Data Exploration of Turbulence Simulations using a Database Cluster

Eric Perlman, Randal Burns
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
{eric,randal}@cs.jhu.edu

Yi Li, Charles Meneveau
Department of Mechanical Engineering
Johns Hopkins University
Baltimore, MD 21218
{yili,meneveau}@jhu.edu

ABSTRACT

We describe a new environment for the exploration of turbulent flows that uses a cluster of databases to store complete histories of Direct Numerical Simulation (DNS) results. This allows for spatial and temporal exploration of high-resolution data that were traditionally too large to store and too computationally expensive to produce on demand. We perform analysis of these data directly on the databases nodes, which minimizes the volume of network traffic. The low network demands enable us to provide public access to this experimental platform and its datasets through Web services. This paper details the system design and implementation. Specifically, we focus on hierarchical spatial indexing, cache-sensitive spatial scheduling of batch workloads, localizing computation through data partitioning, and load balancing techniques that minimize data movement. We provide real examples of how scientists use the system to perform high-resolution turbulence research from standard desktop computing environments.

1. INTRODUCTION

Traditional Direct Numerical Simulation (DNS) of turbulent flows provides poor support for data exploration, particularly in the temporal dimension. DNS uses cluster computers to solve discretized fluid dynamical equations and, thus, evaluates velocity, pressure, or transported properties, such as temperature and pollutant concentrations, in 4-d space/time. However, such simulations only manifest a few time steps at any time, due to storage limitations within the cluster. In DNS, complex multi-scale turbulent fields must be described using large number of grid points whose large size severely limits the experiments that can be run during DNS. Typically, experiments look at each time step only a few times before discarding it.

We present an archival approach to turbulence experiments that allows for data exploration of a DNS. To achieve this goal, we store the entire space-time history generated by the DNS in a cluster of databases. By retaining a complete history, the database cluster also persistently stores the computational effort of the simulation cluster. This radically alters the computational demands of turbulence experiments that explore a DNS. Using the database, an experiment computes simple operations against a large amount of data.

Good temporal support makes several important classes of turbulence experiments very efficient. Moving from time-step to time-step involves accessing data only; no operations associated with the dynamic time advancement of the DNS (e.g. Fourier transforms in the case of pseudo-spectral methods) need to be computed. This allows turbulence experiments to look at large time spans and iterate back and forth through time. We present several exemplars. To study the pre-history of turbulent structures, experiments track particles forward and backward in time, identifying turbulent structures as they evolve and then tracking them backwards to identify the pre-conditions that led to their evolution. We also study more traditional velocity gradient statistics that examine field differences across temporal and spatial spans.

A Web services interface on top of the database cluster permits experiments to be run across networks and makes the database publicly available. We also offer Fortran and MATLAB® interfaces layered above Web services so that scientists can continue to use the tools with which they are most familiar. Powerful data exploration experiments can be managed from the least capable computers, e.g. from a laptop in a café. Public access to our database brings high resolution computational turbulence data to user communities that lacked adequate computation and storage resources previously. Other efforts to share the output of a large DNS [5, 28] either (1) allow data to be downloaded and the user must provide the computational resources or (2) provide user accounts to the systems on which data are stored. While such approaches preserve the computational effort, they do not support users with limited infrastructure or those who may not wish to learn how to run code in a different environment.

A database cluster for DNS turbulence experiments requires specialized techniques for data and workload management. Our implementation includes:

1. cache-sensitive spatial scheduling for batch workloads;
2. overlapped data partitioning that localizes computation to a single node;
3. a 3-level, hierarchical, spatial index based on a Morton-order, space-filling curve that supports data distribution across sites, B+-tree access paths at each site, and efficient I/O schedules for range queries; and,

SC07 November 10-16, 2007, Reno, Nevada, USA

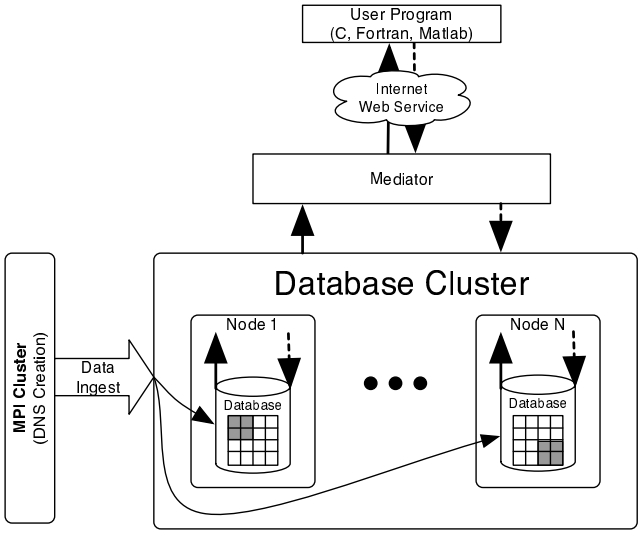


Figure 1: Architecture of the Turbulence Database Cluster

4. dynamic load balancing techniques that move the minimum amount of data between sites.

Our experimental evaluation shows that storing complete space-time histories of DNS makes for an efficient computational environment for applications that need to perform data exploration or when multiple experiments reuse the same DNS solution. A study of the individual design elements in our system confirm the importance of a spatial scheduling and data partitioning for batch workloads.

2. ARCHITECTURE

A cluster of homogeneous database nodes forms the core of our environment (Figure 1). Database nodes both store the space-time history of a turbulence simulation and provide a parallel computing platform for experiments against that simulation. Data are partitioned spatially (indicated by shaded regions) and placed on different nodes. Each database node runs Microsoft Windows 2003 on two dual core AMD Opteron processors with 4GB of memory. The system provides parallel computation. The experimental workload consists of relatively straightforward computations against large amounts of data, which makes I/O parallelism the bottleneck and our focus. Each database stripes data across 12 sets of mirrored 400 GB SATA disk drives. When accessing data, we read entire stripes of data to maximize parallelism.

An MPI compute cluster runs the DNS and produces simulation results as large multi-dimensional arrays within files. We redistribute these data through file transfer to the database nodes that ingest the data. Ingested data are reorganized within our cluster through database-to-database copy.

