 2004, pp. 17–24.
12. S. Muraki et al., "Next-Generation Visual Supercomputing Using PC Clusters with Volume Graphics Hardware Devices," *Proc. 2001 ACM/IEEE Conf. Supercomputing* (SC 01), ACM Press, 2001, p. 44, doi:10.1145/582034.582085.
13. J. Nonaka and K. Ono, "A Decomposition Approach for Optimizing Large-Scale Parallel Image Composition on Multi-core MPP Systems," *Proc. Eurographics Workshop Parallel Graphics and Visualization* (EGPGV 09), Eurographics, 2009, pp. 71–78.
14. D. Pugmire et al., "NPU-Based Image Compositing in a Distributed Visualization System," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 4, 2007, pp. 798–809.
15. X. Cavin, C. Mion, and A. Filbois, "COTS Cluster-Based Sort-Last Rendering: Performance Evaluation and Pipelined Implementation," *Proc. IEEE Visualization Conf.* (VIS 05), IEEE Press, 2005, pp. 111–118.

image compositing. However, 2-3 swap partitions processors into groups, not pairs. This partitioning allows greater flexibility in the number of processors in a group. In the compositing tree, a node has either two or three children, hence the algorithm's name.

To reduce the number of messages exchanged, a processor communicates only with the other processors in its group. Yu's team showed that the number of messages any processor exchanges at any image-compositing stage is bounded by four. Combining direct send's flexibility and binary swap's optimality, 2-3 swap scales well to thousands of processors. As such, we chose it for in situ parallel image compositing.

Integrating Visualization with Simulation

To integrate visualization with simulation requires an interface between the simulation code and the visualization code.

The Simulation Side

First, the simulation initializes the visualization module. The simulation provides the size and coordinates of each processor's global domain and local partition. Each processor doesn't need any other processor's partition information. The simulation code also provides the pointer to the buffer of the local scalar-variable values. In addition, the visualization code must know the pointer to the particle data.

Next, the solver updates the scalar variable and particle data at each time step. At the same time, the simulation code invokes visualization calculations at a given rate. We implement all API functions on the visualization side; we modify the simulation code by adding a few function calls during initialization and solving.

The Visualization Side

A processor directly takes the data region assigned

to it during initialization and performs volume and partial rendering during the simulation run. The processor must calculate the local data region's depth and gather the depth values. Then, visibility sorting of all processors occurs. This sorting enables the system to build the 2-3 swap algorithm's compositing tree, in which the leaf nodes correspond to the sorted list of processors. Each processor calculates the compositing tree and communication schedule by itself, so communication is unnecessary.

This visibility sorting and scheduling is view dependent and recomputes whenever the view changes at runtime. Subsequently, all processors participate in image compositing. A host processor gathers the final image and saves it to the local disk or delivers it to the scientists' desktop.

Results

We tested our approach with a lifted-jet combustion simulation at Sandia National Laboratories. Our test environment was JaguarPF, the Cray XT5 at Oak Ridge National Laboratory's National Center for Computational Sciences.

We assigned each processor core a 27 × 40 × 40 region. We tested four numbers of cores: 240, 1,920, 6,480, and 15,360, with core configurations of 15 × 8 × 2, 30 × 16 × 4, 45 × 24 × 6, and 60 × 32 × 8, respectively.[5] (During the XT5 upgrade's second phase, we tested 240, 1,920, and 6,480 cores. At that time, the XT5 used 20,928 AMD x86 64 Opteron Barcelona quad-core 2.3-GHz processors connected through a SeaStar 2+ internal interconnect. After the XT5 upgrade's fourth phase, we performed the tests with 15,360 cores. At that time, the XT5 used 37,376 AMD x86 64 Opteron Istanbul six-core 2.6-GHz processors connected through a SeaStar 2+.)

Table 1 lists the volume and particle data sizes and timing results. The volume data consist of double floating-point (8 bytes) variables; the particle data consist of single floating-point (4 bytes) variables. We chose $2{,}048^2$, $1{,}024^2$, and 512^2 image resolutions and used 32-bit floating-point precision for the RGBA and depth channels for image compositing.

