



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

FY08 LDRD Final Report LOCAL: Locality-Optimizing Caching Algorithms and Layouts

P. Lindstrom

March 2, 2009

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Auspices Statement

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded by the Laboratory Directed Research and Development Program at LLNL under project tracking code 05-ERD-018.

FY08 LDRD Final Report
LOCAL: Locality-Optimizing Caching Algorithms and Layouts
LDRD Project Tracking Code: 05-ERD-018
Peter Lindstrom, Principal Investigator

Abstract

This project investigated layout and compression techniques for large, unstructured simulation data to reduce bandwidth requirements and latency in simulation I/O and subsequent post-processing, e.g. data analysis and visualization. The main goal was to eliminate the data-transfer bottleneck—for example, from disk to memory and from central processing unit to graphics processing unit—through coherent data access and by trading underutilized compute power for effective bandwidth and storage. This was accomplished by (1) designing algorithms that both enforce and exploit compactness and locality in unstructured data, and (2) adapting offline computations to a novel stream processing framework that supports pipelining and low-latency sequential access to compressed data. This report summarizes the techniques developed and results achieved, and includes references to publications that elaborate on the technical details of these methods.

Introduction/Background

Advances in remote sensing and high-performance computing hardware have led to a recent explosion in the size of geometric data set generated. For example, a single time step from a scientific simulation may involve up to billions of elements, exceeding the amount of memory available even on small-scale parallel machines, not to mention end-user desktops. Visualization and analysis techniques are commonly used to explore and make sense of such data sets. However, the sheer size of data poses significant problems in all stages of the simulation and post-processing pipeline, from simulation restart and visualization dumps, to offline processing of simulation data (e.g. filtering, remeshing, simplification, compression, hierarchy construction, analysis), to interactive queries (e.g. slicing, isocontouring, adaptive view-dependent meshing), to real-time rendering (e.g. organization, shipping, and caching of graphics primitives). Moreover, the data being processed is often unstructured in nature, which further complicates its management and representation.

Whereas CPU and GPU performance as well as storage capacity have by and large kept pace with the exponential growth in data, data access latency (e.g. disk seek time) and bandwidth (e.g. disk-to-memory transfer, possibly from distributed storage) have not. The large differences in performance between cache levels and relatively slow inter-cache data transfer cause major bottlenecks in data processing, especially for large data sets that span all levels of cache, but also for in-core processing and packaging of low-level graphics primitives. Furthermore, as the rapid growth in CPU and GPU performance is likely to be sustained over the foreseeable future, addressing the latency and bandwidth bottlenecks becomes increasingly important. Therefore redesigning post-processing algorithms to make better use of limited bandwidth but plentiful compute resources is critical to achieving efficiency and scalability.

Research Activities and Results

This project developed novel and complementary techniques in three primary areas, with the goal of improving cache and I/O efficiency of post-processing tasks. For tasks requiring processing of the whole data set, e.g. computation of derived fields, data analysis, and data reduction (simplification), we developed a framework for *stream processing* of unstructured data; a scalable linear-time solution to processing huge—even unbounded—amounts of data. The main idea behind stream processing is to reorder the computation to match the sequential organization of the data on secondary storage. For visualization queries like slicing, isocontouring, and stream lines, which can be implemented by accessing only a subset of the data, we designed *cache-oblivious data layouts* for unstructured data. These layouts improve the performance of random access to the data by exploiting the existing cache hierarchy in modern computers, thereby reducing the number of times data has to be transferred from slow but large to fast but small caches. These layouts permute the order of the data to better match anticipated access patterns without requiring any application code changes. Finally, to effectively reduce the volume of data having to be transferred both in sequential and random access computations, we designed several custom compressors for unstructured mesh and point data. Each of these three techniques is described in more detail below.