Our system includes a mediator which acts as a gateway for users and applications. The mediator has knowledge of the placement of data. It takes individual requests and routes them to a node that has the data needed to service that request. Similarly, it takes batch requests, breaks the batch into sub-batches specific to each database and routes the sub-batches accordingly. The mediator implements a Web service as the basic interface into the database cluster. Users and applications discover the interface through the Web Services Description Language (WSDL). Requests and results are transmitted between the mediator and its clients through the Simple Object Access Protocol (SOAP).

"Move the program to the data" is a guiding principle in our system (and in the design of scientific databases in general [23]). Thus, we rely heavily on user-defined functions (UDFs) in the SQL Server 2005 using the common-language runtime (CLR) [20]. The CLR integration allows UDFs programmed in C# or C++ to be computed inside the database. By implementing the data-intensive portions of turbulence experiments inside the database, we minimize the number of interactions between clients and the mediator using the Web service. Examples include computing interpolation functions over regions of space and time, evaluating spatial and temporal derivatives, performing local filter operations, and accumulating statistics over a subregion of the data.

2.1 User Interfaces

We provide a computing interface that can be adopted immediately and enhances existing simulations (in terms of both speed and quality of data). A traditional model of computing, used in many DNS simulations, employs a series of iterative loops that go through each time step and compute new values from the underlying data. To allow these simulations to be modified with little effort, we abstract common tasks, which would typically be located in a subroutine, with calls to functions provided by our Web service. The Web-service routines implement the functionality of the replaced code, iterating over space and/or time, inside the database. By replacing entire loops (multiple levels of loops when possible), we achieve Web service requests consisting of batch workloads.

Batch workloads amortize the overhead of Web-service requests. Each invocation of the Web service has fixed costs to set up the request. Then, each request requires data to be packaged (as XML in a SOAP object), transferred across the Internet, and unpackaged. Most of the overhead arises from latency, not from processing. The efficiencies of batch workloads lie in sending more data in each Web service request, which amortizes fixed costs, and in sending less data overall, by conducting multiple processing steps for each invocation.

The performance benefits of batch workloads come at a loss of generality. Extracting more function from the calling program results in a more specific Web-service routine, which reduces its re-usability and applicability to the broader user community.

To increase the utility of the system, we provide interfaces at multiple levels. These include low-level fundamental tasks, such as reading individual values from the database, and high-level concepts, such as entire experiments packaged in a single Web-service call. By selecting among these interfaces, users customize their application's use of our system. When choosing low-level interfaces, they may suffer some performance degradation, because their application interacts with the Web service more frequently and loses the benefits of batch processing.

Our experience indicates that a good balance can be struck by providing batch read and batch interpolate operations at the time step granularity. These perform a large

Fortran

```
REAL x(dim),y(dim),z(dim)  ! particle coordinates
REAL vx(dim),y(dim),z(dim) ! velocity components
REAL dt = 0.0035 ! time delta
REAL nOrder = 6  ! 6th-order Lagrangian interpolation
DO timestep = 1,1000,1
  ! GetVelocity is a wrapper routine for a
  ! gSoap call to the like-named Web service
  CALL GetVelocity(timestep,dim,nOrder,x,y,z,vx,vy,vz)
  DO p = 1,dim,1
    ! Perform local computation with velocity values
    x(p) = x(p) + vx(p)*dt
    y(p) = y(p) + yx(p)*dt
    z(p) = z(p) + zx(p)*dt
    ! ... do something with new particle locations ...
  END DO
END DO
```

MATLAB®

```
dt = 0.0035   % time delta
nOrder = 6    % 6th-order Lagrangian interpolation
% Create dim points (x,y,z) with three random values
points = rand(dim,3) * 2 * pi
% Iterate over multiple time steps
for timestep = 1:1000
  % GetVelocty is a matrix-friendly wrapper routine
  % to the like-named Web service
  v = GetVelocity(timestep, nOrder, points)
  for p = 1:dim
    points(p,1) = points(p,1) + v(p,1)* dt
    points(p,2) = points(p,2) + v(p,2)* dt
    points(p,3) = points(p,3) + v(p,3)* dt
    % ... do something with new particle locations ...
  end
end
```

Figure 2: Fortran and MATLAB code snippets that use the Turbulence Web service

number of relatively low-level operations in a single Web-service request. Thus, they are highly I/O parallel and still general enough to allow for a great deal of customization. For example, particle tracking experiments advance particles at each time step based on the interpolated velocity at each particle's location. One approach to programming this experiment would put the entire function within the database and have the database advance the particles and return their new locations at each time step. A more general—and nearly as efficient—alternative has the application request the velocity field at each particle's location in a batch. Then, the user application advances their locations. The user can now alter the way the particles interact with the velocity field, e.g. change their mass. This requires the user's application to receive and send a new batch to the database cluster at each time step. However, this cost is amortized over tens or hundreds of thousands of particles.

In addition to preserving the programming model, we also support the computational tools most commonly used by the turbulence community. These include programmatic methods to invoke Web services from both MATLAB and Fortran (Figure 2). For C and Fortran, we provide a wrapper interface to the gSOAP [26] library, which is linked at compile time to the user program. Recent releases of MATLAB offer integrated Web-services support, although we found the need to provide a wrapper to those calls to massage the data types into the appropriate useful MATLAB types. In addition, due to the Web-service model, our routines can be called easily using standard library routines with most modern languages, including Java, C#, and Python. For example, we have a simple visualization utility written in Python, which queries the database and graphically displays particle movements over time (Figure 6(a)).

2.2 Data Generation and Ingest

Our system stores a time series of three-dimensional structured grid data. For the experiments that we present, we generate and store the velocity and pressure field of a 1024^3 DNS of stationary hydrodynamic turbulent flow based on a solution to the Navier-Stokes equations. We perform this simulation on a high performance linux cluster, also located at Johns Hopkins. These DNS results are stored as 256 files, each representing one eighth of the data, split in the Z-dimension. We copy these files to the database cluster through standard file transfer mechanisms over gigabit Ethernet. Once located on the cluster, we process the large volumes of data into smaller units for insertion into the database. To avoid the significant penalty of disk seeks, we stride through the results keeping a small portion ($1024 \times 1024 \times 72$, 1.2 GB) in memory. We arrange this data into small chunks that we place into a subset of database nodes using our pre-determined initial data allocation strategy (see Section 3.1).

We configure the database to optimize for data loading. We disable transactions and statistics gathering on the table while loading data. We utilize an SQL Server .NET BulkCopy interface. This allows us to write middle layer code that parses the simulation output into appropriate data cubes for the database, while allowing the database to bypass the standard time consuming consistency checks.