For the simulation, I/O, and visualization, we measured timing for one time step. Reaction rate evaluation and derivative evaluation (including temporal derivative calculation) dominate simulation time, whereas visualization time accounts for only a small fraction of it. For example, if we perform visualization at each simulation time step for 6,480 cores and $1{,}024^2$ image resolution, visualization time is approximately 6.92 percent of the simulation time; I/O time is more than four times the simulation time. In practice, we usually perform in situ visualization less frequently (every 10th time step), so the visualization time can be two orders of magnitude less than the overall simulation time.

Figure 4 shows the timing breakdown for the visualization stages. Image compositing, which requires interprocessor communication, dominates the total visualization time. In Figure 4a, volume-rendering time decreases as the core count increases. The data region size and output image resolution are fixed, whereas the whole volume size increases as the number of cores increases. So, each core gets a smaller screen projection and requires less rendering time. Compositing time increases as the core count increases because we implement only a nonoptimized 2-3 swap algorithm.

In Figure 4b, compositing time decreases as the output image resolution decreases, which results in a decreased compositing workload. In Figure 4c, we use the three data types (8-bit unsigned byte, 16-bit unsigned short, and 32-bit floating-point) for the image's RGBA channels. Compositing time decreases as image data precision decreases. Users can choose different image sizes and types to suit their compositing-performance and visualization quality needs.

We also rendered multiple images at the same time for multivariate-data visualization. Figure 5 shows the compositing time for multiple images with two compositing modes. The *separate-compositing mode* renders and composites only one image at a time. The *combined-compositing mode* renders and composites n images (corresponding to n variables) at the same time so that the number of messages the processors exchange for compositing is approximately $1/n$ of the separate-compositing mode's messages. However, in practice, the combined-compositing mode's performance gain is marginal. The XT5 has low message-passing-interface latency (approximately 1 μs) for network operations, and the processors exchange about the same quantity of image data for these two modes.

Figure 6 shows each core's visualization time, for a 240-core simulation run with a 512^2 output image resolution. As the figure shows, rendering, which includes boundary data exchange and particle and volume rendering, isn't balanced. This unevenness is due to the domain decomposition we inherit from the simulation and the transfer functions we use for volume and particle rendering. The overall time, however, is generally balanced among all cores because the dominant image-compositing time is well balanced. We can draw

Table 1. Timing breakdown for simulation, I/O, and visualization with various configurations.

	No. of cores			
	240	**1,920**	**6,480**	**15,360**
Volume rendering				
Volume size	405 × 320 × 80	810 × 640 × 160	1,215 × 960 × 240	1,620 × 1,280 × 320
No. of variables	27	27	27	27
Data size (Gbytes)	2.1	16.7	56.3	133.5
Particle rendering				
No. of particles (millions)	0.8	5.2	17.4	41.1
No. of variables	118	118	118	118
Data size (Gbytes)	0.3	2.5	8.3	19.5
Simulation time (sec.)				
Total	8.1659	8.6680	9.6293	11.867
Reaction rate evaluation	3.8779	3.8924	3.8900	3.3164
Mixture average diffusion calculation	1.2728	1.2948	1.3610	1.3041
Derivative evaluation	0.9246	1.2115	1.6240	3.5597
Other temporal derivative calculation	1.7332	1.8995	2.3677	3.2143
Runge-Kutta integration	0.3472	0.3536	0.3566	0.4052
Tracer advection	0.0102	0.0162	0.0300	0.0668
I/O time (sec.)	8.2675 (101.24%)*	25.611 (295.47%)	42.660 (432.64%)	136.690 (1,151.90%)
Visualization time with $2{,}048^2$ image resolution (sec.)				
Total	1.7699 (21.67%)	2.0459 (23.60%)	2.6689 (27.72%)	4.3595 (36.74%)
Boundary particle exchange	0.0012	0.0034	0.0041	0.0061
Particle rendering	0.1255	0.1953	0.2710	0.5314
Boundary voxel exchange	0.0023	0.0039	0.0051	0.0080
Volume rendering	0.0834	0.0499	0.0380	0.0230
Image compositing	1.5575	1.7934	2.3507	3.7910
Visualization time with $1{,}024^2$ image resolution (sec.)				
Total	0.4903 (6.00%)	0.6198 (7.15%)	0.6661 (6.92%)	1.0381 (8.75%)
Boundary particle exchange	0.0015	0.0035	0.0044	0.0063
Particle rendering	0.0387	0.0586	0.0759	0.1438
Boundary voxel exchange	0.0031	0.0042	0.0049	0.0081
Volume rendering	0.0212	0.0125	0.0094	0.0059
Image compositing	0.4258	0.5410	0.5715	0.8740
Visualization with 512^2 image resolution (sec.)				
Total	0.1304 (1.60%)	0.1782 (2.06%)	0.1978 (2.05%)	0.2879 (2.43%)
Boundary particle exchange	0.0012	0.0037	0.0046	0.0066
Particle rendering	0.0167	0.0260	0.0345	0.0649
Boundary voxel exchange	0.0023	0.0043	0.0050	0.0083
Volume rendering	0.0053	0.0031	0.0027	0.0015
Image compositing	0.1049	0.1411	0.1510	0.2066