Stream Processing

Stream processing treats the data on sequential storage as a possibly infinite linear stream of data that passes through a small, fixed-size, in-memory window on which local random access is possible. As such, stream processing is fundamentally an *out-of-core* technique. Each processing module slides its window over the data stream, performing sequential reads through one end of the window and outputting the result of the computation at the other end. This mode of access allows multiple stream modules to be strung together in a pipeline, obviating the need to write intermediate results to disk, and supports compressed I/O with virtually no changes to readers and writers. Usually the memory footprint (the size of the window) is much smaller than main memory (e.g. a few megabytes), which further leads to good cache utilization.

The main challenge of streaming unstructured data is the need to encode and localize dependencies in the stream. For example, a mesh vertex may be referenced by cells anywhere in the stream, making it necessary to keep the vertex in memory from the point of first reference until no more future references in the stream remain. Thus poorly localized references grow the memory footprint and also introduce unwanted latency (i.e. buffering) of the stream (see [Figure 1](#)). For meshes, elements like vertices and cells must also be interleaved in the stream, calling for novel data representations.

The main theory of streaming unstructured meshes, including representations and data ordering, is described in [4]. Several papers further describe stream algorithms for processing unstructured data, including mesh simplification [13], mesh connectivity compression [6, 8, 10], and topology extraction [12]. A generalization of single-processor streaming to parallel architectures is described in [5], with computation of ghost layers around domain-decomposed data sets as application. Compared to prior out-of-core techniques based on external memory data structures, we have demonstrated order-of-magnitude speedups, much smaller memory footprints, and greatly reduced temporary and permanent disk use using our stream algorithms.

Cache-Oblivious Data Layout

Virtually all caches on modern computers manage blocks of contiguous bytes, translating to contiguous data elements (e.g. mesh vertices) in arrays and other linear data structures. For efficient cache utilization, it is important that data elements that are likely to be accessed in succession be stored together in memory, i.e. in the same cache block. The sequential ordering of data elements on linear storage is referred to as the *layout* of the data. *Cache-aware* data layouts are optimized for a particular cache and block size, and possibly also for a given cache replacement policy (e.g. least recently used). Because the memory hierarchy typically consists of several layers of cache of successively smaller size (both in terms of the number and

