Data-Intensive Spatial Filtering in Large Numerical Simulation Datasets

Kalin Kanov,* Randal Burns,*¶ Greg Eyink,†¶
Charles Meneveau‡¶ and Alexander Szalay§¶
*Department of Computer Science, Johns Hopkins University
†Department of Applied Math and Statistics, Johns Hopkins University
‡Department of Mechanical Engineering, Johns Hopkins University
§Department of Physics and Astronomy, Johns Hopkins University
¶Institute for Data Intensive Engineering and Science

***Abstract*—We present a query processing framework for the efficient evaluation of spatial filters on large numerical simulation datasets stored in a data-intensive cluster. Previously, filtering of large numerical simulations stored in scientific databases has been impractical owing to the immense data requirements. Rather, filtering is done during simulation or by loading snapshots into the aggregate memory of an HPC cluster. Our system performs filtering within the database and supports large filter widths. We present two complementary methods of execution: *I/O streaming* computes a batch filter query in a single sequential pass using incremental evaluation of decomposable kernels, *summed volumes* generates an intermediate data set and evaluates each filtered value by accessing only eight points in this dataset. We dynamically choose between these methods depending upon workload characteristics. The system allows us to perform filters against large data sets with little overhead: query performance scales with the cluster's aggregate I/O throughput.**

I. Introduction

Data-intensive architectures have emerged as attractive platforms for storing and managing large datasets generated from numerical simulations. These systems achieve high aggregate throughput based on I/O and network bandwidth [1]. One example of such a system is the Turbulence Database Cluster at Johns Hopkins [2], which provides public access to world-class turbulence simulations, allowing users to perform sophisticated analyses. It has served more than 100 billion point queries to date. A variety of data-intensive computations can be executed on the nodes of the database cluster, including spatial and temporal interpolation, spatial differentiation, and fluid particle tracking. However, filtering operations have remained out of reach, owing to their immense data requirements. Such functionality would greatly enhance the utility of the datasets stored in the Turbulence Database Cluster.

Many studies of scale interactions in computational turbulence require spatial filtering of the vector and scalar fields resulting from simulations [3]. Filtering or coarse-graining consists of computing a convolution in real space of a filter kernel and a vector or scalar field (or multiplication in Fourier space). The operation is fundamental to analysis in disciplines as diverse as signal processing, geostatistics, and computer graphics. For large data sets, these operations are performed typically on individual time steps (or snapshots) stored in the aggregate memory of an HPC cluster. In order to filter world-class simulation data outside of HPC environments, techniques need to be developed that operate on data sets accessed from disk drives. The goal of our work is to make spatial filters efficient to evaluate in real-space on data-intensive clusters.

The difficulty with implementing filtering in real space stems from its massive data requirements. The amount of data that needs to be accessed scales with both the number of locations at which the filtered values are evaluated and with the width of the filter kernel, which can be a substantial fraction of the resolution of the entire simulation. A naive approach that evaluates each value independently reads the same data multiple times when the filter kernels of multiple points overlap. Computing in Fourier space is impractical as this requires the computation of the Fourier transform of an entire snapshot.

We provide a query processing framework that performs filtering based on purely sequential reads and writes in which I/O bounds performance. The framework incorporates two algorithms for the evaluation of filtering workloads. For sparse workloads on smaller kernels, we use the I/O streaming batch query processor [4]. For larger, dense workloads, we introduce a two-pass *summed volumes* algorithm that dynamically builds an intermediate data set based on summing values of each variable to be filtered and evaluates the filtered values in a subsequent pass over the intermediate data. This process was inspired by the use of summed area tables for texture mapping in computer graphics [5], extending it to operate sequentially over dynamically generated three-dimensional data. Both techniques are *data-driven* in that they exploit data sharing among kernels by decomposing computations into partial sums. For each batch query, we choose between these techniques depending upon workload characteristics, such as the total amount of data to be read and the number of target locations to be filtered.

The summed volumes method generates the intermediate dataset of the summed values of the variables to be filtered on-demand, rather than using a precomputed and stored version of this data. While it is possible to precompute and store such a dataset, each precomputed field is as large as the original data and would require tens of terabytes of additional

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

storage. This in turn limits the number of variables that can be filtered. Different scientific analyses require filtering multiple variables and non-linear combinations of variables. On-demand computation evaluates any such combination of variables at runtime from the original (unfiltered) data.

The query processing framework evaluates batch filter queries for computational turbulence 5 to 40 times faster than a naive method of execution. It scales well to multiple nodes in a scientific database cluster and makes effective use of its aggregate I/O throughput. We show results for up to 8 database nodes with 4 virtual servers per node for a total of 32 virtual servers with 1.0 GB/s aggregate I/O read rate.

II. Background

We start with a brief description of the JHU Turbulence Database cluster and its I/O streaming data-driven batch-processing engine. Spatial filtering is designed specifically for the cluster's workloads. We implement filtering algorithms within this architecture and perform all experiments through a test or development Web server accessing the production database nodes.

A. Turbulence Database Cluster

The Turbulence Database provides public access to world-class, high-resolution turbulence simulations through Web-services. The entire space-time histories of Direct Numerical Simulations (DNS) of turbulent flows are stored in a cluster of relational databases [2]. Two datasets are currently stored in the cluster—the output of a forced isotropic turbulence and magnetohydrodynamic simulations. The two datasets occupy over 70TB of the 1.1PB capacity Graywulf cluster [1]. Each of the datasets consists of 1024 simulation time-steps over a 1024^3 regular, spatial grid. Clients access the databases programmatically by means of a Web-services interface. Users retrieve batches of point queries to the simulation parameters (e.g. velocity, pressure, magnetic field, magnetic vector potential and forcing), their interpolated values, their spatial derivatives, or their time evolution [6].

We use the Morton z-order, space-filling curve to spatially partition datasets across the cluster and to index the data within each partition [2]. The entire space is divided into atoms of size 8^3 voxels. Each atom is indexed by time-step and Morton-code (z-order index) of its lower left corner. Thus, each database record consists of three attributes—time-step, z-index and a blob storing the simulation output data. The spatial access method is a standard B+-tree clustered index keyed on the time-step and z-index. We vertically partition the simulation output to multiple tables based on workload access patterns, e.g. all velocity components u_x, u_y, u_z are stored in a single table and pressure is stored separately.

