Stitch It Up: Using Progressive Data Storage to Scale Science

Jay Lofstead
Sandia National Labs
gflofst@sandia.gov

John Mitchell
Sandia National Labs
jamitch@sandia.gov

Enze Chen
University of California, Berkeley
chenze@berkeley.edu

***Abstract*—Generally, scientific simulations load the entire simulation domain into memory because most, if not all, of the data changes with each time step. This has driven application structures that have, in turn, affected the design of popular IO libraries, such as HDF-5, ADIOS, and NetCDF. This assumption makes sense for many cases, but there is also a significant collection of simulations where this approach results in vast swaths of unchanged data written each time step.**

This paper explores a new IO approach that is capable of stitching together a coherent global view of the total simulation space at any given time. This benefit is achieved with no performance penalty compared to running with the full data set in memory, at a radically smaller process requirement, and results in radical data reduction with no fidelity loss. Additionally, the structures employed enable online simulation monitoring.

***Index Terms*—storage, io, database, python, analytics, spparks, kinetic monte carlo**

I. Introduction

Welding and additive manufacturing (AM) models [1] have been successfully incorporated into the *spparks* [2], [3] kinetic Monte Carlo framework. For the purpose of engineering design and analysis, material microstructures are predicted by simulating the process of melt, fusion and solidification. In these models, a heat source, typically a laser, creates a very localized melt pool and surrounding region called the heat affected zone (HAZ); outside of the HAZ, the temperature is below the threshold for grain growth and evolution. In the case of welding, the laser heat source moves along a joint melting and fusing material from both sides of the joint which subsequently solidifies. At any particular instance during the process, models need only consider the localized HAZ around the laser location. Material in the remaining areas outside of the HAZ remains unchanged. This is the key concept and genesis of motivation for *Stitch-IO*.

In contrast, the traditional approach creates the entire simulation domain, over which the laser travels, in-core. Computation is attempted across the entire domain, but only causes data changes in the HAZ as the laser moves across the domain. For IO, the entire domain is written for each output. From a computational standpoint, this is wasteful since most of the data at any particular time step on the simulation domain is unchanged and trivially predictably so. Also, because of the local nature of the process described above, compute resources are wasted on most of the domain excepting the time the laser moves into that process's area.

Using the traditional computational and IO approach, practical engineering simulations of welding and additive manufacturing are impossible. Length scale differences between grains and manufactured parts necessitate use of enormous computational resources. Metallic microstructures are typically measured and simulated in length units of μm, while typical welded structures and AM components are measured in cm and larger. *Spparks* uses a regular grid of lattice sites to simulate and model grain growth and evolution where the ideal distance between sites is $1\,\mu\mathrm{m}$. Each site is associated with a grain which is typically represented by a 32-bit integer. Consider a small weld simulation $5\,\mathrm{cm}$ long ($50\,000\,\mu\mathrm{m}$) with a plate thickness of $0.07\,\mathrm{cm}$ ($700\,\mu\mathrm{m}$) and $0.4\,\mathrm{cm}$ ($4000\,\mu\mathrm{m}$) HAZ width. This translates to a computational lattice having $50000 \times 700 \times 4000 = 140$ billion sites; the computation is very small in size relative to typical component parts, yet would require the very largest computers and would likely be very difficult if not impossible to run. In another example, Lockheed Martin recently used a metal 3-D printer to create a titanium cap for a fuel tank. This part is $1.16\,\mathrm{m}$ in diameter [4]. Relevant simulations to predict microstructures for this build are currently impossible. Stitch-IO and *spparks* are two steps towards making that possible. The extreme data volumes for realistic simulations of microstructure during AM demands rethinking the data storage further—even when considering just $0.1\,\%$ of the volume is actual material. Fitting the entire model into memory cannot be done for any machine that exists today.

This fuel tank cap is far from the only example. A spray system Sandia is investigating wants to understand deposition and layering of material over a seemingly small space, yet because of length scale considerations, practical computations are limited because of the traditional computational and IO approach. Building such a model at the scales required similarly will not fit into memory anywhere.

A third example class can be described in finite element models. Consider an impact event creating a shock wave that propagates through a material. As the wave moves, state of the material is affected in the region of the shock front but nothing ahead or behind the wave (outside a given window) changes.

All of these examples share an extreme total simulation domain size along with "productive" compute being localized to just a small portion of that domain on any given time

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00016

step. Trying to run such a model using traditional techniques and using traditional IO libraries is extremely wasteful, if not impossible. Given the tiny compute active area for the simulation, being able to run these extreme-scale simulations on a laptop or small cluster seems like a reasonable goal. We do not need the extreme compute capability nor the world class storage capacity. Herein we address both of these goals.

To address this compute waste, *spparks* has been adapted so that only the portion of the domain affected by the current computation is loaded into memory for any step. When the affected region reaches a boundary, it can exit to reset for the next incremental area. For output, it either generates an image suitable for creating a movie or a text file representing the data in that simulation area. The image or text file approaches are the best available options due to a lack of a suitable IO library to handle only writing a portion of the simulation domain for any given time step and being able to automatically stitch together a coherent view of a given region at any time. For other kinds of analyses, neither option provides an ideal solution.

Stitch-IO provides a new way to think about IO. Instead of writing the whole domain each time, just the portion that changes can be written in a lossless format that is capable of reconstituting a complete simulation domain on request. The entire domain size is never requested nor directly known. Time is a fundamental construct that addresses how to reconstruct a coherent domain view.

Various *spparks*-based applications and the class of finite element codes benefit from this approach. This paper describes the design approach and quantifies the overheads and costs associated with the choices. The decisions made have also opened up new opportunities for running these simulations. For example, it is possible to run multiple simulation instances, each in a different part of the simulation domain, all writing to the same file at the same time. Simultaneously, an analysis application can probe the file to reveal the progress written. In this case, we can demonstrate a multi-writer, multi-reader application.

Our contributions are as follows:

- An approach for ignoring total simulation domain size while storing data.
- An approach for direct data analysis and visualization using typical analysis tools while the simulation is running.
- An approach for recreating a coherent simulation domain view at any given time and sub-region specified.