**Percentages represent the ratio of I/O or visualization time to simulation time*

similar conclusions for runs with larger number of cores.

The total visualization time in Figure 6 is less than in Table 1. In Table 1, we simply add each stage's maximum time for the total visualization time. The result corresponds to the worst-case upper bound. In practice, however, these maximum times can occur at different cores. For example, a core with the maximum compositing time might not have the maximum rendering time. So, Figure 6 reports a smaller visualization time.

Figure 7 shows the volume-rendering results for six selected variables; Figure 8 shows detailed views of integrated volume and particle rendering

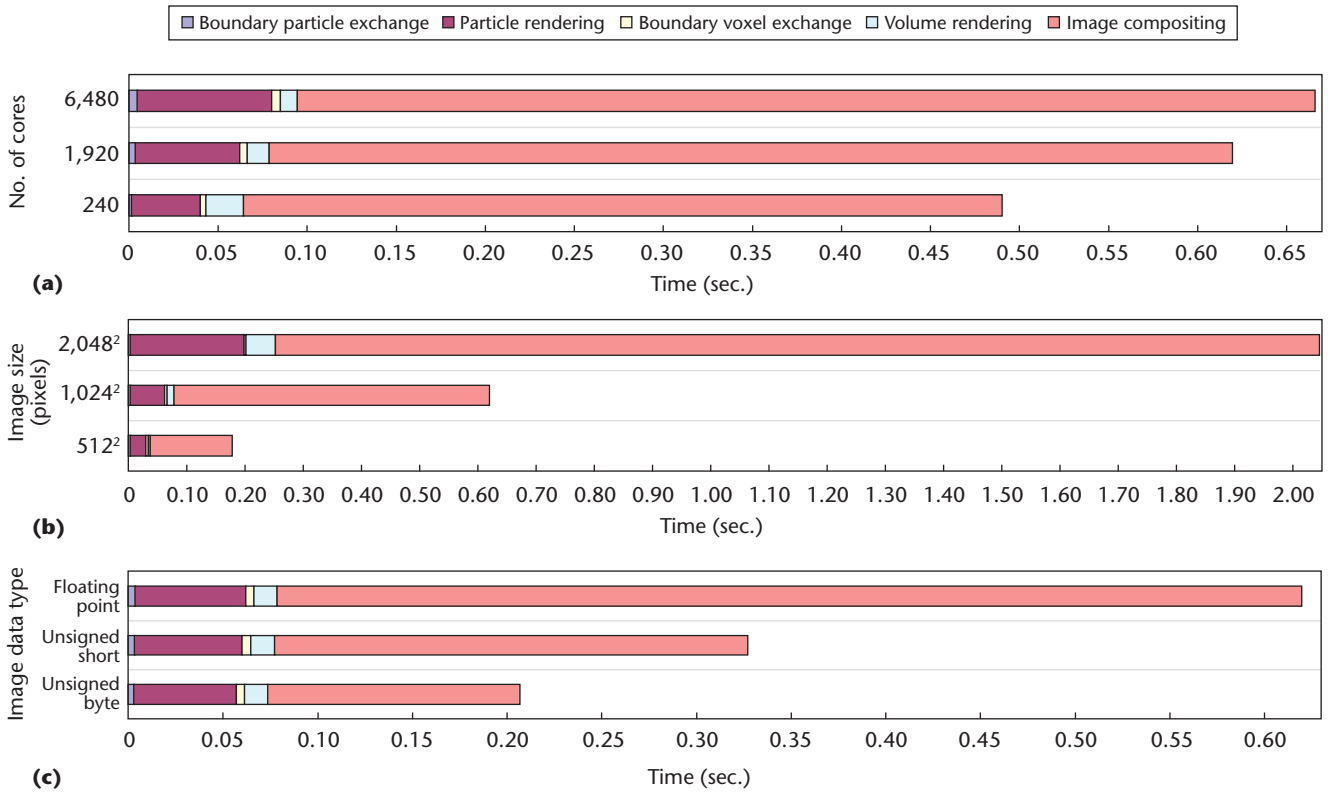


Figure 4. Timing for in situ visualization stages. (a) This chart shows different core counts with 1,024² output image resolution and 32-bit floating-point RGBA image data. Compositing time increases as the core count increases because we use a nonoptimized 2-3 swap algorithm. (b) For three output image resolutions with 1,920 cores and 32-bit floating-point RGBA image data, compositing time decreases as output image resolution decreases. The result is a decreased compositing workload. (c) For three output image data types with 1,920 cores and 1,024² output image resolution, compositing time decreases with decreased precision in the image's RGBA channels. The result is a decreased compositing workload.

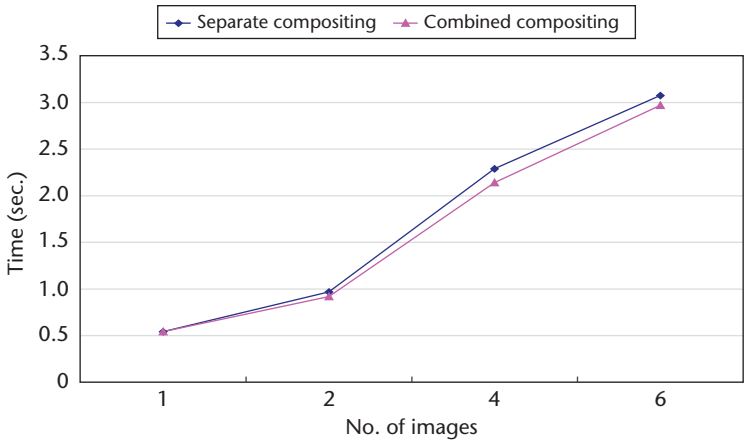


Figure 5. Compositing time for rendering multiple images with two compositing modes on 1,920 cores. The output image resolution is 1,024², with RGBA floating-point data. The performance gain for using the combined-compositing mode is marginal.

on some of these variables. Our ability to render volume and particle data simultaneously lets scientists study their data in greater detail. These in-situ-visualization results provide valuable visual feedback to the scientists for monitoring their simulations on the fly.

Discussion

Both the simulation and visualization codes require replicating data around the domain decomposition boundaries. Ideally, the two codes can also share these boundary data, but in practice, this isn't feasible. The simulation code uses four-voxel-wide boundaries for gradient and derivative calculations. To lower this memory overhead, it uses a single buffer to keep the boundary data for only one variable. That is, at the end of each iteration, the code keeps only one variable's boundary data. Although volume rendering uses only two-voxel-wide boundaries for seamless rendering, that buffer is unlikely to keep the variable values that the renderer needs.

Keeping boundary data for all variables on the simulation side incurs too much memory overhead and requires modifying the simulation code. So, in our implementation, the rendering code conducts an additional boundary exchange among cores to obtain the needed two-voxel-wide boundary data. According to our study, this communication overhead is quite acceptable. Most importantly, this decision eases the integration of the two codes.