B. I/O Streaming

I/O streaming performs decomposable kernel computations by means of partial-sums and a single pass over the data. It eliminates redundant I/O and exploits data sharing when the kernels of multiple point queries overlap [4]. We developed I/O streaming in the Turbulence Database cluster for the evaluation of batch queries. The method evaluates batch requests that consist of multiple target locations. The method is data-driven as individual queries are evaluated incrementally as each relevant data atom that they need to access is retrieved from disk. The entire batch of queries is executed in a single pass over the data. What enables this mode of execution is that each individual query can be decomposed into partial-sums with the result of the query being the assembled partial-sums.

The I/O streaming method includes a pre-processing step. During this step the method reads the set of input locations and for each location determines the set of data atoms that cover its associated kernel of computation. Each data atom is identified by the z-index of its lower left corner. These z-indexes form the keys of a dictionary data structure, in which the values of the dictionary are lists of queries. Each query in such a list needs to access the atom with the corresponding z-index. Thus, during the pre-processing step a query is added to the list associated with each atom that it needs to access if the atom already exists in the dictionary or a new entry is created if it does not.

Once all queries have been processed the keys of the dictionary give us the set of unique data atoms that have to be retrieved from the database. The data atoms are retrieved in a single pass over the data. This forms our I/O stream. As each data atom is retrieved from disk it is accessed by each query that needs data from it in succession. A partial sum is computed and stored for each query over the part of the kernel that overlaps the retrieved data atom. The sum is updated when more of the data atoms needed by the query become available. This incremental evaluation applies to any decomposable kernel computation.

A decomposable kernel computation over grid data is one that consists of linear combinations of the values at grid nodes. In one dimension it is given by:

$$g(x') = \sum_{i=1}^{N} l_i(x') \cdot f(x_{n-\frac{N}{2}+i}) \tag{1}$$

in which the kernel of computation is of size N, the data are stored at discrete locations x_i on the grid (integer i), the location x_n is the one closest to x' and l_i are coefficients that do not depend on the data, but can depend on the target location x' and the grid resolution.

Given Π, a permutation of the set $(1, ..., N)$, and a set $(S_1, ..., S_m)$ of contiguous non overlapping subsets of Π, a decomposable kernel computation can be broken down into partial sums as follows:

$$g(x') = \sum_{j=1}^{m} g_j(x') \tag{2}$$

in which g_j are the partial-sums over the sets S_j:

$$g_j(x') = \sum_{i \in S_j} l_i(x') \cdot f(x_{n-\frac{N}{2}+i}). \tag{3}$$

This allows us to perform incremental evaluation if only a portion of the data is available.

To implement filtering with I/O streaming, the coefficients l_i define the filter function or convolution to be applied. For example, for a box filter all of the coefficients are equal to the inverse of the volume of the region defined by the filter width as in that case the filtered value is equal to the average of the values in this region.

III. Filtering

We describe the use of spatial filtering in computational turbulence in order to motivate how this function enhances the utility of the JHU Turbulence Database cluster. Specifically, we characterize how filtered fields from our databases will improve sub-grid modeling in large-eddy simulations.

A common approach to studying turbulent flows conducts numerical simulations of the Navier-Stokes equations. Such *direct numerical simulations* (DNS) resolve all time and length scales of the solution. The limitation of DNS is that interesting, real-world turbulent flows are extremely expensive and, in most cases, impractical to compute, because of the amount of computation needed to resolve the smallest scales at high resolution.

In contrast, Large Eddy Simulations (LES) compute only the large scales and model the smallest scales in order to reduce computational requirements. Thus, LES can be used to model much more complex flows. However, the development of realistic models for the small scale motions remains an open research problem.

In LES, the Navier-Stokes equations are transformed by means of low-pass filtering and solutions yield a filtered velocity field. Separating the small and large scales of the motion is done by convolving the velocity filed with a kernel, $G_\Delta(\mathbf{r})$ [7] in which Δ is the length-scale, down to which the fluid motions are resolved. The filtering operation (denoted by an over-line) is then given by

$$\overline{\mathbf{u}}(\mathbf{p},t) = \int G(\mathbf{r})\mathbf{u}(\mathbf{p}-\mathbf{r},t)d\mathbf{r}, \tag{4}$$

in which the integration may be over the entire domain of the flow, depending on the spatial support of $G(\mathbf{r})$.

In order to evaluate the models used in LES, one can compare the simulation results with available experimental data or data from a DNS. To enable meaningful comparisons [8], the latter must be filtered or coarse-grained to a resolution comparable to the LES. Another important method to evaluate sub-grid scale models in LES is to study field variables that arise from filtering the Navier-Stokes equations leading to LES. When filtering the nonlinear advection term, one obtains the so-called sub-grid stress tensor, defined according to:

$$\tau_{ij} = \overline{u_i u_j} - \overline{u}_i \overline{u}_j, \tag{5}$$

where u_i, u_j are the components of the velocity field. This tensor represents the momentum fluxes associated with the small-scale turbulent motions that are not explicitly resolved in LES, but that must be included in the evolution equation of the large-scales in LES. The fundamental "closure problem" in turbulence [3] is to express τ_{ij} in terms of the filtered large-scale velocity field $\overline{u}_i$ (sub-grid scale modeling). The ability to measure the "exact" stress field according to its definition (Eq. 5) from a full solution of the Navier-Stokes equations is important to improve the quality and accuracy of sub-grid scale models.

There are a variety of different filter functions that can be used, including the spectrally sharp filter, the Gaussian filter, and the box filter [3]. The I/O streaming method can be applied to any filter function, whereas the alternative summed volumes method applies only to a filter function with uniform weights in the convolution kernel, i.e. the box filter. Many studies use the box filter because it has local support and is easily computed by the average of the values in the region around the target location: the points $\mathbf{p}$, where $(\mathbf{p}_0 - \frac{1}{2}\Delta) \leq \mathbf{p} \leq (\mathbf{p}_0 + \frac{1}{2}\Delta)$ given a target location $\mathbf{p}_0$.