For this paper, the bottom line is the following: *We have enabled many new simulations previously impossible by radically reducing required compute resources; we provide a lossless data compression approach all without an increase in wall clock time compared to a full, in-memory model.*

The rest of the paper is structured as follows. First, a walkthrough of the design constraints and decisions that have led to Stitch-IO are presented in Section II. An experimental evaluation follows in Section III. A review of related work is in Section IV. Section V discusses conclusions we can draw and future work plans.

II. Design

The target application classes for Stitch-IO all benefit by performing compute in small areas at a time over multiple runs or initializations. Given this goal, the focus is on enabling writing the smallest reasonable part of a simulation domain at any given time and then being able to assemble a coherent view of an arbitrary part of the entire simulation domain for analysis. This section first presents the idea of *stitching* and then discusses design constraints and decisions.

A. How Stitching Works

The idea with *stitching* is to reformulate the compute problem from a full domain exploration divided over parallel processes into a series of compute volumes (*cv*s). Then, the simulation can work on each compute volume at small scale progressively building up the full simulation results. While this may seem excessively slow, recall that breaking down the problem this way enables many different simulations not previously possible. Further, it does it in a way that adds little to no additional wall clock (charged) compute time compared to if the work was done at full scale with the full domain in memory. And since it runs on a small fraction of the processes, the computational cost is tiny and the data storage is a small fraction of what it would be were the full domain written for each output.

Consider Figure 1. If the simulation was run at full scale, the entire domain would be decomposed and distributed across many processes; each process taking one blue block of domain which is shown divided by the blue lines (the whole domain). The teardrop shape in the lower left orange box represents a weld pool melt as part of a welding simulation. During a time step, the weld simulation is highly localized. All of the action during a time step takes place in the vicinity of the teardrop (the HAZ). The orange box containing the teardrop represents the compute volume for a single *spparks* run. That means that as far as *spparks* knows, it is running the problem just over the region defined by the orange box. Once *spparks* completes working across the orange box, it exits for the next phase.

The next phase is to generate an additional domain area, such as the purple block. *Spparks* initializes this data for the simulation by writing the initial state and exiting. It needs the edge of the previous area in order to properly initialize the new *cv* seamlessly with the previous *cv*. Then *spparks* restarts again, initializing and reading from the purple block so that it can move the teardrop area across the purple area. Next comes the green block using the same event sequence. This continues until the melt pool hits an edge of the full simulation domain where it wraps around in a serpentine pattern or finishes a layer and moves vertically.

Figure 2 depicts grain evolution during simulation of the AM weld model with *stitching*. Much like the progression of a physical AM process, only the first *cv* has to be initialized at the start of the simulation as the melt pool only affects sites along the bottom edge; the rest of the domain is assumed to be unaffected and therefore does not have to be stored in the Stitch-IO file (represented in black). As the melt

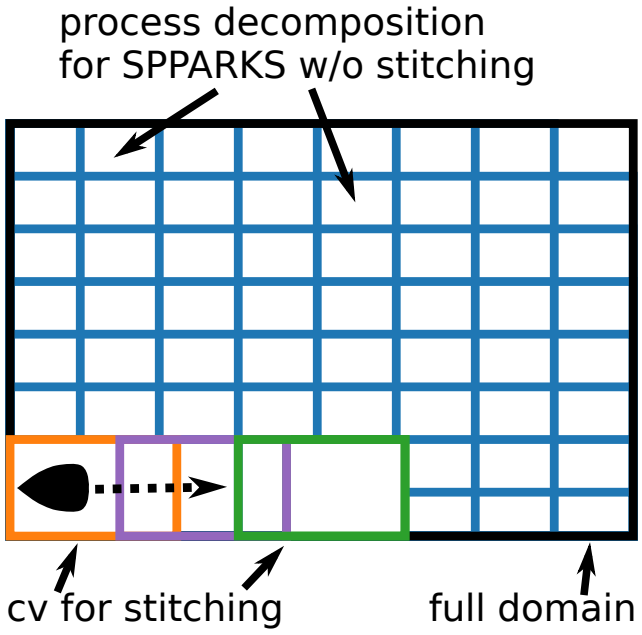


Fig. 1: Breaking down the domain for *stitching*.

pool progresses, it moves in a serpentine pattern, reversing directions with each transverse increment, evident by the grain inclinations along the raster direction. The *cv*s are generated one at a time, and only the current one is used for computation and IO—what is shown is a result of *stitching* together the previously computed *cv*s. Note that all sites outside of the current HAZ are constant at later time steps, which is reflected by the color of the grains in the figure not changing. In the end, Stitch-IO is able to recover the full simulation domain by *stitching* together the individual *cv*s in a manner that is commensurate with the progression of the AM build.

Not surprisingly, the batch script for running a *stitching* application is a bit more complex to account for the initialize-process-shift cycle. The upside is that the script can likely also be used on a laptop running Linux with little to no modification (excepting job control commands and directory paths). The code runs the same in both environments. During development of Stitch-IO, testing was performed on laptops, desktops, and a capacity cluster at Sandia. In all cases, everything ran correctly.

B. Design Constraints

There are several design features that are a strong departure from existing IO libraries. As mentioned in the Introduction, Stitch-IO seeks to scale arbitrarily while still being able to address extreme scale problems. Some design constraints address this directly while others address scientist productivity.

1) Arbitrary Blocks: Given the small subset of the simulation domain used for each time step, some storage mechanism independent of the entire domain size, but with arbitrary bounds, is required. We chose to store blocks annotated with the bounds within the total simulation domain it represents along with a timestamp to aid reconstructing the total simulation domain. Further, the entire simulation domain is never defined. Instead, it is inferred based on what is written. It is possible to query what the total bounds written have been so far, but that is learned indirectly through the extremes for each block written.