We sacrifice some aggregate ingest performance in order to reduce implementation complexity and support incremental loading: ingesting data into the databases concurrent with data generation on the MPI cluster. The multiple steps of transferring files, partitioning files, and loading data fragments reduces data rates. However, the ability to dynamically load partial data series into the database will allow experimentation on the data before a commitment is made to generate the rest of the data. This will be particularly useful when we begin exploring data sets in the 2048^3-4096^3 range. One of our goals is to require minimal modifications to the existing codes that perform DNS. To this end, the files we process are large Fortran-(or C-)style arrays. We use a number of disk seeks to divide these into appropriate block sizes.

Our measured throughput rate for data insertion is close to 12MB/s on each node. This task can be run in parallel on the individual nodes with little interference, giving us a total combined throughput of roughly 50MB/s when running data ingest threads on four nodes.

3. DATA ORGANIZATION AND WORKLOAD MANAGEMENT

A unified, spatial addressing scheme underlies all indexing, access paths, and location queries. We assign Morton-order (bit-interleaved) addresses to the voxel space of the original DNS. Multiple abstract and concrete data structures employ the same addressing scheme for multiple purposes.

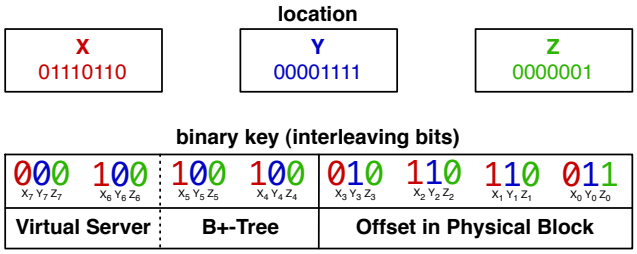


Figure 3: Addressing and partitioning data using Morton-order curves

These include:

1. *data distribution*: assigning cubic partitions of space along Morton-order boundaries to cluster nodes;
2. *data partitioning*: dividing regions of space into fixed-sized storage blocks on which we conduct I/O;
3. *indexing workload*: clustering workload into spatially related requests using a quad-tree;
4. *processing spatial range queries*: identifying the minimum cost I/O schedule that covers the requested space; and,
5. *scheduling workload batches:* identifying spatially-related point queries to be co-scheduled.

3.1 Indexing and Partitioning

In distributing data to cluster nodes, we employ two-levels of data partitioning. We divide the data space into a minimum size element or atom of size 64^3 voxels equal to 8 MB of data. The atom represents the fundamental unit of I/O. We choose its size in order to balance memory and disk performance—too large an atom pollutes the cache and results in large transfer times from disk, whereas too small an atom reduces I/O efficiency because data transfers are too short to amortize disk seek times. From the atoms, we build an index using a Morton-order, space-filling curve that logically partitions the space into cubes of side 2^k for $k = 0, \ldots, \log(n)$ for n atoms.

We logically divide the address space into a three-level hierarchy, segmenting addresses into high-, middle-, and low-order bits (Figure 3). These differentiate the roles of different portions of the address space. High-order bits direct the partitions of the original data space to servers in the database cluster. Middle-order bits are used to index the atoms stored at each site. The low-order bits are used to conduct range queries on a Morton-order representation of the atom in memory. The middle- and lower-order bits help realize spatial clustering and locality, but do so on different data structures: a database B+-tree-index and a space-filling curve.

Initially, we distribute spatially-contiguous collections (partitions) of these atoms to the nodes in our cluster. We create roughly eight to sixteen partitions per node and use a virtual processor approach to assign partitions to nodes [19], mapping each partition to a virtual processor and then mapping multiple virtual processors to physical cluster nodes based on expected load and storage capacity. For example, we split a 1024^3 time-step into 4096 64^3 atoms that we distribute to four nodes as 16 256^3 partitions. Virtual processors allow us to add or remove cluster nodes, because it gives us lots of (relatively) small partitions to move among sites. It also allows us to handle clusters for which the number of nodes is not a power of two, reducing the skew in storage used at each site. Virtual processors would also balance loads across heterogeneous computers. At present, our cluster is homogeneous.

We also place short spans of consecutive time steps on the same server to optimize for jobs that run across multiple time slices. For example, in particle tracking, a particle in the same region of space stays on the same server for computation at multiple time steps. Thus, we avoid packaging the request and moving it to a new server. The data storage design revolves around the spatial and temporal properties of the data.

The initial placement is very coarse-grained. At runtime, we will update the partitioning of data based on workload statistics and redistribute the new (smaller or larger) partitions among cluster nodes (see Section 3.4). Repartitioning may involve "shifting" the boundary between high and middle bits of the address space, adding three bits to the virtual server address when dividing a partition into 8 smaller partitions and dropping bits when merging smaller partitions. This process is similar to adding and removing bits from a extendible-hashing index as files grow and shrink [7].

The middle-order bits create a spatial index on the data atoms stored at each site. Using these bits as the search key in a B+-tree indexed file storing the atoms ensures that atoms that are contiguous in Morton order are contiguous in both the B+-tree index and in the file [21]. This data structure supports range queries to cubic regions of space aligned to Morton-order boundaries in a single disk access for the atoms (and I/O in the index). In fact, the compact spatial organization of the Morton order results in low I/O complexity for all convex regions [18].

The low-order bits address the data for an individual voxel inside the atom. Once an atom is loaded into memory, these bits translate between simulation space and memory addresses. While these bits also form a compact space-filling curve that localizes accesses to spatial regions, the performance benefits are minimal because data are in memory at this point and memory supports random access.

3.2 Edge Replication

For performance reasons, it is desirable to localize each computation to a single database node. Operations that span nodes need to ship partial results back and forth among databases, adding latency to job completion and complexity to the implementation of that function.

We use a small amount of replication to localize computations to each node. Any computation that utilizes a kernel of 8^3 or less will be fully contained in a single atom. We guarantee this by adding a border of length 4 to each edge of the cube—storing 72^3 data point values for a 64^3 atom. Figure 4 shows edge replication in two dimensions. The space is periodic in all dimensions—it wraps around—so that partitions at the left edge replicate data on the right edge, etc. Each atom overlaps its neighbors by 4 voxels on each side and corner. We chose a border of size 4 because it supports the kernels of computation for 6th and 8th order Lagrangian interpolation functions—a primitive operation used in many of our experiments.