IV. Computing Summed Volumes

We describe the summed volumes technique for evaluating box-filters on the largest batch queries: those that filter many points at large kernel widths. These queries also capture some of the most interesting science, such as creating a high-resolution smoothed view of a region of interest. The I/O streaming technique is more general, it supports arbitrary filter functions, and is often more efficient. However, I/O streaming exhibits scalability problems when each input data contributes to many kernels. It becomes expensive to compute the partial sum for each target filter kernel individually. Summed volumes avoids this problem by sharing partial sums computations among all kernels at the expense of a two-pass algorithm and the generation of an intermediate dataset. As a result, the summed volumes technique exhibits stable performance over all parameterizations of kernel sizes and numbers of points.

The summed volumes algorithm is inspired by the use of summed area tables for texture mapping [5]. We extend the technique to volumetric data, to support sequential I/O, and to dynamically compute the sums rather than precomputing and storing the dataset.

The first pass of the algorithm determines the bounding data region of the target locations and generates a summed volumes dataset over that region. Every data point in this intermediate dataset stores the sum of all data values below and to the left of it. The second pass of the algorithm uses this summed volumes dataset to efficiently compute a box filter for every target location. Figure 1 depicts this process for a three-dimensional dataset. We elaborate on the two passes of the algorithm below.

The on-demand computation of a summed volumes dataset in d dimensions determines the bounding data region of the set of input points, defined by lower left $H_L = (x_{0_L}, x_{1_L}, ..., x_{d-1_L})$ and upper right $H_U = (x_{0_U}, x_{1_U}, ..., x_{d-1_U})$ corners. The coordinate x_{i_L} is given by $min(x_i) - \frac{1}{2}\Delta$, where the minimum is taken over the x_i-th coordinate of all of the input points. The coordinate x_{i_U} is given by $max(x_i) + \frac{1}{2}\Delta$, where the maximum is again taken

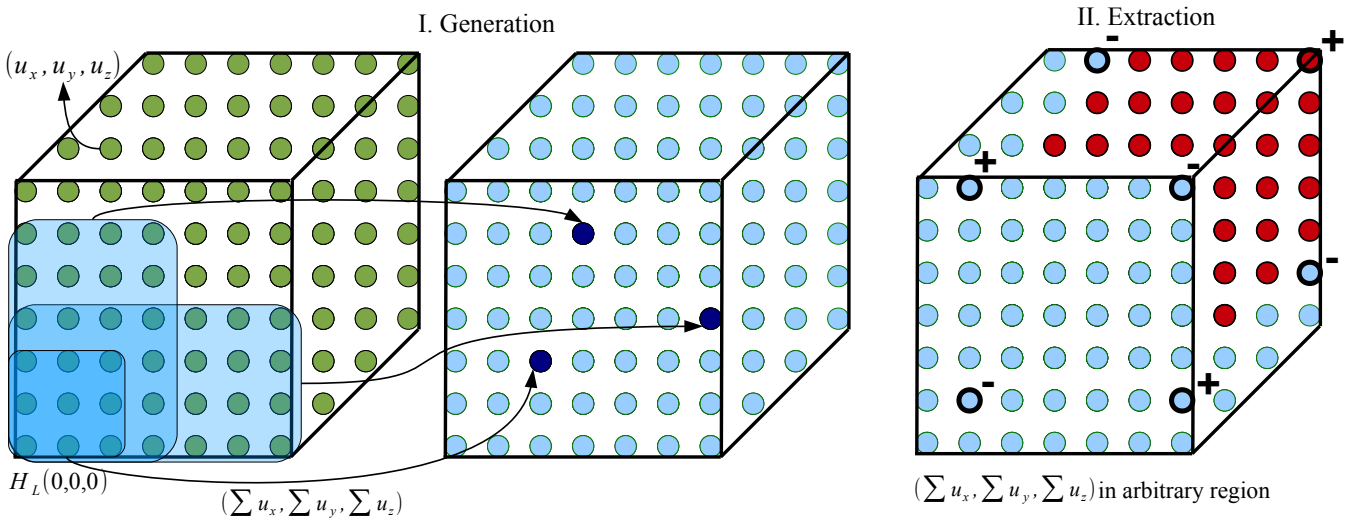


Fig. 1. Using summed volumes to compute the sum of grid values over an arbitrary region of interest. Every point in the summed volumes dataset (middle) stores the sum of all values between the lower left corner $H_L(0,0,0)$ of the grid and that point in the original data set (left). The image on the right depicts the extraction process from the generated summed volumes dataset. The filtered value in an arbitrary region (shown in red) is extracted by subtracting the sums outside of this region from the value at its top right corner. This process looks at eight data points.

over the x_i-th coordinate of all of the input points. This region defines the total amount of data that has to be retrieved and processed.

In the case of the Turbulence Database cluster, the set of database atoms that cover this region are retrieved and processed in Morton z-order. This is the same order in which data are laid out of disk, ensuring that disk I/Os are performed to increasing offsets, which is as or more efficient than a single sequential pass.

The summed volumes dataset is manifested on an d-dimensional grid. The value at every grid point in the summed volumes dataset is the sum of all values contained in the hyper-rectangle defined by the corresponding point in the original dataset and the lower left corner of the grid (Figure 1). The dynamic generation of the summed volumes dataset proceeds as follows.

Points in the original dataset are summed in Morton z-order in a single pass. The z-order visits points in a grid in non-decreasing order in each dimension; when accessing a point, the sums of all its lower left adjacent points have already been computed. We add the value of point $(x_0, x_1, ..., x_{d-1})$ to the sums already computed for its lower adjacent points to calculate the sum of all points between that data location and the lower left corner of the grid:

$$s(x_0, x_1, ..., x_{d-1}) = u(x_0, x_1, ..., x_{d-1}) + \sum_{i_0=0}^{1}\sum_{i_1=0}^{1}...\sum_{i_{d-1}=0}^{1}(-1)^{i_0+i_1+...+i_{d-1}-1}. s(x_0 - i_0, x_1 - i_1, ..., x_{d-1} - i_{d-1}) \quad (6)$$

in which u denotes values in the original dataset, s denotes values in the summed volumes dataset and $s(x_0, x_1, ..., x_{d-1})$ is assumed to be initialized to 0. In total 2^d additive operations and 2^{d-1} lookups are performed per data point.