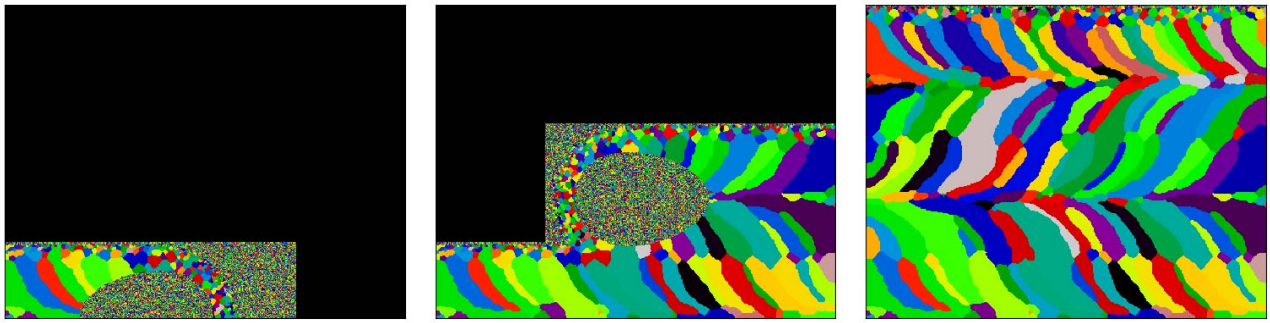
2) Floating Point Time Values: While typical IO libraries focus on integer-based time steps, this is not natural for a simulation, which are run using time scales frequently corresponding to the speed of physics phenomena. Using an integer-based time step is an abstraction that counts the time increments rather than representing the actual time elapsed in the simulation physics. Instead of maintaining this abstraction, we enable floating point time to the user, but use an integer index into that list of times internally. We address the IEEE floating point inexact matching problem through a tolerance mechanism to match a selected time against anything that exists already. We use both an absolute tolerance and a relative tolerance value. The combination of these offers a flexible way to address round-off errors and is user-configurable on a run-by-run basis.

3) Automatic Block Stitching*:* By moving away from deploying a simulation such that the entire simulation domain is in memory at the same time, we gain the ability to scale from a laptop up to a cluster to a supercomputer, depending on the storage availability and how long the scientist is willing to wait bounded by the problem size limitations. The welding example demonstrates the relatively small grain count affected by any time step. There is a strict upper bound on how far this can be decomposed before the efficiency gains of parallelism are lost due to excessive communications. The expectation is around 1000 processes, which is about the upper limit for a relatively efficient computation no matter the entire simulation domain size. Other models may have different natural decomposition limits. For the problem classes being addressed that have not been possible or feasible before, this small size advantage to address extreme-scale problems opens technology choices others have abandoned due to overheads and bottlenecks at extreme scale.

Moving to a block-based IO setup is crucial to handle large simulation domains without tremendous wasted space. For any given read operation, pieces from many blocks are assembled into a whole such that the newest values no newer than the requested time within the requested region are returned.

Continuity is handled by running from the starting time until the next time that meets or exceeds the maximum specified. Discontinuity in data is handled through the "no value present" values. This intentional design choice allows arbitrary data selection even where there are oddly shaped gaps. The application understands these values and properly initializes them prior to performing the calculations.

4) Relational Database Functionality: Relational databases with proper indices and SQL queries have long been a powerful way to select data. By embracing this technology in an area not typically enamored of the overheads a database engine requires is a risk but offers functionality benefits with little additional effort. Given the small scale, the benefits outweigh the potential penalties, as explored in the evaluation



(a) $t = 26\,\mathrm{MCS}$ (b) $t = 98\,\mathrm{MCS}$ (c) $t = 198\,\mathrm{MCS}$

Fig. 2: Example of grain evolution produced by the *spparks* AM weld model with *stitching*.

below. Transactions offer consistency guarantees enjoyed for enterprise workloads and can be expanded to support parallel clients trivially [5], [6].

5) Native Python Support: Crucial to the way that scientists work today, Python has become the analysis tool of choice for many and is growing rapidly. We embrace this by offering a fully-functional Python interface in addition to the native C library interface used by the simulations. Additionally, by the nature of database transactions, the Python interface can be used during the simulation run without causing the simulation to fail due to a busy file or to return incorrect or incomplete information.

C. Design Details

The various design constraints described above all address how to make a system with the right flexibility and features to address the problem domains motivating this work. The actual design and implementation depart from traditional IO design wisdom in many ways and are discussed below, followed by a detailed description of Stitch-IO itself.

1) Why a database?: Stitch-IO's design and implementation are built around a single decision: Use an embedded relational database engine to handle concurrency, simplify searching for blocks for *stitching*, and offer resilience. In our case, the initial choice is SQLite [7]. This public domain relational database is used in millions of embedded applications worldwide and has low system requirements and constraints. It offers a server-less mode that is ideally suited for an application like Stitch-IO. The full ACID transactions support and journaling offer both consistency and resiliency without having to write any additional code. Further, the ability to place an index on relevant columns needed for block selection when *stitching* allows efficient selection and ordering to be performed, and avoids the typical list-scan approach native to traditional IO libraries. At scale, the traditional scanning approach may not even be possible due to the list size. An indexed database table on storage can still be accessed efficiently.

The current approach for *stitching* is not optimal. First, the return buffer is initialized with the "no value present" value. Next, blocks are returned ordered by timestamp and then in X, Y, and Z copying any element that falls within the bounds of the requested dimensions into the return buffer based on a joint array traversal of both the return buffer and the currently selected intersecting block. For the portion of each block within the requested region, Stitch-IO simply copies the values into the element location. Since these blocks are ordered by time with the newest last, it ensures that the latest value for any location ends up in the return buffer. More efficient block intersection queries and potentially moving from newest to oldest using some bookkeeping about what values have been updated may prove more efficient. Exploring more efficient approaches is left to future work.

While a disjoint model with data stored in a separate location from the metadata was initially considered (similar to what EMPRESS [8] can do), SQLite adequately supports blobs enabling a simpler overall design. For some very large-scale applications, where total data in the Stitch-IO format might be $O(1\,\mathrm{PB})$, splitting the data apart from the metadata may be a better approach. Exploring this is left partially for future work and partially has already been explored with EMPRESS.

2) C and Python interface: C/C++ and Fortran interfaces for IO libraries have been the standard for over 30 years; however, the rapid development of rich data analytics tools integrated with Python and the Fortran-like array manipulation possible with NumPy arrays have led a growing number of scientists to demand good Python support for data access and analysis. The tool variety, flexibility, and quality has in some ways made existing heavyweight analysis tools obsolete. Fixed user interfaces to access data are no longer sufficient. Standardized data access approaches that integrate with existing tools, such as Pandas DataFrames, along with the native NumPy and SciPy functionality have led to scientists being able to quickly get analysis tasks coded and completed compared to traditional approaches.