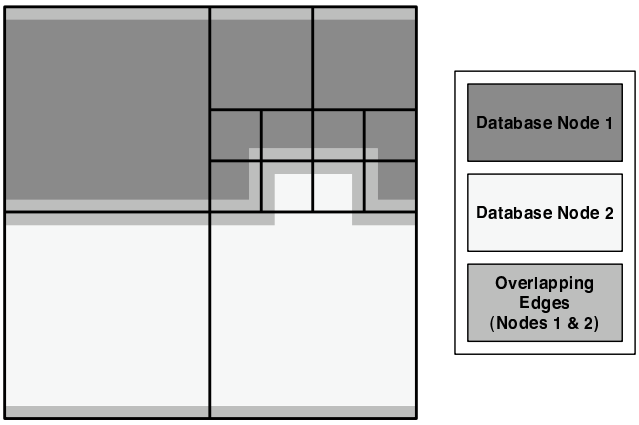


Figure 4: Partitioning with edge replication of a single time-step across two database nodes

While the extra space necessary is substantial (43%), we prefer the storage overhead to the complexity of performing fundamental operations, such as interpolation, across multiple nodes and the I/O costs of multiple block accesses. Storage overhead decreases when increasing the atom size, e.g. 20% for 128^3, which should be weighed against disk I/O and memory concerns when selecting the atom.

We could reduce the overhead of edge replication by storing it only on partition boundaries, rather than on the atoms themselves. However, this would make it more complex and less efficient to move partitions when the cluster configuration changes or as part of dynamic load balancing (see Section 3.4). When changing a partition, the edge replication would need to be adjusted, which would involve querying the replicated region and adding it to new partition. By replicating edges on atoms, no action needs to be taken when partitions change and the replicated data are stored as part of the atom.

3.3 Spatial Scheduling

The bulk of our workload consists of batch requests with high particle densities, i.e. a large number of particles in the same cubic region. Points near each other in space generally require data within the same atom(s). Recall that many computations, such as Lagrangian interpolation, tend to access cubic regions of space.

Based on this observation, we reduce I/O costs significantly through the spatial scheduling of batch computations. This makes the order in which we evaluate computations conform to a data access pattern. We associate each computation in a batch with a point in space, generally the point at which a query is defined or the center of a region query. We sort and then schedule the points in Morton order, using the same space-filling curve that addresses the underlying data. In general, this results in each database atom being loaded into memory once. All points centered in an atom are scheduled together and points that use this atom as part of their kernel of computation should be scheduled either closely before of after points within the atom, due to the localization properties of the Morton-order curve Atoms may be read from disk more than once when they are referenced by two requests that are separated by enough time that enough intervening atoms have been loaded into cache to evict the originally loaded atom. This should happen rarely and only for regions of space separated by Morton-order boundaries at the highest resolutions.

The multiple resolutions of the Morton-order curve map well onto the various underlying caches. At the lowest levels, all particles within an 8^3 reference data capable of fitting entirely in an L1 cache and 32^3 in an L2 cache.

3.4 Load Balancing

Turbulence experiments present highly-skewed workloads that mandate dynamic load balancing. DNS data are regular: a uniform-dense voxel space at each time step. However, turbulent structures are irregularly distributed in the data space. These structures govern the distribution of workload within the data space. For example, particles in a tracking experiment may cluster around a vortex. The turbulent structure and, thus, the workload vary across multiple experiments against the same output. The data needs to be partitioned and distributed across nodes without knowledge of the spatial distribution of workload. We cannot replicate large portions of the data because storage space ultimately limits the size of the simulations and the utility of the database. In all, this leads to large load imbalances, which reveal themselves at runtime.

We present a dynamic load management system that adapts to the highly skewed workloads of turbulence experiments. We take advantage of correlation within regular decompositions, using a Morton-order organization of the data space and an oct-tree decomposition of the workload requests, to identify regions with the highest workload to data-size ratio. By moving these regions from overloaded nodes to underloaded nodes, we achieve the minimal execution time schedule based on moving both data and workload that also accounts for the costs of moving data between sites.

We describe our load-balancing strategy through an example in a two-dimensional, reduced space. Figure 5 shows a representation of a batch of point queries in the data space. On this batch of points, we construct a region oct-tree [21] (quad-tree in three dimensions) so that each region has a maximum number of point queries. The region oct-tree provides a regular decomposition of space along the same power of two boundaries on which the Morton-order, space-filling curve creates logical partitions. Thus, the number of points in each region represents the amount of workload in the underlying partition. For each region in the quad-tree, we compute the workload density: the number of points over the volume of the region.

Using the workload density, we balance load by moving the minimum amount of data. The goal of load balancing is to achieve an identical completion time at each node with a minimum amount of load balancing overhead from data movement. Given a workload batch, the mediator constructs the oct-tree and estimates completion time on each node by comparing the workload distribution to the data distribution. It then chooses the densest data partitions in the entire space on overloaded nodes and instructs the overloaded nodes to copy these to underloaded nodes. The mediator then routes requests to these dense regions to their new locations. At the end of the batch, the nodes that received workload can discard the copied data, returning the cluster to its original storage configuration. Nodes could retain (cache) copied data to serve future requests to the

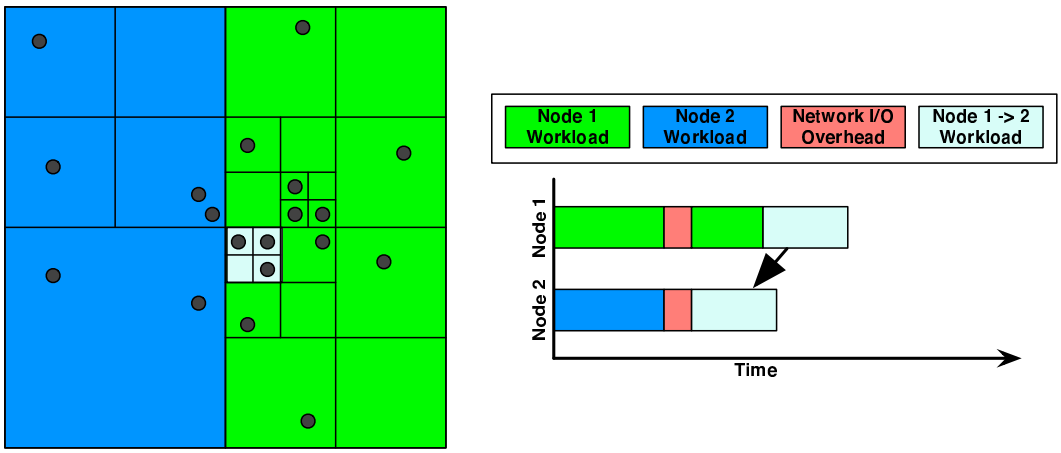


Figure 5: Example of workload being transferred between nodes to balance load

same time-step without copying the data again. However, this would require the mediator to track copies of partitions over time. Given that we batch workload within each time-step, we expect little reuse of the same time-step and, thus, have not yet found it necessary implement caching.