The next step in the algorithm computes the filtered value from the generated summed volumes dataset. First, we extract the sum of values in the region defined by the filter kernel. The sum of all values lying inside an arbitrary hyper-rectangle defined by its lower left $H_L = (x_{0_L}, x_{1_L}, ..., x_{d-1_L})$ and upper right $H_U = (x_{0_U}, x_{1_U}, ..., x_{d-1_U})$ corners is given by:

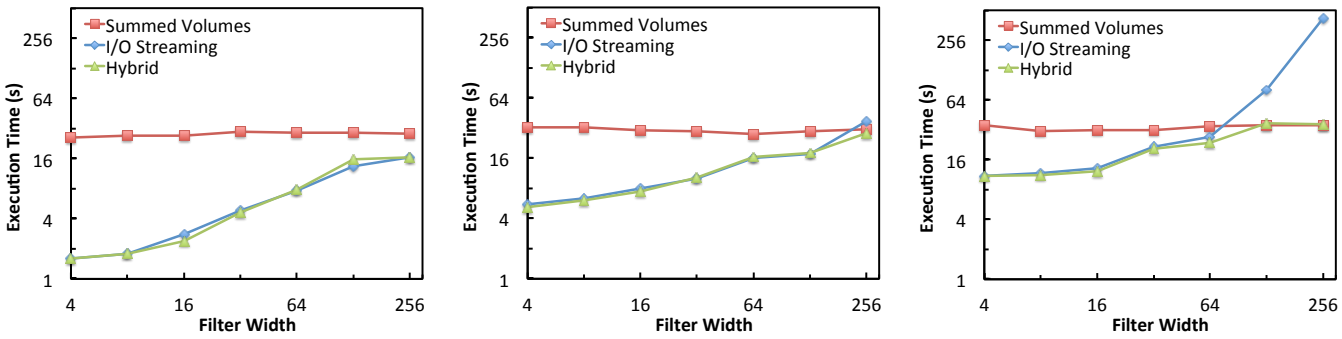
$$\sum_{i_0=0}^{1}\sum_{i_1=0}^{1}...\sum_{i_{d-1}=0}^{1}(-1)^{i_0+i_1+...+i_{d-1}}. s(x_0(i_0), x_1(i_1), ..., x_{d-1}(i_{d-1})) \quad (7)$$

where the coordinates $x_j(i_j)$, for $j \in (0, 1, ..., d-1)$, are equal to x_{j_U} if $i_j = 0$ and $x_{j_L} - 1$ if $i_j = 1$. Thus, 2^d points have to be accessed.

Computing a box filter takes the average of the data values in a region centered on the target location. With the above technique we can compute the sum of the data values in the filter kernel associated with a target location $p' = (x'_0, x'_1, ..., x'_{d-1})$ by specifying the corners of the region as $H'_L = (x'_0 - \frac{1}{2}\Delta, x'_1 - \frac{1}{2}\Delta, ..., x'_{d-1} - \frac{1}{2}\Delta)$ and $H'_U = (x'_0 + \frac{1}{2}\Delta, x'_1 + \frac{1}{2}\Delta, ..., x'_{d-1} + \frac{1}{2}\Delta)$. Therefore, given the sum in that region all that one has to do is divide by the region's volume.

In three dimensions, we access eight points in the summed volumes dataset to compute the sum in a rectangular region. We then multiply by a filter function in order to compute the filtered value. In this case only constant filter functions (e.g. the inverse of the volume) can be used as the sums are precomputed. The rightmost image in Figure 1 shows the points that have to be accessed and whether the values are to be added or subtracted in order to compute the sum of the points in red (one of the points accessed is not shown as it is in the back of the grid).

On-demand computation of the data sets allows us to



(a) 1024 points (b) 10,240 points (c) 102,400 points

Fig. 2. Execution time for randomly distributed points in the entire 1024^3 space with varying filter widths.

support user-defined filters using complex expressions, such as $\overline{u_i u_j}$ that is needed to evaluate the sub-grid stress tensor (Eq. 5), While the intermediate data could be precomputed, this quickly becomes impractical. We have already identified several interesting filtered fields and each precomputed field would be as large as the storage for a variable from which it is derived. For example, the 1024 time steps of the raw velocity field data alone occupy 12 TBs on disk. Precomputing and storing summed volumes datasets for each time step for the 9 quantities needed for the evaluation of the sub-grid stress tensor will require 36 TBs of additional storage.

V. Putting It All Together

The I/O streaming and summed volumes techniques exhibit different performance characteristics and dynamically choosing between them on a per-query basis improves performance dramatically. Summed volumes works better for workloads in which the filter kernels have substantial overlap. In these cases, both techniques retrieve the same amount of data from disk. However, I/O streaming routes each data point to all of the kernels that need it, performing a different computation for each data point in each kernel. By manifesting an intermediate data set, summed volumes shares the computation of each data point among all kernels. This reduces computation, but it also restricts summed volumes to box filters in which each data point contributes equally to each kernel.

Our process for choosing between I/O streaming and summed volumes relies on analysis of the scaling properties of these techniques. We then determine how to parameterize this analysis with experimental results to define a dynamic query optimizer. Given an atom size s^3 and kernel size k^3, the average number of data atoms that have to be accessed by a query is given by $(\frac{k+s-1}{s})^3$. Therefore, since the data are partitioned into atoms, for a batch consisting of p queries, the total number of individual data points that need to be accessed can be estimated as $p \cdot (k+s-1)^3$. We define the density of input queries or kernel overlap as the number of queries accessing the same data point. It can be estimated as $\rho = \frac{p \cdot (k+s-1)^3}{x \cdot y \cdot z}$, in which x, y, and z define the dimensions of the bounding data region for the entire batch.