These trends show no sign of slowing and demand that any new IO and data format work seamlessly in this environment or be subjected to obscurity. To ensure that Stitch-IO works well in this environment, the APIs were developed in both Python and C simultaneously with testing harnesses written in both languages to ensure that all functionality worked properly and naturally in both languages.

As a side benefit, Python has native SQLite support offering a way to inspect the "raw" Stitch-IO file and potentially implement alternative data access and analysis routines not offered by the stock API. This additional flexibility and functionality is sorely missing on existing IO libraries. This embrace of standard tools at even a low level within a library signals a shift in thinking from "How do I build the most efficient tool at a cost of complexity and flexibility?" to "How do I build a tool that works well for the user and offers a convenient way for them to go beyond what I provide?" Any new data storage and access approach should consider this shift to best serve the applications and scientists we are trying to support.

3) Both Parallel and Serial: Since Python is typically not run with MPI or some other parallelization framework, having a serial library version is crucial. It is important that this serial version does not, in any way, affect anything done with a parallel version of the library or data storage and vice versa. To address this, the primary Python test cases are serial with some parallel cases available in a second set. The C test cases are primarily parallel with some serial tests as well. While this does change the interface to the `open` command to include the MPI communicator for parallel builds, that is the sole API difference between the two. Under the covers, additional changes that support better concurrency controls for the parallel build are used.

4) API Complexity: The basic *spparks* examples all use 32-bit integers as their standard element values; however, this is not sufficient for all cases and additional API variations are available for 64-bit floating point and integer values. Further, the element value may also be a vector. This is specified through the use of the length descriptor. Multiple fields, potentially of different types, can be created, written, and read all from the same file. The API is only a little more than basic `open`, `write`, `read`, `list_vars`, and `close`.

D. Design

Stitch-IO consists of the Python module interface, a C interface and implementation, and the underlying data storage mechanism, currently SQLite. A typical call sequence in Python is illustrated in Algorithm 1. The basic flow is an `open`, `create` a field, and set basic parameters. After this, we `query` the field to be simulated and `read` to obtain both existing data and "no value present" values for untouched parts of the simulation domain. The actual computation is inserted next. Finally, we `write` the resulting blocks as the simulation time progresses. For *spparks*, there is an outer loop in the job script moving the region and then an inner loop inside *spparks* for moving across the computation volume.

Some of these lines may not be completely obvious. `create_field` (line 2) takes a type from an enum (1 = 32-bit integer), the per-site length (vector length for the location), and the "no value present" value (-1). One of Stitch-IO's features is the use of a floating point time value to better match the simulation time. The `abs_tol` is the absolute tolerance for matching time while the `rel_tol` is the relative tolerance. These are combined into the comparison

Algorithm 1 Stitch-IO Python Call Sequence

```
 1: fid = stitch.open ('filename.st')
 2: field = stitch.create_field (fid, 'spin', 1, 1, -1)
 3: abs_tol = 1.0e-9
 4: rel_tol = 1.0e-15
 5: nvp = -1
 6: t1 = 0.0
 7: b1 = numpy.fromiter ([0, 6, 0, 2, 0, 2], ...)
 8: s1 = numpy.ones (b1)
 9: stitch.set_parameters (fid, abs_tol, rel_tol, nvp)
10: ...
11: field = stitch.query_field (fid, 'spin')
12: data, new_t = stitch.read_block (fid, field, t1, b1)
13: ...
14: new_t = stitch.write_block (fid, field, t1, b1, s1)
15: ...
16: stitch.close (fid)
```

to determine if the time matches or not. The `req_time` is the time value provided by the user and `time` is a time value created previously. A time is considered matching if the following condition is met:

$$|\text{time} - \text{req\_time}| < [\text{abs\_tol} + (\text{req\_time} \times \text{rel\_tol})]$$

Line 6 is the time value. Line 7 creates a bounding box from $(0, 0, 0)$ to $(6, 2, 2)$. Line 8 is a buffer containing the values to write. Lines 12 and 14 have an interesting return value: `new_t`. This is a "new time" flag. For writing, this flag signals if the time written existed previously or not. This is useful for sanity checking a simulation run.

III. Experiment Design

Experiments are divided into three sets to better evaluate the comparison points. First we evaluate strong and weak scaling for a full domain setup showing the Stitch-IO performance against the native *spparks* IO techniques including no output, text file output, image file output, and Stitch-IO file output. Next, we evaluate *stitching* where we run *spparks* with both Stitch-IO and text file IO to gauge performance. Third, we run a set of *stitching* tests just using a testing harness and the Stitch-IO library compared against a full domain HDF5 example to investigate the resulting output sizes and the potential for using an alternative like HDF5 instead of Stitch-IO.

For the strong scaling experiments, we chose a total simulation domain of $1200 \times 1200 \times 88$ for an initial test. We can run this on a single process or as many as 512. This is empirically about the upper limit for a reasonable communication vs. computation overhead trade-off. For weak scaling, we set the per process data size at 100,000 sites with a single field. We run at least 5 experiments for each configuration.

The evaluation is performed on the Chama capacity cluster at Sandia National Laboratories. It consists of 1232 nodes each with two 2.6 GHz Intel Sandy-Bridge CPUs with 8 cores/socket. Each node has 64 GB of DDR3 RAM and connects with a 4X QDR InfiniBand network configured in

a fat tree. The file system is the site shared GPFS offering 4.6 PB of storage. There are also 8 login nodes each with the same hardware as the compute nodes. All nodes run RHEL 7.

A. *Full Domain* spparks

A performance comparison between using the Stitch-IO library and the alternative IO techniques, including no output, are performed. Strong and weak scaling results are presented in Figure 3 and Figure 4 respectively. Because we were not *stitching* (described below) and only measuring Stitch-IO performance, the AM weld application was not used; both the strong and weak scaling studies were conducted using the *spparks* Potts model. The comparisons are with single text IO (all processes collect to one that writes to a single file), distributed text IO (each process writes to its own file), and image IO (all processes collect to one that generates an image writing the resulting JPG image to storage). The text IO is very inefficient mimicking the H5dump output. This was the default output format created by the *spparks* developers because it was easy to edit when working on different physics models. Image IO is a more typical output except when numerical values are necessary.