A perfect load balance may be achieved for two nodes by selecting the partition granularity and fine-grained request routing. Each partition that we copy to balance load contains a discrete amount of workload that may not exactly match the load difference between two nodes, leaving a residual load imbalance even after moving the partition. We address this effect in two ways. First, the mediator selects partitions at the appropriate granularity. It may break a larger partition into 8 smaller partitions and only copy some of those smaller partitions. Second, once a partition has been copied to a new site, it is replicated at two sites and the mediator can distribute the individual point queries across the two sites. Thus, our policy is to replicate a single partition—the smallest, densest partition that exceeds the load difference between two sites. Each separate partition has significant cost to ingest and index at the new site. Thus, we prefer to copy a single, larger partition to multiple, smaller partitions. By choosing slightly more load than needed, we can use fine-grained request routing to the two copies to balance load exactly.

As described, the system balances completion time perfectly with minimum data movement between two nodes, but may not do so among many nodes. With more nodes, all nodes need to be balanced to an average value, which changes depending upon which data are copied. This is a more complex combinatorial problem. In practice, this has not been a big concern. The highly-skewed nature of turbulence workloads results in one (or a few) nodes that are highly overloaded, which distribute workload to many underloaded nodes. Thus, pairwise exchanges between nodes realize very good solutions for all nodes.

4. APPLICATIONS AND USAGE

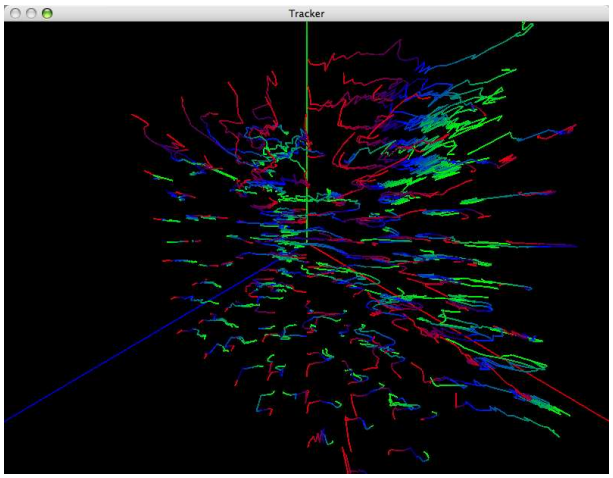
Storing the entire space-time histories of DNS simulations dramatically improves the performance of many classes of turbulence studies while ensuring that the experiments are repeatable and verifiable. We present two archetypal applications that use our turbulence database: a particle tracking experiment that iterates back and forth through time, examining each time-step multiple times, and a turbulent statistics application that compares large volumes of data across large temporal spans.

4.1 The Pre-History of Turbulent Structure

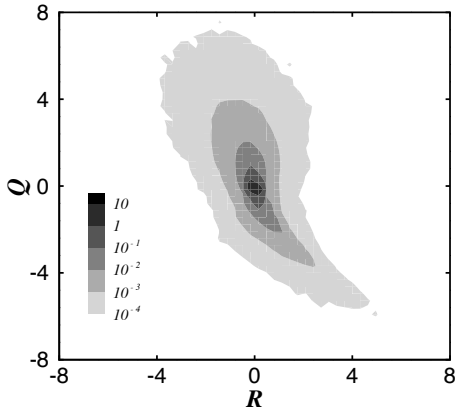
Particle tracking experiments can be used to explore turbulent structures. These experiments place particles in a turbulent vortex structure and then calculate the subsequent or previous time-step and update the location of these particles [29]. The particles act as tracers to follow a turbulent structure according to an evolution equation. Figure 6(a) shows the movement of particles in a turbulent velocity field computed on the database cluster. In order to advance the particle position, the velocity vector at the particle position must be found from appropriate interpolation of the grid-based velocity available in the database. For this purpose, the fundamental operation performed on the database is a nth order interpolation for every particle. (Typically 6th- or 8th-order Lagrange polynomials are used.)

To give the greatest flexibility to the user (See Section 2.1), users may choose to apply the interpolated field values to advance particle locations on their local computer. Gains from massive parallelism arise from tracking multiple particles at a time. For instance, a query may be for 10^4–10^7 interpolated velocities to advance large ensembles of particles simultaneously. Users may instead use an implementation coded inside the database using the Adams-Bashforth second order scheme for the time advancement of the particle positions. We examine the consequences arising from this trade-off in Section 5.2.

The turbulence database brings several benefits to particle tracking. First, it eases the exploration of data, allowing multiple iterations through time without re-computing velocity fields. Parameters to the point advancement routines can be modified and re-run on the full resolution data in near real time. Also, the interpolation queries access a large volume of data (n^3 for an nth order polynomial). By performing interpolation in the database, we access lots of data to extract a single quantity (the velocity at a point). Transferring the query result only, rather than the source data, exemplifies the network benefits of "moving the computation to the data."



(a) Visualizing a Particle Tracking Experiment



(b) $R - Q$ plot generated from velocity gradients

Figure 6: Experiments from the DNS Turbulence Database

4.2 Evaluating Velocity Gradient Statistics

The velocity gradient tensor is the spatial derivative of the velocity vector in each spatial direction. It provides critical information about the local structure of turbulence and, therefore, has received much interest in the turbulence research literature lately (see Li et al. [16] and references therein). The velocity gradient at an arbitrary location is evaluated with Lagrangian interpolation on the stored velocity data (i.e., no velocity gradient data need to be stored). The interpolation polynomials allow easy evaluations of gradients and can be evaluated as efficiently as the interpolated velocities. We then compute the joint probability density function of two coordinate-transformation-invariant quantities formed from the components of these velocity gradients (Figure 6(b)).