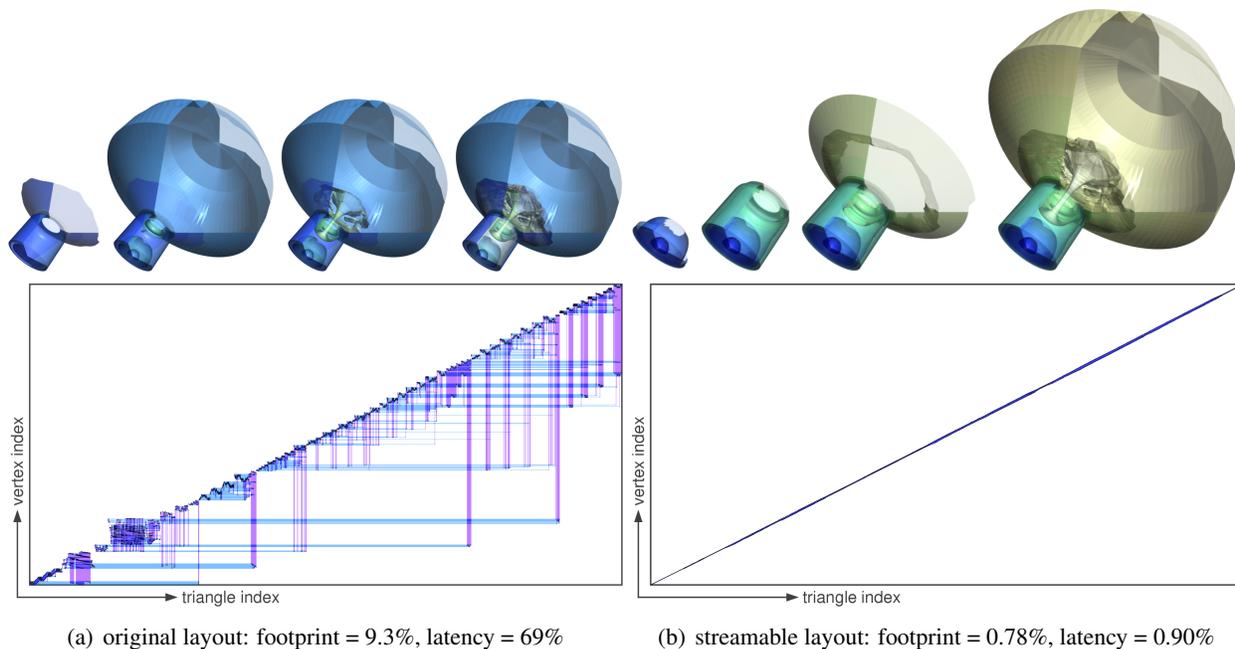


Figure 1: Snapshots of triangle sequences and diagrams for two different layouts of the same unstructured NIF target triangle mesh. The diagrams show references between triangles and vertices for the order in which they appear in the mesh file, with colored bars connecting the first and last vertex referenced by a triangle, and vice versa. Optimizing the sequential layout of vertices and triangles reduces the footprint (minimal memory needed) and latency (maximum time a vertex is buffered) for streaming the mesh through memory (percentages are of total mesh size).

size of blocks), there is no unique data layout optimal for all levels of cache. Worse yet, a layout carefully optimized for a given cache size may not only be unsuitable for another cache but may perform worse than unoptimized (though reasonably coherent) layouts.

To address this dichotomy, we have developed *cache-oblivious* layouts not optimized for any particular cache size, but which result in good average case performance across the whole memory hierarchy. The main idea is to model the data elements and accesses as an undirected, possibly weighted graph, and to optimize the graph layout so as to reduce edge crossings between blocks of successively smaller size, in effect nesting caches in a geometric progression. The graph nodes correspond to atomic data elements; its edges represent a non-negligible likelihood that two adjacent nodes are accessed together. Ideally all edges would join consecutive nodes in the layout, a goal that in general cannot be satisfied for two- and higher-dimensional data.

We have derived a cache-oblivious measure of locality that, using a single number, captures the locality of a graph layout. This measure makes it possible to characterize the quality of a layout, to compare layouts of the same data set, and to assess whether possibly expensive reordering is worthwhile. Furthermore, rather than relying on a particular constructive algorithm for ordering the graph, our measure enables different strategies for optimizing its layout. Not only is this measure applicable to unstructured meshes and graphs, but also to other data types like structured grids, images, and tree-like data structures. Figure 2 shows a surface mesh highlighting a large number of edges that join consecutive nodes. With their “fractal” appearance, our layouts can loosely be described as generalizations of space-filling curves to unstructured data. Indeed, when applied to structured data, a self-similar, space-filling pattern emerges.

Our cache-oblivious measure is described in [15], and improves upon earlier work by us [17]. Optimization of different data types and further applications are presented in [14, 18].