1) Strong Scaling: Simulation domain sizes were chosen to be on the order of 100 million total lattice sites so that performance evaluations could be completed reasonably quickly for a typical-sized problem. This represents a simple welding or additive manufacturing scenario with a single layer. This is large enough to test the physics, but small enough to run in reasonable time. This led to dimensions of $1200 \times 1200 \times 88$ sites. As seen in Figure 3, we observed that Stitch-IO performance was comparable to other forms of IO and had nearly ideal scaling behavior up to 256 processors. For 256 processors and beyond, we observed degradation in scaling behavior for Stitch-IO that was reflected, to varying degrees, in the other IO methods as well. We speculate that the worsening performance and higher variability at 512 processors is due to file system contention, although a deeper analysis is required for verification. However, in all cases, we found Stitch-IO to perform between 2 to 10 times faster than *spparks* single text IO and achieve much better scaling; note that single file text IO has burdensome communication issues because it requires each processor to send its chunk of data to a single output processor; this is not the case for distributed text IO which is faster and scales better. The resulting Stitch-IO file size was 1.4 GB on average, which is almost 4 times smaller than the average size of the text files (5.5 GB). Thus the Stitch-IO file is able to store output data more compactly than the text file; furthermore, Stitch-IO file sizes did not vary by more than 1 MB as the number of processors was scaled up.

The strong scaling results show that Stitch-IO still holds with the pack. Note that the natural scalability limit for the *spparks* code is revealed between 256 and 512 processes.

2) Weak Scaling: A weak scaling study was also conducted; in this case, the number of sites per process was kept around 5×10^5 sites/process. This led to the domain dimensions

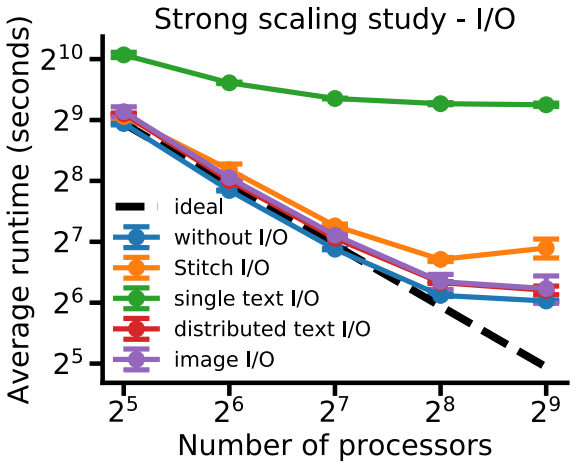


Fig. 3: Application strong scaling

shown in Table I. Figure 4 shows results for the same IO formats. Again, Stitch-IO remains competitive with other formats up to 512 processors, and still runs 2 to 10 times faster than *spparks* IO with a single text file. Once again, the Stitch-IO file is 3 to 4 times smaller than the text file on average, with the ratio increasing for more processors. All of the curves deviate from ideal behavior, which suggests that the increased parallel overhead is not exclusively a Stitch-IO issue. This result shows that it is feasible to use Stitch-IO as an IO tool on larger parallel computing clusters, though we emphasize far fewer than 512 processors are required if *stitching*.

The weak scaling results show that the Stitch-IO library has similar performance to the other reasonably performant IO techniques. The single text file approach is a typical N-1 gather-write pattern.

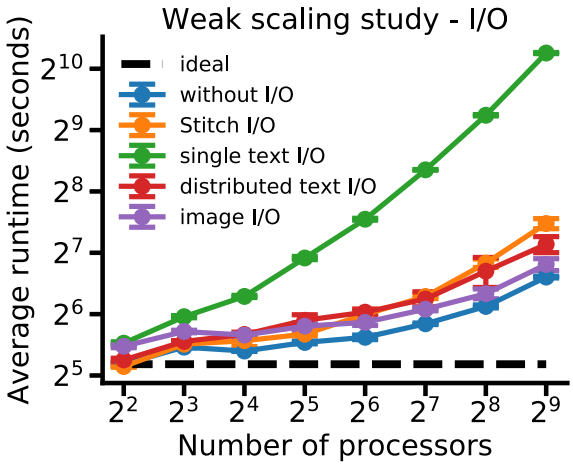


Fig. 4: Application weak scaling

B. *Stitching* spparks

For performance measurements, the domain size was progressively doubled as shown in Table II. When running *spparks* without *stitching*, a weak scaling type approach was used: The number of processors was doubled while attempting to keep the number of sites per process at approximately $7031\,\text{sites/process}$; without *stitching*, the required number of processors and file sizes increase rapidly (see Figure 6). The domain sizes we chose were limited by the traditional approach and its computability on a reasonable number of processors. When simulating the same domains with *stitching*, the number of processors was kept *constant* at 16 (approx. $6230\,\text{sites/process}$) on the individual *cv*, which is accessible by desktop machines. Output was written every $2\,\text{MCS}$ (Monte Carlo Sweeps) to a Stitch-IO file and distributed text files for *stitching* and no *stitching* tests, respectively.

First we compare *spparks* performance with and without *stitching* for the AM weld model. There were approximately $7031\,\text{sites/process}$ in the simulations without *stitching* and the same area was used for the *stitching* simulations on just 16 processors (approximately $6230\,\text{sites/process}$). Each simulation was run for full coverage and output was produced every $2\,\text{MCS}$. The numbers next to the data points indicate the number of processes used for that computation. The results are shown in Figure 5. While *spparks* without *stitching* generally ran faster for all cases, it used many times more processors than *stitching* (up to 2^6 times more). *Stitching* has much improved scaling and runs at about the same performance or sometimes a bit slower than *spparks*, despite using only 16 processors. Figure 6 shows a comparison of the file sizes for Stitch-IO compared against the single text file output. The size of the output file(s) are plotted against the domain size for both cases. The Stitch-IO file is 2 to 75 times smaller than the corresponding set of text files (note the log scale on the y-axis), which is a dramatic reduction in storage requirements without any fidelity loss due to aggressive compression techniques typically employed to achieve such results.