We analyze both the I/O and computation scaling properties of both techniques. I/O streaming retrieves $p \cdot (\frac{k+s-1}{s})^3$ atoms from disk and performs roughly $p \cdot (k+s-1)^3$ operations because the size of each atom is s^3. Summed volumes retrieves the entire bounding region of data, $\frac{x \cdot y \cdot z}{s^3}$ atoms, generates the intermediate data set using $8 \cdot x \cdot y \cdot z$ operations and extracts the filtered results in $8 \cdot p$ operations for a total of $8 \cdot (x \cdot y \cdot z + p)$ operations.

When kernels have little overlap, density $\rho \leq 1$, I/O streaming accesses less data and performs fewer operations. I/O streaming is always preferred.

When kernels overlap, density $\rho > 1$, the best choice depends upon the density and number of points. We assume that I/O streaming and summed volumes access the same amount of data. The data atoms covering the entire region $(\frac{x \cdot y \cdot z}{s^3})$ provide an upper bound for I/O streaming and an exact figure for summed volumes. The I/O requirements match in practice, because I/O streaming's best strategy performs a sequential read of the entire data volume. The techniques differ only in their computational requirements.

We empirically determine the parameters at which the computation of I/O streaming and summed volumes match. We use summed volumes if $\frac{p \cdot (k+s-1)^3}{8 \cdot (x \cdot y \cdot z + p)} > 10$ and use I/O streaming otherwise. For most batch queries $p \ll x \cdot y \cdot z$, the fraction $\frac{p \cdot (k+s-1)^3}{8 \cdot (x \cdot y \cdot z + p)}$ is approximately $\frac{\rho}{8}$. The constant 10 reflects the difference between the sequential memory accesses of I/O streaming and the random memory accesses of summed volumes. Random memory accesses are $\sim$10 times slower. This inequality covers the case when kernels do not overlap as well, choosing I/O streaming at low densities.

VI. Experimental Results

We evaluate the performance of the I/O streaming and summed volumes methods for batch spatial-filtering queries on microbenchmarks and anticipated usage pattern workloads derived from scientists' initial queries. While we have traced more than 100B point queries in the Turbulence Database cluster [9], spatial filtering represents an entirely new capability and none of the existing queries have comparable data requirements. We also compare performance with direct

execution of a set of queries that create coarse-grain representations of the entire space: a query type that will be used frequently in practice. The benchmarks show that direct evaluation of individual queries is impractical, while using the developed query processing framework results in 5 to 40 times improvement in the execution time of queries. The query processing framework is also able to effectively determine which method to utilize for all workloads.

The experiments were run through a development Web-server mediator that connects to the production nodes of the Turbulence Database cluster. The data are evenly partitioned across either 4 or 8 nodes according to spatial regions in the Morton z-order. Database nodes are 2.33 GHz dual quad-core Windows 2008 servers with SQL Server 2008 and 24 GB of memory. Each node is connected to two MD1000 SAS disk boxes that contain a total of 30 750 GB, 7,200 rpm SATA disks. We utilize 4 virtual servers per database node (unless stated otherwise) in order to make use of the multicore parallelism available on each node. Data tables are partitioned across 4 logical data volumes on each of the nodes. The measured I/O read rate through SQL Server Management Studio for our queries is 130 MB/s per node, for a total aggregate I/O read rate of 1.0 GB/s across 8 nodes. All experiments were run with a cold cache.

Figure 2 shows the execution time of three batch queries as a function of the filter widths for 1024, 10,240 and 102,400 locations randomly distributed in the entire 1024^3 volume. The query calculates the filtered value of the vector field at each individual location. The figure compares I/O streaming, summed volumes, and the hybrid method that uses the query processing framework of Section V to choose between them on a per query basis.

The execution time of summed volumes remains more or less constant for all three batches and at all of the different filter widths. Summed volumes manifests the same intermediate data set in all cases. The amount of data that must be read and processed depends only on the bounding data region of the entire batch. Since the points are distributed randomly in the entire volume, the entire data volume is read and a summed volumes data set of an entire time step is created. Summed volumes filters an entire 1024^3 timestep in just over 30 seconds. I/O streaming typically outperforms summed volumes because it performs I/O only to the data atoms needed by the filtering kernels. With fewer points and narrower kernels, this can be much less than an entire timestep as indicated by the results.

However, the performance of I/O streaming degrades for larger numbers of points and wider kernels. At some point, I/O streaming retrieves the entire volume of data. Beyond this point, I/O remains constant and the execution time begins to be dominated by computation, which scales as $p \cdot k^3$ for I/O streaming, in which p is the number of positions and k the filter width. At the largest filter widths in our experiment, I/O streaming can be more than ten times worse than summed volumes. The query processing framework uses the workload characteristics to effectively decide which method to deploy.

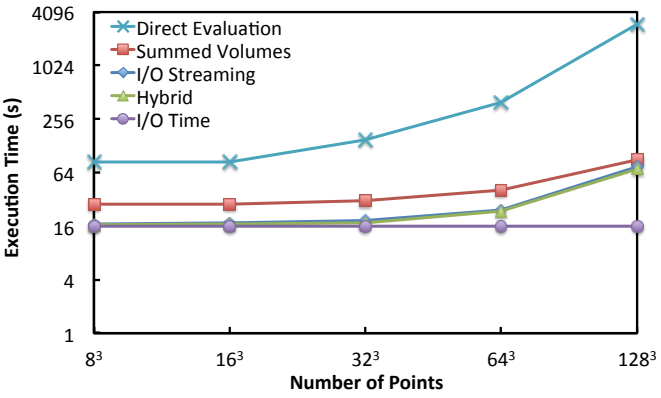


Fig. 3. Execution time for coarse graining queries covering the entire data volume at different density.

The hybrid approach tracks the best performance of either method.