Such a density function conveys a large amount of information about the small-scale structure of turbulence. It describes the frequency of observing combinations of the (so-called) R and Q parameters. For instance, when one of the parameter (R) is negative and the other (Q) is positive, the vorticity (fluid rotation) of the flow region is being stretched by the straining of the flow. Another quadrant of the joint distribution, e.g. both parameters are positive, is associated with the less probable processes of vortex compression. (These states are more unstable since the vortices may "buckle"). Therefore, it is of interest to evaluate the joint probability density function (PDF) to describe intrinsic properties of the flow.

4.3 Computational Restrictions

Storing space-time histories in physical space places limitations on the computations that the database supports. A cluster of databases offers a poor environment in which to compute all space Fourier transforms. They are memory limited and have slow networks, relative to the computational clusters on which the data were originally generated.

But much interesting science is conducted in Fourier (wave-number) space, not physical space. For example, many experiments perform particle tracking in filtered velocity fields. Particle tracking occurs in physical space, but filtering is most often done in Fourier space. To accomplish this, the system must be able to convert back and forth. A task for a computational cluster, not our turbulence database.

Currently, our turbulence group is working on conducting filtered-particle tracking experiments in locally filtered velocity fields, computing kernel-based transforms over spatial sub-regions. Initial results indicate that this approach introduces small discrepancies with all-space Fourier methods, sacrificing some degree of fidelity, but preserves experimental outcomes over moderate temporal and spatial ranges.

We hold that the merits of storing space-time histories will drive the turbulence community to continue developing techniques to conduct experiments in physical space.

5. EVALUATION

Our performance evaluation focuses on isolating the benefits of individual design decisions. Specifically, we examine the effects of spatial scheduling, different batch workload strategies, and Web-services overhead.

The aggregate performance of the turbulence database compares well with conducting the same experiments on the DNS computational cluster. Experiments that track a random workload (the worst workload for our database) of 10,000 particles over 50 time-steps takes less than 20 minutes. Generating the data underlying this experiment takes an order of magnitude more time. We do not present a direct comparison between computing turbulence experiments on a database cluster with traditional computing models for many reasons. The experiments run on different hardware—there are over 40 nodes in the compute cluster compared with our 4 database nodes. The database cluster does incur a one-time cost for data generation. However, that cost is amortized over the many experiments run against that data. Finally, comparative results are sensitive to parameters, such as number of particles and size of the DNS. We note that when using the turbulence database, experiments scale with the number of queries (e.g., particles being tracked) regardless of the size of the DNS. Whereas in traditional computing models, the experiment scales with both the number of queries and the size of the DNS, and the size of the DNS is the dominant factor.

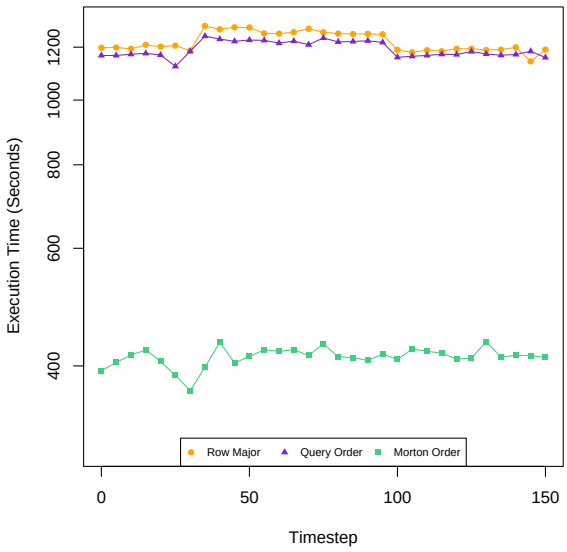


Figure 7: A 10,000 particle trace executed against the database using Morton-order scheduling, Row-major scheduling and the original order of particles in the trace.

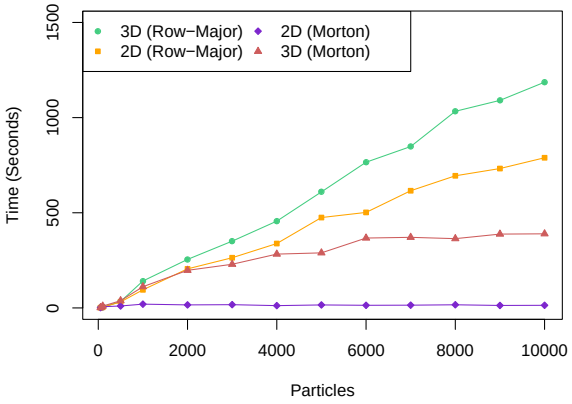


Figure 8: Execution times for a random points spread across a 1024^2 plane or 1024^3 volume, using Row-major and Morton-order scheduling.

5.1 Scheduling

In these experiments, we isolate the benefits of spatial scheduling and show its necessity in our environment. We compare three scheduling disciplines:

- No ordering: The system executes the workload in the order it was received.
- Row-(or Column-)Major: Particles are ordered based on their indexes in **C** or **Fortran**-style three-dimensional array.
- Morton order: Particles are ordered based on a Morton-order, space-filling curve as described in Section 3.3.

We run several workloads on these schedulers: a trace from a prior experimental run and random workloads across both a plane and the entire cubic space. Given our data placement (Section 3.1), we know that striding through the database in a Row-(or Column-)major fashion can lead to memory thrashing as an atom is loaded into memory repeatedly.

Figure 7 shows that spatial scheduling provides a speedup of three times or more on a trace that records the location of particles over 150 time-steps during a particle tracking experiment. The close correlation between the execution time with no ordering and that of row-major ordering is an artifact of the initial computation that generated the trace. This trace starts as a large number of points dispersed evenly across the space and, for the early time steps, delivers the requests in roughly row-major order.

We also measure the benefits of spatial scheduling for random workloads (Figure 8). This allows us to consider the full space 1024^3 with reasonable request densities and to examine how the benefits of scheduling scales with the density of particles. For the 2D tests, the Y-dimension was fixed and both the X and Z values were populated with random values. For the 3D tests, all three dimensions were chosen at random. Both cases show the "clustering" ability of Morton-order scheduling. Unlike the Row-major ordering, the Morton order accesses each atom of underlying storage exactly once over the course of the execution. We do not include results of running these workloads in the order requests are presented (i.e., no ordering): the time taken exceeds any reasonable bound as the probability of even two points collocating in a region of space is small.