C. *File Size Comparison*

Comparing the resulting file size against alternative options is important for two reasons. First, with lossless data reduction being a feature for Stitch-IO, it is important to show the resulting file size reduction to give an indication of how well the data reduction works in practice. Below are several test cases to show how well this data reduction is accomplished.

Second, with the design decision to use a database rather than a compact file format for Stitch-IO, evaluating how the overheads involved in using the database compared against other techniques is important. Naively, Stitch-IO has a generous number of indices to make querying faster. In other work we have performed [8], using indices on about half of the database columns ends up using half of the storage space as the database scales. More careful planning for which columns having an index can improve performance will shrink this overhead. These optimizations are left for future work.

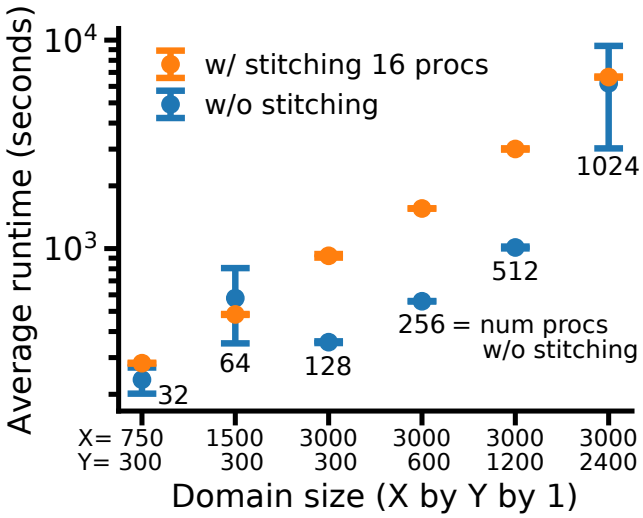


Fig. 5: Average runtime (seconds) for simulations with *stitching* (orange) and without *stitching* (blue). Error bars represent 1 standard deviation. Simulations with *stitching* were performed using 16 processors for all domain sizes while those without *stitching* were scaled from 32 up to 1024 processors.

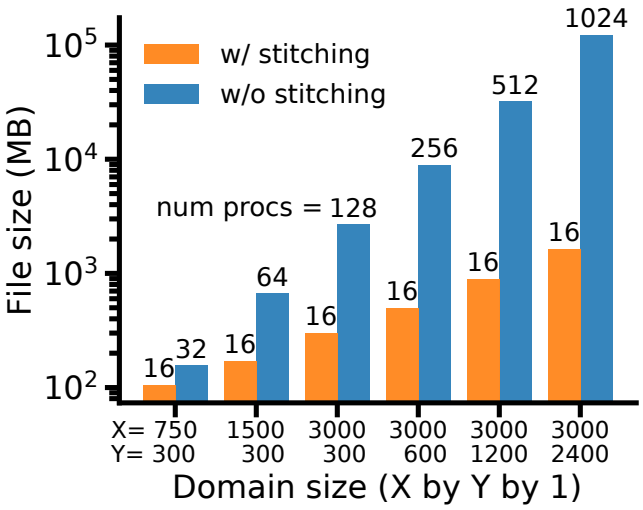


Fig. 6: File sizes (MB) for simulations with *stitching* (orange) and without *stitching* (blue), with the data stored in Stitch and text files, respectively. The number of processors in each case match that for Figure 5.

Table I presents a set of weak scaling parameters and results, where the number of sites per process was kept around $5 \times 10^5\,\text{sites/process}$. The Stitch-IO file size was 3 to 4 times smaller than the size of the corresponding text file, the only other viable option at scale. This would be the worse case setup since we would use the same number of processes for the stitching case as we do for the full model in other cases. Note that while HDF5 is a clear winner up through an $860 \times 860 \times 88$ model, stitching is already winning in terms of size beyond that. Given an approximate size of $1000\,\text{sites/in}$, modeling anything of non-trivial size is better off using Stitch-

IO for the data storage. Other factors of note are that the representative HDF5 format is a single dataset variable per time step while for the Stitch-IO case, it is one database entry per process per time step. In spite of this disparity, the per-variable overheads in HDF5 eventually overwhelm the decomposed Stitch-IO approach. Were an HDF5 file to be used in a similar way, even if the overhead ratio does not change, it would be overwhelmingly large. Understanding the increasing overheads for variables at scale in an HDF5 file is not within scope for this paper. Stitch-IO maintains strictly the overheads associated with rows in a database table linearly.

Additional tests using various data compression techniques with HDF5 are also beyond scope for this paper. Given the extreme lossless data reduction Stitch-IO achieves, no HDF5 integrated approach can approach this lossless "compression" ratio. Performing these comparisons is left for future work.

Table II shows a comparison for doing a traditional-style full domain model. Num procs is the number of processes for simulations without *stitching*. The Text file size is the sum of all distributed text files, and it is considerably larger than the size of the corresponding single Stitch-IO file. When running *spparks* without *stitching*, a weak scaling type approach was used: The number of processors was doubled while attempting to keep the number of sites per process at approximately $7031\,\mathrm{sites/process}$; without *stitching*, the required number of processors and file sizes increase rapidly (see Figure 6). In this case, the Stitch-IO file size is radically smaller than the existing text output file size. HDF5 has not been integrated into *spparks* preventing a direct comparison; however, as a point of reference, in Table I, the Stitch-IO file has (Num procs $\times$ MCS iter outputs) chunks written to the file. The HDF5 file has one variable entry per MCS iter output. In the most extreme case, there are 540 variables in the HDF5 file. If it were adjusted to accurately store the various chunks similar to how Stitch IO works, the total size will explode based on the overheads these tests demonstrate.

D. Discussion

While Stitch-IO is not always the fastest IO technique, it is comparable to the other techniques and far better than the single text file approach in all cases. The real runtime advantages for Stitch-IO lie in the stitching examples. Note that for the process count, when compared against HDF5 it has no worse performance and uses a small fraction of the space without resorting to lossy compression.

SQLite has one disadvantage we work around. When a process takes a write lock, the entire database is locked using POSIX file system locks. This is a deep part of the SQLite design and therefore not easily changed. To address concurrency, we serialize all access with token passing. All of the experiments performed used this technique to address concurrency. API compatible approaches, such as Berkeley-DB [9], [10], that only lock the table and use a file for locks, are alternatives and are being evaluated. A deeper exploration of the trade-offs between these two databases for applications like Stitch-IO are underway.