As our test results show, image compositing re-

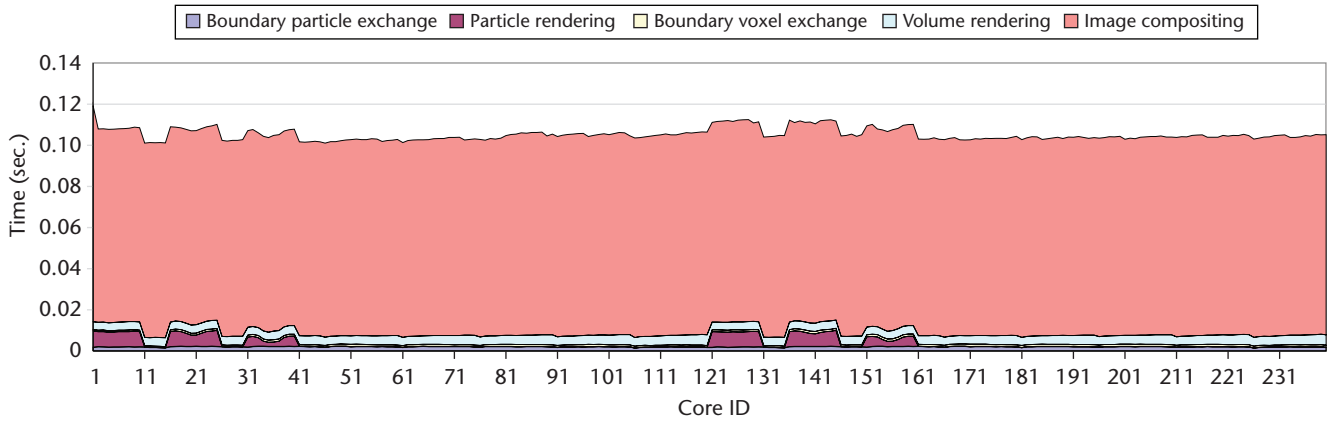


Figure. 6. The visualization time for each core of a 240-core simulation run with a 512^2 output image resolution. The overall time generally balances among all cores because the dominant image-compositing time is well balanced.

mains the bottleneck of parallel rendering at the scale we consider here. However, the overall rendering cost is acceptable for this in-situ-visualization setting. If the cost became too high in a different setting, we could reduce it by optimizing two aspects of the compositing algorithm. First, we currently use full-size partial images the processor cores produce as input. We can improve this by eliminating background pixel data and considering only effective pixel data for compositing. Second, our current implementation imposes synchronization at every compositing-tree level. This level-by-level synchronization is unnecessary owing to the 2-3 swap algorithm's local communication: as we mentioned before, a core communicates only with other cores in its own group at any image-compositing stage. This observation suggests a higher level of parallelism in which nodes at different compositing-tree levels can execute compositing tasks concurrently.

We implemented a client program that runs on a remote user's desktop or laptop and communicates with a simulation over the network (see Figure 9). Users can specify and send visualization configurations (such as transfer functions, views, number of images, and rendered variables) to the simulation and receive images at runtime. For the run-in-batch mode, we specify the transfer functions and views offline with data that a small set of sample runs generated. We then use these transfer functions and viewing parameters for in situ visualization. We recommend further research on in situ, automatic transfer function design and view selection[5] so that the visualization results better reveal the essence of simulation data.

With in situ visualization, scientists can capture features between time steps that the simulation code doesn't save. In a turbulent-combustion simulation, the frequency to save restart files depends on the *Kolmogorov timescale*—the smallest mechanical-flow timescale in a turbulent flow. For the lifted-jet simulation, the Kolmogorov timescale ranges from 2.5 to 10 μs, depending on the location in the flow. So, the restart-file frequency