These results show how very large simulations can be run efficiently relative to smaller ones, which have much higher average costs for each operation. For the smaller workloads in which $n <= 4096$ (the number of underlying data atoms), most particles result in the need to read the atom into memory. For $n >= 4096$, we are guaranteed that multiple particles will sometimes fall into the same underlying region of space. When this occurs, the second, third, etc. operation may be executed with computation alone, requiring no time-consuming I/O. This also accounts for the difference in the 2- and 3-dimensional cases, because the 2-dimensional experiment restricts data to a planar subset of all data, which reside on 256 underlying data atoms. This also increases the density of particles within each of those atoms. The Morton-order schedule ensures that these particles in the same region of space are grouped into the same physical I/O, regardless of the initial query order. This can clearly be seen from the flattening of the lines for larger numbers of particles.

5.2 Work-flow Trade-offs

We compare three different disciplines for running the particle tracking code of Fig 2. We place the iteration over time at three different locations: on the client, on the cluster mediator, and within the database itself. We perform this simulation using 1,000,000 particles spaced over the 1024^3 resolution data.

Figure 9 shows the additional cost of performing the iteration farther away from the database at the mediator and client respectively. This is primarily from the latency associated with the return of one result set and the input to the

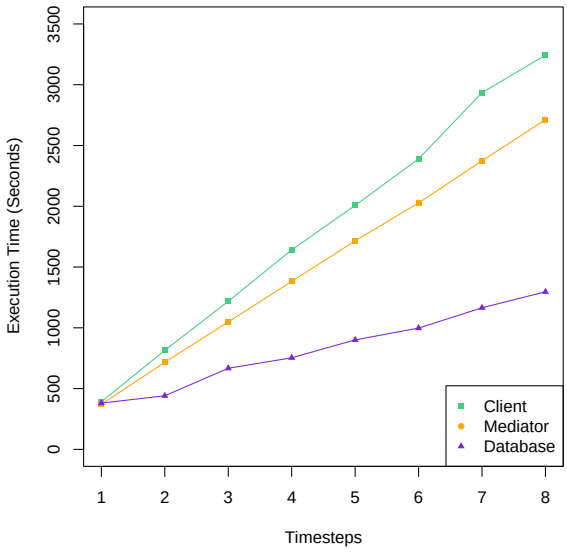


Figure 9: Execution times for a particle trace of 1,000,000 particles run iteratively across an increasing number of time-steps.

next batch query. When inside the database, these values are stored into a local array and then used as soon as the last particle in a given time-step is complete. While noticeable, our experiments show that the split of computation between the database and client is a feasible model for performing turbulence calculations, offering an acceptable trade-off in performance for the additional flexibility in computational parameters.

We find that performing the computations on the client instead of on the mediator layer incurs an additional cost of 15%–20%, increasing slightly for each additional iteration. However, if we run the entire computation directly on the database, we see a 40%–50% improvement. This represents the ideal situation in which workload does not move between servers.

These results validate our previous assertion that batch execution of all queries within a single time-step strikes a reasonable balance between efficiency and generality. Having the client perform the iteration degrades performance substantially, but not catastrophically. At the same time, it allows the user to customize an application, e.g. changing particle mass in a particle tracking experiment.

5.3 The Need for Batch Workloads: Network and Web Service Overhead

The encoding and decoding SOAP messages incur a large cost when used for scientific workloads [6]. We use a null operation function to measure the base costs associated with the transfer of SOAP messages. A number of minor changes in SOAP can lead to an overall performance improvement [1], but do not change the fundamental scaling properties of the transfer layer. Figure 10 shows the measured cost of SOAP messaging. These costs are linear with respect to the size of the message being sent, although this experiment uses a null message with no data. A million messages take

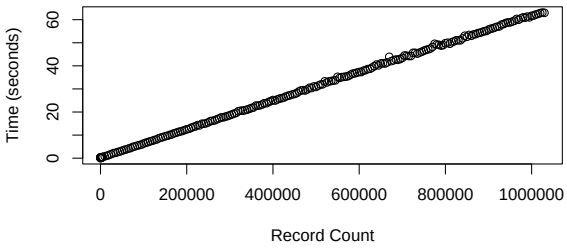


Figure 10: Overhead costs from SOAP messaging

over a minute to transfer, which is of practical concern given that we present experiments that track a million particles

As we described in Section 2.1, experiments should minimize the amount of data transferred to limit SOAP overhead by performing as much computation as possible on the databases cluster nodes. As an example, we consider the particle-tracking experiment from our evaluation of workflow trade-offs (Section 5.2). Were we to run this experiment using individual queries for each point, we would expect it to take 3000 times longer—it would take days, instead of minutes—due to the high overhead cost for each particle.

6. RELATED WORK

Other research projects have made the results of large-scale turbulence simulations publicly available. The Computational Fluid Dynamics (CFD) database provides downloads of several data sets up to 1024^3 in size at the Cineca supercomputing center [5]. Yeung maintains a 2048^3 database DNS database [28]. To our knowledge, no previous projects focus on temporal support—they give few time-steps—nor do they provide Web-services interfaces.

Many other projects have employed Web services to make scientific computing applications available over the Web. Malik et al. [17] provide Web-service access to a federation of Astronomy databases. Web-services interfaces exist to Geo-spatial databases [11] and Biological databases [10]. Kandaswamy et al. [12] describe a framework that automatically integrates Grid applications into Web services, allowing existing scientific applications to be made available over the Web. The prevalence of Web services for scientific applications has caused the community to look at performance issues, specifically limiting the negative effects of XML encoding [1, 27].

Scheduling queries and execution to maximize memory hierarchy benefits is an essential aspect of most scientific numerical simulations. Krishnamoorthy et al. [13] explore computational environments in which execution plans are memory hierarchy aware in order to realize significant performance improvements. The work focuses primarily on an algorithm for determining proper execution of a workload with a dependency graph. The Sequoia programming language [8] was designed around making memory-hierarchy dependent applications portable across multiple architectures. While our scheduling is cache-aware, requests may be scheduled in a cache-oblivious fashion with good results [9].

The issues of dynamic load balancing and batch scheduling also arise in other scientific particle simulations. Standard N-Body simulations [3, 22] exploit the spatial characteristics of the simulation space to achieve parallelism.

Using oct-trees with existing database techniques has been studied in the computational database system Weaver [24], and in a distributed setting in Octor [25], which focuses on workload decomposition with correlations that occur across nodes.