The real functional advantages for Stitch-IO are still being explored. The daunting full domain sizes and/or the amount of effectively idle compute to run these kinds of models has prevented them from being expressed for computers. The stitching approach and the Stitch-IO library have reframed the problem approach enabling many cases previously thought impractical or impossible. The initial cases have been compelling for the application scientists leading them to be advocates to their peers with similarly intractable problems. Given the first successful demonstrations for welding and a 2-D additive manufacturing setup, multiple additional groups within Sandia have clamored for access to the code so that they can finally perform simulations for problems they feared would not be possible for the foreseeable future. It is also being promoted to the DOE complex as a whole as a potential solution to intractably large simulation requirements.

IV. Related Work

The related work is divided into sections on data management, computational models, and visualization.

A. Data Management

Traditional IO libraries, such as HDF5 [11], NetCDF [12], ADIOS [13], and PnetCDF [14] all work under the assumption of a fixed simulation domain size. While this works for existing applications, there are application classes where this does not work well. For example, the introduction offers new simulation classes currently not possible.

Structurally, ADIOS does not enforce the domain bounds, but it does assume that the whole domain is present for reading. Should data be missing, it would be deemed a corrupted file. Stitch-IO makes no data availability assumptions and instead uses a "no data present" value much like NetCDF offers to represent requested areas that do not have a value yet. Unlike NetCDF, instead of storing the "no value present" value at each location, data is simply not stored.

Stitch-IO offers unprecedented lossless data compression for the application classes targeted. While many libraries (HDF5, ADIOS) support data compression plug-ins, these are mathematical compression rather than just eliminating all of the data that does not change over time. Lossy compression methods achieve extreme data reductions, but at a fidelity loss.

Other difference-based compression approaches [15], [16] have been evaluated, but they require some way to either annotate that the data changed or something to compare against to determine if the value differs. The Stitch-IO approach eliminates both the annotation and comparison by only having a region slightly larger than the area affected computationally in memory at a time guaranteeing that no unaffected data will either be stored or compared against for data storage. This is also the region written for each output which avoids writing any unaffected portions of the simulation domain.

Systems like BORG [17] and EDO [18] try to rearrange data to optimize placement. Stitch-IO's focus is on not writing redundant data in the first place.

TABLE I: Weak scaling parameters and comparison of file sizes.

Num procs	MCS iter outputs	Domain size	Num sites	Stitch-IO file size	Text file size	HDF-5 file size
4	15	150 × 150 × 88	1980000	23 MB	59 MB	1.3 MB
8	22	216 × 216 × 88	4105728	47 MB	137 MB	4.0 MB
16	30	300 × 300 × 88	7920000	91 MB	277 MB	11 MB
32	65	426 × 426 × 88	15969888	186 MB	608 MB	46 MB
64	90	600 × 600 × 88	31680000	363 MB	1.26 GB	124 MB
128	160	860 × 860 × 88	65084800	753 MB	2.67 GB	391 MB
256	300	1200 × 1200 × 88	126720000	1.42 GB	5.53 GB	1.7 GB
512	540	1700 × 1700 × 88	254320000	2.91 GB	12.1 GB	5.9 GB

TABLE II: Parameters used in the *stitching* performance study.

Num procs	Domain size	Num sites	Stitch-IO file size	Text file size
32	750 × 300 × 1	225000	105 MB	156 MB
64	1500 × 300 × 1	450000	170 MB	663 MB
128	3000 × 300 × 1	900000	300 MB	2.68 GB
256	3000 × 600 × 1	1800000	497 MB	8.92 GB
512	3000 × 1200 × 1	3600000	890 MB	31.8 GB
1024	3000 × 2400 × 1	7200000	1.64 GB	122 GB

PLFS [19] and similar approaches that reform how data is stored to enable faster access still manage the entire simulation domain for every time step. Comparatively, Stitch-IO avoids the vast majority of IO by implicitly rather than explicitly writing null or unchanged values for the entire simulation domain for areas that were not affected during this time step.

SciDB [20] provides the closest functionality to Stitch-IO. However, SciDB differs in some fundamental ways. First, SciDB is organized for offering generally 2-D arrays into a form where data elements can be searched and queried. Stitch-IO is fundamentally 3-D arrays, with the option for a tensor at each site, but optimized for finding blocks rather than finding individual data values.

Other attempts to use database technology as part of IO operations and storage have varied over the years. No, et al. [21] investigated using a relational database to store metadata that linked to a separately stored data file. This work never materialized into a popular IO library product presumably because the database was a fixed service (MySQL or Postgres) and not directly portable with the data to an archive or a different platform. Stitch-IO addresses this limitation by using an easily portable single package.

R-trees and other decomposition techniques are efficient at querying sub-volumes within a space. It assumes that a given space will have multiple areas. If each output is a different volume, a new set of R-trees is required for each output. Stitch-IO is different in that there is one virtual space per time step that is comprised of the most recent data written to each location. There is not a single, coherent view of this data on which to place an R-tree index. Multi-version B-trees [22] similarly want to work against a small data item. The equivalent for scale-up simulations is the entire simulation domain. This storage technique may be useful as an alternative implementation, but evaluating this is left for future work.

The StitchData [23] machine learning data management system has no connection with this project.

B. Computational Models

Mathematically, the *stitching* approach is no different from any other full-fidelity simulation. For example, an adaptive mesh refinement approach adjusts the computation mesh based on how the simulation progresses leading to some areas having more detail than others. Other approaches that use prediction to find the "busy" areas using previous time steps and extrapolation and reduced order modeling have similar limitations. These other approaches either guess or use calculation to determine what to compute. For *stitching*, it is either known ahead of time where the computation will be required or immediately obvious based on the just completed computation, eliminating the need for any potential prediction errors or additional computation. *Stitching* runs the full model everywhere, but takes advantage of the limited changes per time step to isolate both what part of the simulation domain is calculated on as well as what data is stored. Dynamic load balancing approaches shift work around based on where the most work is needed. Unlike the *stitching* approach, the whole domain is still preserved for each output.