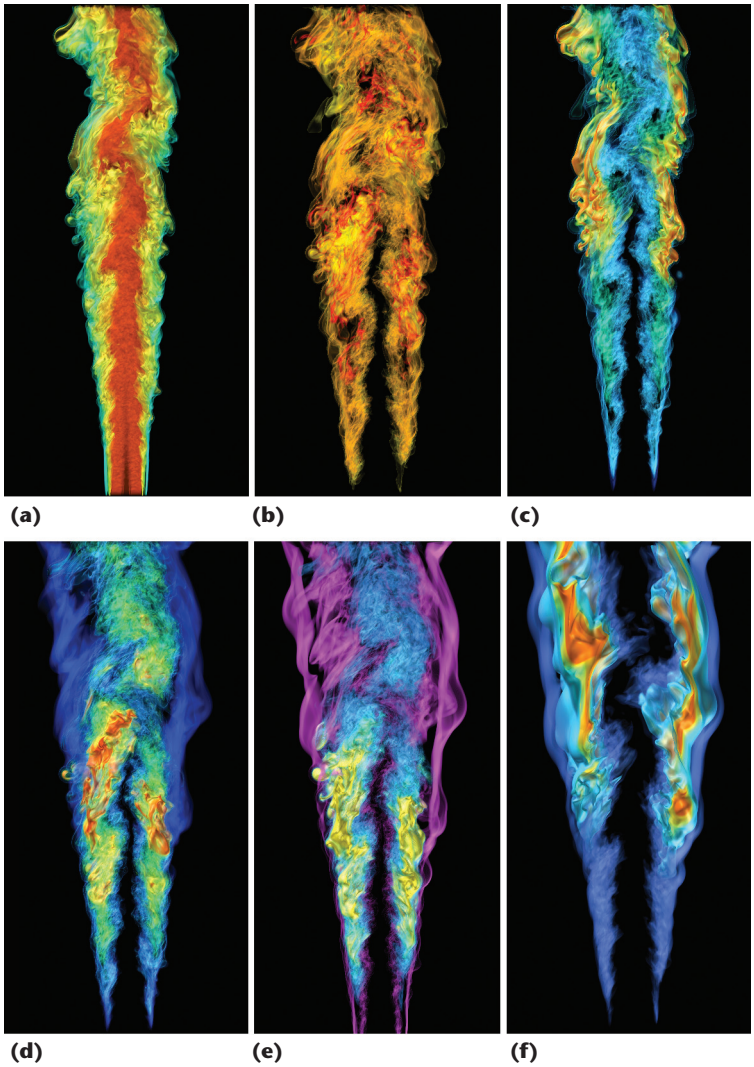


Figure 7. Volume rendering for six selected variables involved in the combustion process: (a) C_2H_4, (b) CH_2O, (c) CH_3, (d) H_2O_2, (e) HO_2, and (f) OH. All six images are generated under the combined-compositing mode in a single pass.