An application of spatial filtering for large scientific datasets is the generation of lower resolution dataset or coarse-graining of the high-resolution data. This is done by filtering the data at locations placed on a cubic lattice, where the points on the lattice are placed so as to cover the entire data volume. The width of the filter used in this case is a function of the separation between the points on the lattice, r. We use $2 \cdot r$ as the filter width. The execution times of queries of this type are presented in Figure 3. Since the entire data volume is coarse-grained, all of the data for a time-step is retrieved. The execution time of the I/O streaming method, summed volumes method and the hybrid method are compared alongside the time required to perform just the I/O. The execution time of a direct implementation of spatial filtering is also presented. In this case the individual queries forming a batch are executed one at a time by retrieving the data necessary for each query and evaluating the filter before moving on to the next.

As the number of locations in the cubic lattice increases the execution time of the direct evaluation of individual queries goes from minutes to hours. The execution time of I/O streaming on the other hand tracks the I/O time for batch queries with a small to medium number of target locations. The method is able to achieve over 1.0 GB/s aggregate read rate in those cases. This makes it five times faster than direct evaluation for the fewest number of points and more than 40 times faster for the largest set of points. Because the query density is small for all workloads of this type, we would expect the execution time to remain close to the I/O time in all cases. The processing that happens on the Web server and the fact that its resources are shared among other development deployments is the reason for the increase seen at a large number of target locations. Because we measure the execution time at the client as the time to submit a batch query and obtain the results, this includes the transfer of target locations to the Web server and the distribution of queries to each of the database nodes. For coarse-graining fields, I/O streaming always outperforms summed volumes, because as the number

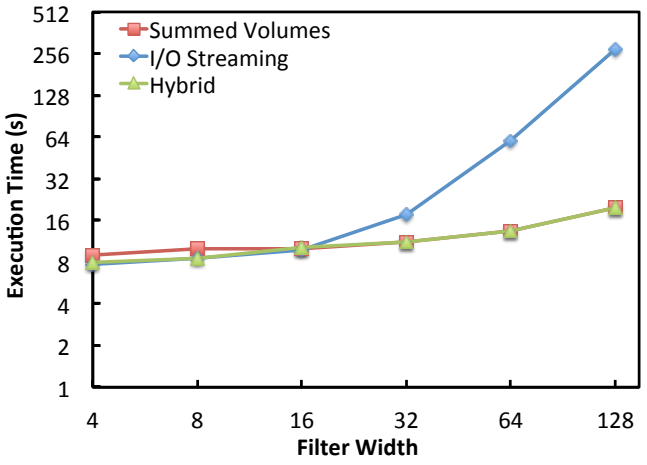


Fig. 4. Execution times of queries requesting the filtered value of the velocity field on a cubic lattice of 64^3 points.

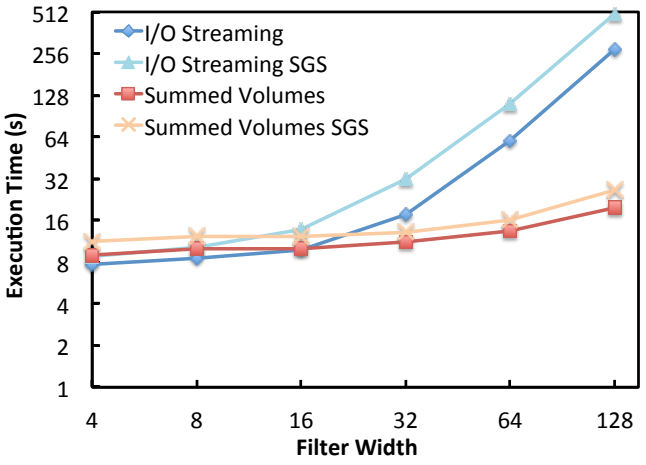


Fig. 5. Execution times of queries requesting the sub-grid stress tensor (SGS) compared with the execution times of queries requesting just the filtered value of the velocity field on a cubic lattice of 64^3 points.

of points increases, the filter width decreases so that the kernel overlap of queries remains constant.

Another application of spatial filtering generates a smoothed view of a region of interest. We present timing results for queries of this type in Figure 4. The locations where a filtered value is desired are placed on a cubic lattice of size 64^3 and are separated by $4 \cdot dx$, in which dx is the grid resolution of the simulation. At the point where the execution times of the two methods crossover (filter width 16), there are already $\sim$180 queries accessing every data point and this number increases to $\sim$11,600 for filter width 128. The summed volumes method clearly performs better for the queries that perform significant smoothing and use large filter widths. The query processing framework detects this for this query type as well and chooses the summed volumes method for the evaluation of the large filter width queries. It utilizes the I/O streaming method otherwise.

As discussed in Section III, there is interest in computing complex filter quantities of non-linear combinations of simulation parameters, such as the sub-grid stress tensor. In Figure 5 we show the execution times of sub-grid stress tensor queries (labeled "SGS") for the same distribution of target locations as in the experiment discussed above (Figure 4). We compare these execution times with those requesting just the filtered velocity components. As we can see in the figure the overhead of maintaining and computing 9 quantities as opposed to 3 is not significant even though intermediate results are three times as large. It is $\sim$24% on average for the summed volumes method and no more than 34%. For the I/O streaming method it grows to $\sim$80%, but only for the queries with large filter widths for which the hybrid technique would choose summed volumes. Otherwise it is less than 20%.

Our last experiments look at the scale-out of the service and the parallelization of computation using summed volumes. Figure 6 presents the execution times of batch queries consisting of 102,400 points randomly distributed in the entire 1024^3 volume with varying filter widths. These are the same queries as those presented in Figure 2(c). The execution times of these queries are evaluated for a database configuration consisting of 4 nodes and another consisting of 8 nodes. For the 8 node configuration, we also present results for a setup that treats each individual node as 2 or 4 virtual servers (labeled "VS"). The data reside in the same database table, but each virtual server only queries the portion that it is assigned. The plot also shows the I/O time for the retrieval of the data only without any additional processing.