Out-of-core techniques are more similar in that they only compute on part of the domain at a time. However, they still assume they will cover the entire domain for every time step. Stitching works based on knowing that only part of the domain requires computation on any given time step and only computing and storing data accordingly.

A deeper evaluation and discussion of the science applications and benefits are being prepared for publication separately.

C. Visualization

Time-space partitioning trees [24], and other approaches to accelerate visualization are somewhat similar. Primarily, these visualization techniques rely on a minimum difference to avoid re-computation and can introduce small, potentially invisible, errors. These visualization techniques are assuming the entire simulation domain potentially changes each time step, such as is common for CFD applications. As these simulations progress, some areas may settle, at least temporarily, into minimal, inconsequential changes related to rendering. For the

stitching approach, we never lose fidelity because we only compute on areas that are changing for the given time step.

V. Conclusions and Future Work

Stitch-IO offers a new way to think about IO library construction that is strongly organized to support the application needs resulting in a radically smaller storage size and reasonable time expenditures. The incorporated database technology offers richer data selection features without having to recreate such functionality on a custom data storage format.

For future work, we wish to demonstrate Stitch-IO with a finite element model to understand what surprises this environment may offer. Our initial explorations have suggested few, if any, surprises. Exploring the potential analysis benefits for using the storage approach even though there is a fixed simulation domain and the entire domain is output for each step is also desired.

Additional potential customers within the NNSA complex are also being recruited to address their intractable simulation data needs as well. These additional cases may offer new models that prove difficult to address, such as non axis aligned bounding boxes and non-rectangular simulation domain decompositions. While an aligned, rectangular bounding box can allow this to work today, it is not ideal. Exploring these options may reveal additional opportunities for optimization.

This library is available (at publication time) on GitHub under an LGPL v2.1 license.

Acknowledgements

We thank Steve Plimpton for guidance on integrating Stitch-IO into *spparks* and helpful scaling study discussions and Stewart Silling and Veena Tikare for support and helpful discussions related to this work. We are grateful to the following Sandia programs for funding this work: 1) ASC Physics and Engineering Models (P&EM) Advanced Manufacturing, Materials Performance and Solid Mechanics, 2) Integrated Modeling and Applications, and 3) Advanced Certification Program. This work was also supported in part under the U.S. Department of Energy NNSA ATDM project funding and the U.S. Department of Energy Office of Science, under the SSIO grant series, SIRIUS project and the Data Management grant series program manager Lucy Nowell.

References

[1] T. M. Rodgers, J. A. Mitchell, and V. Tikare. A Monte Carlo model for 3D grain evolution during welding. *Modelling and Simulation in Materials Science and Engineering*, 25(6):064006, 2017.

[2] S. Plimpton, A. Thompson, and A. Slepoy. SPPARKS. http://spparks.sandia.gov/index.html, 2009.

[3] S. Plimpton et al. Crossing the mesoscale no-man's land via parallel kinetic Monte Carlo. Technical Report SAND2009-6226, Sandia National Laboratories, 2009.

[4] Lockheed Martin. Giant satellite fuel tank sets new record for 3-d printed space parts. https://news.lockheedmartin.com/2018-07-11-Giant-Satellite-Fuel-Tank-Sets-New-Record-for-3-D-Printed-Space-Parts, 2018. Accessed: 2018-08-16.

[5] Jay Lofstead, Jai Dayal, Ivo Jimenez, and Carlos Maltzahn. Efficient transactions for parallel data movement. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 1–6, New York, NY, USA, 2013. ACM.

[6] Jay Lofstead, Jai Dayal, Karsten Schwan, and Ron Oldfield. D2t: Doubly distributed transactions for high performance and distributed computing. In *IEEE Cluster Conference*, Beijing, China, September 2012.

[7] Michael Owens. Embedding an sql database with sqlite. *Linux Journal*, 2003(110):2, 2003.

[8] Margaret Lawson and Jay Lofstead. Using a robust metadata management system to accelerate scientific discovery at extreme scales. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, PDSW-DISCS '18, pages 13–23. IEEE, 2018.

[9] Oracle berkeley db sql api vs. sqlite api – a technical evaluation. https://www.oracle.com/technetwork/database/database-technologies/berkeleydb/learnmore/bdbvssqlite-wp-186779.pdf, September 2010.

[10] Oracle berkeley db sql api vs. sqlite api – integration, benefits and differences. https://www.oracle.com/technetwork/database/database-technologies/berkeleydb/bdb-sqlite-comparison-wp-176431.pdf, November 2016.

[11] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[12] R Rew, E Hartnett, J Caron, et al. Netcdf-4: Software implementing an enhanced data model for the geosciences. In *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*, 2006.

[13] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.

[14] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39, Nov 2003.

[15] Bogdan Nicolae and Franck Cappello. Ai-ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 155–166, New York, NY, USA, 2013. ACM.

[16] Jay Lofstead, Gregory Jean-Baptiste, and Ron Oldfield. Delta: Data reduction for integrated application workflows and data storage. In *International Conference on High Performance Computing*, pages 142–152. Springer, 2016.

[17] Medha Bhadkamkar Jorge Guerra Luis Useche, Sam Burnett Jason Liptak, and Raju Rangaswami Vagelis Hristidis. Borg: Block-reorganization for self-optimizing storage systems. In *7th USENIX Conference on File and Storage Technologies*, 2009.

[18] Yuan Tian, Scott Klasky, Hasan Abbasi, Jay Lofstead, Ray Grout, Norbert Podhorszki, Qing Liu, Yandong Wang, and Weikuan Yu. Edo: improving read performance for scientific applications through elastic data organization. In *2011 IEEE International Conference on Cluster Computing*, pages 93–102. IEEE, 2011.

[19] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[20] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[21] Jaechun No, Rajeev Thakur, and Alok Choudhary. Integrating parallel file i/o and database support for high-performance scientific data management. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 57. IEEE Computer Society, 2000.

[22] Benjamin Sowell, Wojciech Golab, and Mehul A. Shah. Minuet: A scalable distributed multiversion b-tree. *Proc. VLDB Endow.*, 5(9):884–895, May 2012.

[23] Stitch data. https://stitchdata.com, May 2019.

[24] Han-Wei Shen, Ling-Jen Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, VIS '99, pages 371–377, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.