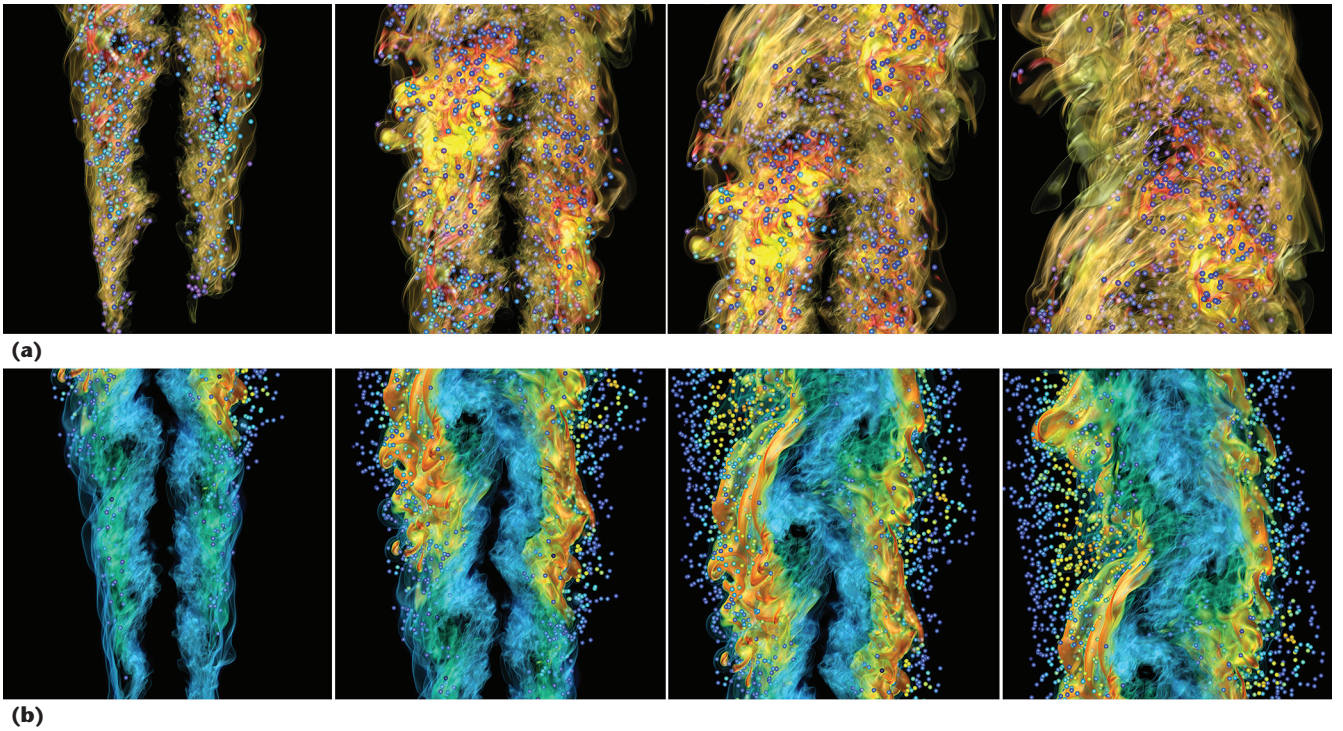


Figure 8. Detailed views of the volume and particle data mix rendering. (a) The volume variable is CH_2O; the particle variable is HO_2. (b) The volume variable is CH_3; the particle variable is OH. Our high-resolution in situ visualization lets the scientists examine details and monitor the simulation on the fly.

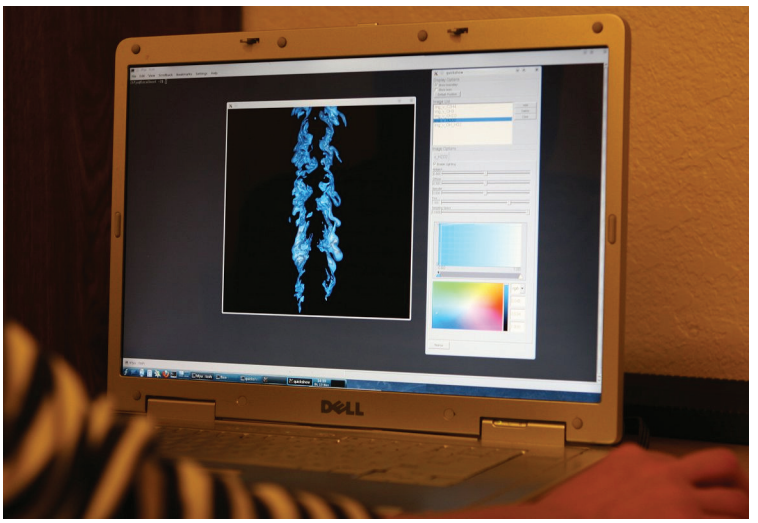


Figure 9. In situ visualization of a combustion simulation. A user in California remotely steers the visualization of a simulation running on 2,500 cores of Oak Ridge National Laboratory's Cray XT5 in Tennessee.

is 2.5 saved files per Kolmogorov timescale. This frequency ensures that system can recover Eulerian turbulent statistics from the restart files, given a sufficiently long time series.

However, this interval isn't necessarily sufficient to accurately depict the flow's instantaneous evolution. For example, certain small features might move rapidly. Ignition kernels upstream of the flame base are 0.05 to 0.1 mm in size, and the jet's center has a velocity of 204 meters per second. So, the kernels move approximately their own dimension every 0.245 μs. Thus, rendering the field more frequently than this—for example, four times per restart file—will smoothly capture these features' path. The in situ techniques enable this frequent rendering.

Before in situ visualization, Sandia's combustion scientists subsampled the data to reduce both data movement and computational requirements for visualization. The procedure involved many steps, multiple tools and platforms, and enormous I/O cost. Because of this cumbersome iteration, they created visualizations mainly for publication or presentation. With in situ visualization, the scientists can significantly reduce I/O and use visualization more conveniently for diagnostics to verify the simulation's correct configuration and operation.

In situ visualization's integration effort and computational cost make it feasible and practical for large-scale turbulent-combustion simulation. Beyond in situ visualization, however, a broader possibility is in situ processing, including data packing, feature extraction, and analysis.

In situ processing could let scientists study the full extent of the data their simulations generate and eventually steer simulation at the petascale level. For example, many scientists we've been working with are convinced that in situ feature extraction is feasible. They believe this because all

relevant data about the simulated field are readily available for analysis during the simulation.