Summed volumes has significant processing requirements, but computations can be parallelized so that I/O consumes about half of overall performance. We parallelize computation by creating virtual servers on each physical node that perform the summed volumes computations on different cores. Execution time on 8 nodes is roughly 75 seconds compared with 114 seconds on 4 nodes. Adding two and four virtual servers reduces this to 55 seconds and 34 seconds respectively. This realizes an overall speedup of 3.4 and the computation takes just 2.1 times as long as the raw I/O on 8 nodes.

VII. Related Work

The computation of spatial filters for data-intensive computing has not been extensively studied, likely owing to the fact that computing a filtered value at a single location in the spatial domain of a time series requires accessing a substantial portion of the data of an entire time step, which can be several GB in size. Additionally, data-intensive architectures and systems have only recently emerged as attractive platforms for the storage of high-resolution numerical simulation datasets. Traditionally filtering has been done during simulation [10] or on large snapshots stored in the distributed memory of an HPC cluster [11].

Filtering has been extensively studied in the field of image processing [12], [13]. The focus of this line of work is the development of new filters and filtering techniques [14] and the parallelization of the computation as the data to be filtered

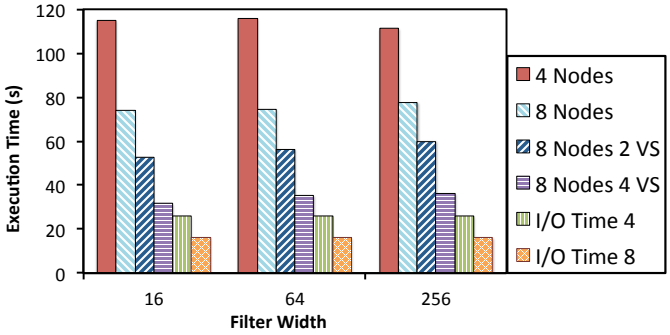


Fig. 6. Execution time of the summed volumes method on different server configurations for 102,400 randomly distributed points in the entire 1024^3 space with varying filter widths.

is not large and can fit in the memory of a CPU or a GPU [15]. Our work focuses on out-of-core solutions and external memory algorithms for the filtering of large three-dimensional scientific datasets, storing vector fields.

DataCutter [16] is a middleware infrastructure for subsetting and aggregation of large datasets on archival storage systems. DataCutter uses the term "filter" to refer to subsetting data. This concept is unrelated to the spatial filters in our work, which compute coarse-grained fields by convolution. Such processing tasks are user-defined in DataCutter and we have shown that naive implementations can make their evaluation impractical and render them unusable. The Earth System Grid [17] is a similar type of infrastructure to DataCutter, which aims to support high-performance, interactive analysis and remote access of simulation data. It makes use of the Climate Data Analysis Tools (CDAT) [18] to perform client-side analyses.

Coarse-graining and data reduction techniques are used for the visualization of large datasets [19]. This is done during a post-processing step [20] or requires parallel rendering algorithms [21] on massively parallel processors. These techniques are not designed with the goal of performing statistical analyses on the data and introduce error that makes them unsuitable for scientific experiments.

Data sieving [22] aggregates noncontiguous accesses to the file system into a few, large contiguous requests. Similarly, the methods that we present aggregate the requests of all queries in a batch. Additionally, with summed volumes the data is transformed into a summed volumes dataset for the efficient extraction of filtered quantities.

Database support for array data, such as the *Array Manipulation Language* [23] or SciDB [24], provides ways to define and manipulate arrays. As of now, these systems do not support the optimization of convolutions. I/O streaming and summed volumes could be incorporated as an execution strategy in such systems. MauveDB [25] introduces model based views for incomplete and sparse data, but does not include a treatment of batch queries with potentially overlapping data requirements.

Push-based database systems such as DataPath [26] take a data-centric approach to query processing. Data are pushed through the memory hierarchy and are shared between computations. The data are read continuously by means of table scans and pushed through waypoints that perform the required computations. The I/O streaming and summed volumes techniques also aim to share data and computation among batch queries, but achieve this by means of pre-processing a batch query and retrieving only the necessary data in a single I/O stream.

VIII. Conclusions

We have presented a query-processing framework for the execution and evaluation of batch queries that apply spatial filters to large numerical simulation datasets. We extend the I/O streaming method that we developed previously for the evaluation of interpolation and differentiation. We also develop a summed volumes method that resolves the scalability problems of the I/O streaming method for the largest filtering queries. It exhibits stable performance across all workload parameterizations. Summed volumes dynamically computes an intermediate dataset of summed volumes for each variable to be filtered and allows for the computation of filters that combine multiple variables, such as sub-grid stress tensor. We dynamically select the best performing method for each query. The result reduces query processing times by a factor of 5 to 40 when compared with direct evaluation of individual queries.

Summed volumes computes and stores intermediate data sets in memory and the memory capacity of the database cluster limits scalability. Computing a spatial filter of an entire time-step requires memory equivalent to the size of the time-step (currently 12 GBs per simulated vector field). Each time-step is distributed across the nodes of the cluster and therefore only a fraction of the total amount is retrieved in the main memory of each node. As the time-steps grow in size we expect to be able to allocate new nodes in the cluster, which will allow us to not only store more data but also have additional memory capacity. However, computing more complex filter quantities requires the generation of multiple intermediate fields. For example, computing the sub-grid stress tensor requires twice as much memory as filtering the vector field alone. The intermediate dataset that is stored in memory scales with the number of independent quantities to be filtered and is thus limited by the memory capacity of the cluster.

In our future work, we intend to materialize the intermediate summed volumes dataset to a temporary table on disk when filter quantities consist of multiple variables that are too large to fit in memory. Creating such a temporary table can serve as a persistent cache and can be placed on an SSD attached to each database node. For queries that hit data in the cache the execution time will be improved substantially as the materialization of the summed volumes dataset will not be necessary. Each query will be evaluated by looking up only 8 values in the cached dataset. This approach will also require the reevaluation of the density threshold for choosing between I/O streaming and summed volumes. Materializing

the summed volumes dataset to stable storage will incur higher I/O costs, but subsequent queries to the same data will not have to recompute the summed volumes dataset.