The process of further dividing our oct-tree based on workload patterns resembles the refinement in structured adaptive mesh (SAMR) applications [4, 2]. SAMRs offer a varying resolution of data across the grid. Our varying workload density is analogous to the regions of the SAMR containing higher resolution data. Most of the load balancing work in this area [14, 15] applies directly to our problem space.

7. DISCUSSION

We have introduced a new environment for computational turbulence based on storing the complete space-time histories of the solution to direct numerical simulations (DNS). The environment has the benefit of providing good support for the data exploration of turbulent flows, particularly for examining large time-spans or iterating back and forth through time. It also allows for reuse of the same DNS, which has recently become of interest to the turbulence community [28, 5].

By implementing the system as a Web-service interface to a database cluster, we support high-resolution turbulence experiments from desktop computing platforms and make both the data and the experiments available to the public. We also use the Web-service interface to support traditional computing tools, such as Fortran and MATLAB®, allowing existing experiments to use our environment with minimal changes.

We use an indexing and data organization scheme that will scale with the future growth of turbulence simulations, both in the number of discrete time-steps stored and the resolution of the data. With the addition of more nodes and larger hard disks to our existing infrastructure, we will easily be able to store and make publicly accessible the outputs of some of the larger DNS being run today (2048^3–4096^3). While these data are 8–64 times greater than what we currently store, data ingest is a one-time task and experimental performance within the database cluster depends on the amount of data being examined, not the size of the DNS.

More information on our project can be found at `http://turbulence.pha.jhu.edu/`.

Acknowledgments

We would like to thank Minping Wan for generating the DNS solution stored by our database and Shiyi Chen who oversaw this process. We also appreciate the countless contributions of the rest of the Turbulence Research Group at Johns Hopkins: Hussein Aluie, Laurent Chevillard, Gregory Eyink, Carlos Rosales, Dimitry Shapovalov, Alex Szalay, Ethan Vishniac, Yunke Yang, and Zuoli Xiao. We would like to thank Jan vandenBerg and Alainna Wonders for their support in configuring and maintaining both the compute cluster and the turbulence database cluster and Tamás Budavári for his assistance and guidance with the use of gSOAP and the .NET framework.

This material is based upon work supported by the National Science Foundation under Grant Number 0428325.

8. REFERENCES

[1] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential serialization for optimized SOAP performance. In *High Performance Distributed Computing (HPDC)*, 2004.

[2] S. B. Baden. *Structured Adaptive Mesh Refinement (SAMR) Grid Methods*. Springer Verlag, Secaucus, NJ, USA, 1999.

[3] I. Banicescu and S. F. Hummel. Balancing processor loads and exploiting data locality in N-body simulations. In *Supercomputing (SC95)*, 1995.

[4] D. Calhoun and R. J. LeVeque. An accuracy study of mesh refinement on mapped grids. In *Adaptive Mesh Refinement - Theory And Applications: Proceedings of The Chicago Workshop On Adaptive Mesh Refinement Methods*, volume 41 of *Lecture Notes in Computational Science and Engineering*. Springer Verlag, 2003.

[5] C. S. Center. iCFDdatabase: The CFD database. `http://cfd.cineca.it/`.

[6] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *High Performance Distributed Computing (HPDC)*, 2002.

[7] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[8] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Supercomputing (SC06)*, 2006.

[9] B. He and Q. Luo. Cache-oblivious query processing. In *Conference on Innovative Data Systems Research (CIDR)*, 2007.

[10] Institute for Systems Biology. `http://www.systemsbiology.org`, 2007.

[11] E. Jäger, I. Altintas, J. Zhang, B. Ludšacher, D. Pennington, and W. Michener. A scientific workflow approach to distributed geospatial data processing using Web services. In *Symposium on Scientific Database Management (SSDBM)*, 2005.

[12] G. Kandaswamy, L. Fang, Y. Huang, S. Shirasuna, S. Marru, and D. Gannon. Building web services for scientific grid applications. *IBM Journal of Research and Development*, 50(2/3), 2006.

[13] S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, A. Rountev, and P. Sadayappan. Hypergraph partitioning for automatic memory hierarchy management. In *Supercomputing (SC06)*, 2006.

[14] Z. Lan, V. E. Taylor, and G. Bryan. Dynamic load balancing of SAMR applications on distributed systems. In *Supercomputing (SC01)*, 2001.

[15] X. Li, S. Ramanathan, and M. Parashar. Hierarchical partitioning techniques for structured adaptive mesh refinement (SAMR) applications. In *International Conference on Parallel Processing (ICPP)*, 2002.

[16] Y. Li and C. Meneveau. Material deformation in a restricted Euler model for turbulent flows: analytical solution and numerical tests. *Physics of Fluids*, 19, 2007.

[17] T. Malik, A. S. Szalay, T. Budavari, and A. R. Thakar. SkyQuery: A WebService approach to federate databases. In *Conference on Innovative Data Systems Research*, 2003.

[18] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1), 2001.

[19] N. Nedeljkovic and M. J. Quinn. Data-parallel programming on a network of heterogeneous workstations. In *High Performance Distributed Computing (HPDC)*, 1992.

[20] B. Rathakrishnan, C. Kleinerman, B. Richards, R. Venkatesh, V. Rao, and I. Kunen. Using CLR integration in SQL Server 2005. Technical report, Microsoft, 2005.

[21] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[22] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical N-body methods for multiprocessor architectures. *ACM Transactions on Computer Systems*, 13(2):141–202, 1995.

[23] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. Designing and mining multi-terabyte Astronomy archives: the Sloan Digital Sky Survey. In *ACM SIGMOD International Conference on Management of Data*, 2000.

[24] T. Tu and D. R. O'Hallaron. A computational database system for generating unstructured hexahedral meshes with billions of elements. In *Supercomputing (SC04)*, 2004.

[25] T. Tu, D. R. O'Hallaron, and O. Ghattas. Scalable parallel octree meshing for terascale applications. In *Supercomputing (SC05)*, 2005.

[26] R. A. van Engelen. gSOAP: SOAP C++ Web Services. `http://gsoap2.sourceforge.net/`.

[27] R. A. van Engelen. Pushing the SOAP envelope with Web services for scientific computing. In *International Conference on Web Services*, 2003.

[28] P. K. Yeung. DNSdb. `http://www.ae.gatech.edu/people/pyeung/DNSdb.html`.

[29] P. K. Yeung and S. B. Pope. Lagrangian statistics from direct numerical simulations of isotropic turbulence. *Journal of Fluid Mechanics*, 207, 1989.