The ability to steer a large-scale simulation lets scientists close the loop and respond to simulation results as they occur by interactively manipulating input parameters. As parameter changes become more instantaneous, the causal effects become more evident, thus helping scientists develop intuition, understanding, and insight into their modeling, algorithms, and data. We'll continue studying in situ processing for selected applications to understand this new method's impact on simulations, subsequent visualization tasks, and scientists' work processes.

Acknowledgments

Our research sponsors include the US Department of Energy (DOE), Office of Advanced Scientific Computing Research; the DOE Basic Energy Sciences Division of Chemical Sciences, Geosciences, and Biosciences; the DOE through the Scientific Discovery through Advanced Computing program, agreement DE-FC02-06ER25777; and the US National Science Foundation, grants OCI-0325934, OCI-0749217, CNS-0551727, CCF-0811422, CCF-0808896, OCI-0749227, OCI-0950008, and CCF-0938114. Sandia National Laboratories is a multiprogram laboratory that Sandia Corp., a Lockheed Martin Company, operates for the DOE under contract DEAC04-94-AL85000. We also acknowledge the computing resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which the DOE supports under contract DE-AC05-00OR22725.

References

1. K.-L. Ma, "In-Situ Visualization at Extreme Scale: Challenges and Opportunities," *IEEE Computer Graphics and Applications*, vol. 29, no. 6, 2009, pp. 14–19.
2. K.-L. Ma et al., "In-Situ Processing and Visualization for Ultrascale Simulations," *J. Physics: Conf. Series* (*Proc. Scientific Discovery through Advanced Computing Conf.*), vol. 78, 2007, article 012043, doi:10.1088/1742-6596/78/1/012043.
3. C. Ding and Y. He, "A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems," *Proc. 2001 ACM/IEEE Conf. Supercomputing* (SC 01), ACM Press, 2001, p. 50, doi:10.1145/582034.582084.
4. H. Yu, C. Wang, and K.-L. Ma, "Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing," *Proc. 2008 ACM/IEEE Conf. Supercomputing* (SC 08), IEEE Press, 2008, article 48, doi:10.1145/1413370.1413419.
5. U.D. Bordoloi and H.-W. Shen, "View Selection for Volume Rendering," *Proc. IEEE Visualization Conf.* (VIS 05), IEEE Press, 2005, pp. 487–494.

Hongfeng Yu *is a postdoctoral researcher at Sandia National Laboratories. His research interests include scientific visualization, parallel computing, and user interface design. Yu has a PhD in computer science from the University of California, Davis. Contact him at hyu@sandia.gov.*

Chaoli Wang *is an assistant professor of computer science at Michigan Technological University. His research interests include large-scale data analysis and visualization, high-performance computing, and user interfaces and interaction. Wang has a PhD in computer and information science from Ohio State University. He's a member of the IEEE. Contact him at chaoliw@mtu.edu.*

Ray W. Grout *is a postdoctoral researcher at Sandia National Laboratories. His research interests include combustion processes, high-performance numerical solvers, massively parallel computations, heterogeneous computing, and data analytics tools. Grout has a PhD in engineering from the University of Cambridge. Contact him at rwgrout@sandia.gov.*

Jacqueline H. Chen *is a distinguished member of technical staff at Sandia National Laboratories, an adjunct professor of chemical engineering at the University of Utah, and a director on the Combustion Institute Board of Directors. Her research interests include terascale simulations of turbulent combustion, time-varying visualization of terascale simulated data, combustion feature topology, and parallel feature detection and tracking algorithms for combustion. She's a member of the editorial advisory boards for* Combustion and Flame, Progress in Energy and Combustion, Proceedings of the Combustion Institute, *and* Computational Science and Discovery. *Chen has a PhD in mechanical engineering from Stanford University. Contact her at jhchen@sandia.gov.*

Kwan-Liu Ma *is a professor of computer science and the chair of the Graduate Group in Computer Science at the University of California, Davis. His research interests include visualization, high-performance computing, and user interface design. Ma directs the university's Visualization and Interface Design Innovation group and the Scientific Discovery through Advanced Computing Institute for Ultra-Scale Visualization. He's an associate editor in chief for* IEEE Computer Graphics & Applications, IEEE Transactions on Visualization and Computer Graphics, *and* Computational Science and Discovery. *Ma has a PhD in computer science from the University of Utah. Contact him at ma@cs.ucdavis.edu.*