We plan to evaluate the effects of parallelizing parts of the execution of the I/O streaming and summed volumes methods at a finer level. For I/O streaming, when a database atom is routed to multiple queries the computation of partial sums is independent across the different queries and could be done in parallel. This could improve the scalability of I/O streaming for large numbers of points with overlapping kernels. For summed volumes, the intermediate summed volumes dataset could be generated using multiple threads. However, it is important to make sure that at every stage in this process the data below and to the left is already processed. A wavefront access pattern ensures that this is the case and is also amenable to parallelization [27].

At present, a stored procedure or a user-defined function is implemented in the database for each filtered quantity of interest. Even though this allows us to finely tune the execution of each procedure, it requires substantial implementation effort. We plan to design and develop declarative user interfaces that will allow users to specify different filter functions that will be compiled and executed automatically by the query processing framework.

The implementation of filtering capabilities increases the utility of archived numerical simulation datasets. Filtering capabilities are invaluable for the study of LES sub-grid scale modeling. They allow for new types of analyses to be performed on the data. Users with modest computing capabilities can execute large filtering queries that are data-intensive, because the evaluation is done on the database cluster transparently to the user.

Acknowledgment

The authors would like to thank the Turbulence Database Group at Johns Hopkins University for their insightful comments and suggestions, as well as providing us with potential usage patterns and applications of spatial filtering. This work is supported in part by the National Science Foundation under Grants CMMI-0941530, CCF-0937810, and OCI-108849.

References

[1] A. S. Szalay, G. Bell, J. Vandenberg, A. Wonders, R. Burns, D. Fay, J. Heasley, T. Hey, M. Nieto-Santisteban, A. Thakar, C. van Ingen, and R. Wilton, "GrayWulf: Scalable Clustered Architecture for Data Intensive Computing," *HICSS*, 2009.

[2] E. Perlman, R. Burns, Y. Li, and C. Meneveau, "Data Exploration of Turbulence Simulations Using a Database Cluster," in *Supercomputing*, 2007.

[3] S. Pope, *Turbulent Flows*. Cambridge University Press, 2000.

[4] K. Kanov, E. Perlman, R. Burns, Y. Ahmad, and A. Szalay, "I/O Streaming Evaluation of Batch Queries for Data-intensive Computational Turbulence," in *Supercomputing*, 2011.

[5] F. C. Crow, "Summed-area Tables for Texture Mapping," in *SIGGRAPH*, 1984.

[6] H. Yu, K. Kanov, E. Perlman, J. Graham, E. Frederix, R. Burns, A. Szalay, G. Eyink, and C. Meneveau, "Studying Lagrangian Dynamics of Turbulence Using On-demand Fluid Particle Tracking in a Public Turbulence Database," *Journal of Turbulence*, vol. 13, no. 12, pp. 1–29, 2012.

[7] A. Leonard, "Energy Cascade in Large-Eddy Simulations of Turbulent Fluid Flows," *Advances in Geophysics*, vol. 18, p. A237, 1974.

[8] C. Meneveau and J. Katz, "Scale-Invariance and Turbulence Models for Large-Eddy Simulation," *Annual Review of Fluid Mechanics*, vol. 32, no. 1, pp. 1–32, 2000.

[9] The JHU Turbulence Database Cluster. [Online]. Available: http://turbulence.pha.jhu.edu/

[10] T. Lund, "The use of Explicit Filters in Large Eddy Simulation," *Computers & Mathematics with Applications*, vol. 46, no. 4, pp. 603–616, 2003.

[11] J. C. Del Álamo, J. Jiménez, P. Zandonade, and R. D. Moser, "Self-similar Vortex Clusters in the Turbulent Logarithmic Region," *Journal of Fluid Mechanics*, vol. 561, pp. 329–358, Aug. 2006.

[12] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., 2006.

[13] T. Randen and J. Husoy, "Filtering for texture classification: a comparative study," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, no. 4, pp. 291 –310, Apr. 1999.

[14] J. Ebling and G. Scheuermann, "Clifford Fourier Transform on Vector Fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 4, pp. 469–479, Jul. 2005.

[15] O. Fialka and M. Cadik, "FFT and Convolution Performance in Image Filtering on GPU," in *InfoVis*, 2006.

[16] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz, "Distributed Processing of Very Large Datasets with DataCutter," *Parallel Comput.*, vol. 27, no. 11, pp. 1457–1478, Oct. 2001.

[17] B. Allcock, I. Foster, V. Nefedova, A. Chervenak, E. Deelman, C. Kesselman, J. Lee, A. Sim, A. Shoshani, B. Drach, and D. Williams, "High-performance Remote Access to Climate Simulation Data: a Challenge Problem for Data Grid Technologies," in *Supercomputing*, 2001.

[18] Climate Data Analysis Tools. [Online]. Available: http://www2-pcmdi.llnl.gov/cdat

[19] L. A. Freitag and R. M. Loy, "Adaptive, Multiresolution Visualization of Large Data Sets using a Distributed MemoryOctree," in *Supercomputing*, 1999.

[20] K. liu Ma, "Parallel rendering of 3D AMR data on the SGI/Cray T3E," in *Frontiers*, 1999.

[21] T. W. Crockett and T. Orloff, "A MIMD Rendering Algorithm for Distributed Memory Architectures," in *PRS*, 1993.

[22] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi, "Passion: Optimized I/O for Parallel Applications," *Computer*, vol. 29, no. 6, pp. 70–78, Jun. 1996.

[23] A. P. Marathe and K. Salem, "Query Processing Techniques for Arrays," in *SIGMOD*, 1999.

[24] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *SIGMOD*, 2010.

[25] A. Deshpande and S. Madden, "MauveDB: Supporting Model-based User Views in Database Systems," in *SIGMOD*, 2006.

[26] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez, "The datapath system: a data-centric analytic processing engine for large data warehouses," in *SIGMOD*, 2010.

[27] L. Yi, C. Moretti, S. Emrich, K. Judd, and D. Thain, "Harnessing Parallelism in Multicore Clusters with the All-pairs and Wavefront Abstractions," in *High Performance Distributed Computing*, 2009.