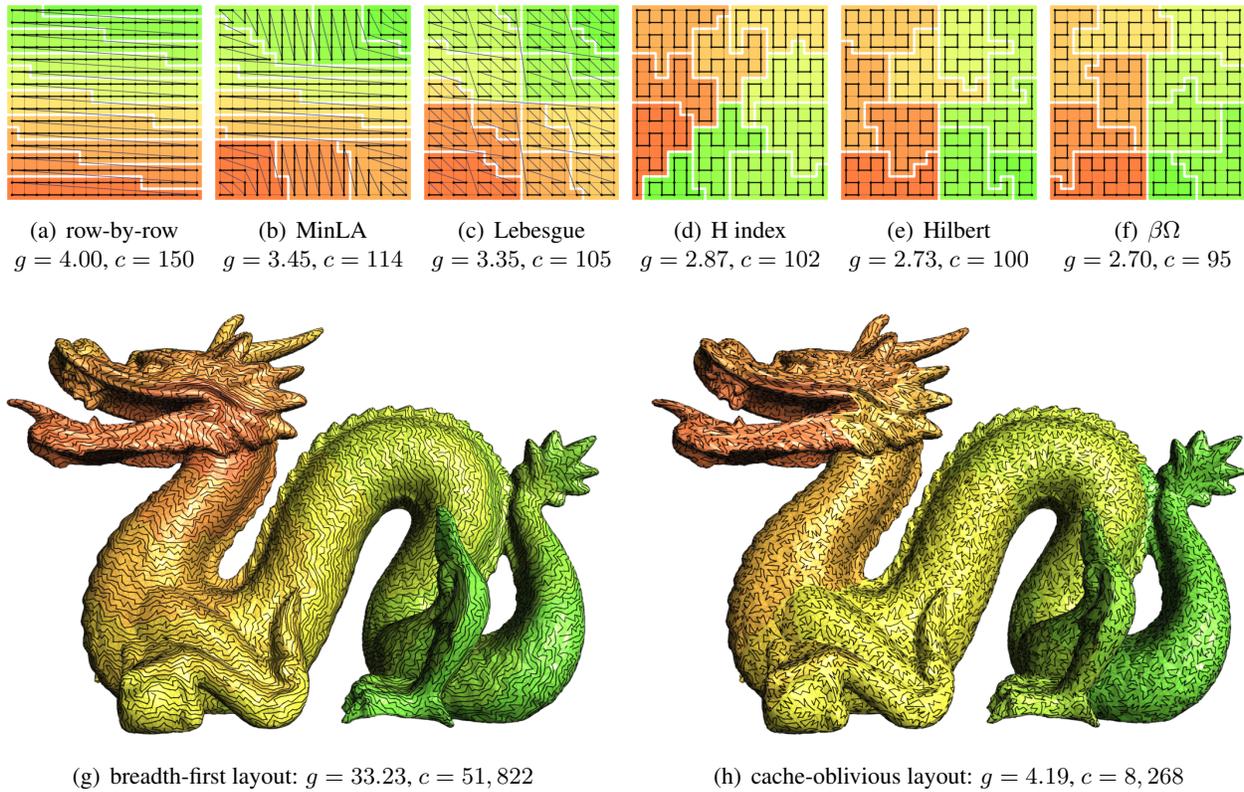


Figure 2: (a–f) Our cache-oblivious locality measure, g , of a layout correlates well with the edge cut, c , over a wide range of cache block sizes (a block size of 27 vertices is illustrated here). The edge cut for a given block size or domain decomposition is an important performance measure. It captures, e.g., the amount of communication needed in parallel applications, as well as the likelihood of a cache miss when traversing an edge in the grid. (g–h) Our measure is defined for any undirected graph, and thus applies equally to unstructured meshes. Here a breadth-first, row-by-row like layout and our cache-oblivious layout are compared using a block size of 256 vertices. The edges drawn connect consecutive vertices in the layout.

Data Compression

Data-parallel applications like scientific simulation and post-processing perform I/O in a very bursty manner, leading to periods of massive contention for shared I/O resources among the processing nodes. In such bandwidth-constrained environments, data compression can be an effective tool for boosting throughput by greatly reducing the amount of data being transferred. However general-purpose compression schemes such as GNU *gzip* perform very poorly on unstructured data such as computational meshes, in part because the data is usually heterogeneous and not byte-oriented, and in part because patterns are not readily visible in the data without additional transformation. In mesh data the connectivity usually consists of multi-byte integer sequences, which in a streaming environment must be interleaved with high dynamic range single- or double-precision floating-point numbers for the geometry and associated fields. Furthermore, to be adopted in simulation dumps the compression scheme must be fast, have a small memory footprint, operate online (streaming), and be entirely lossless.

This project has developed a number of custom compressors for simulation data such as unstructured meshes and floating-point arrays. Contrary to prior work on mesh compression, which achieves maximum reduction by deterministically and globally reordering the data, our compressors operate on streaming representations of the data much akin to the *gzip -c* mode of operation via Unix pipes. As such, compressed sequential I/O can be realized virtually transparently to the reading and writing applications.

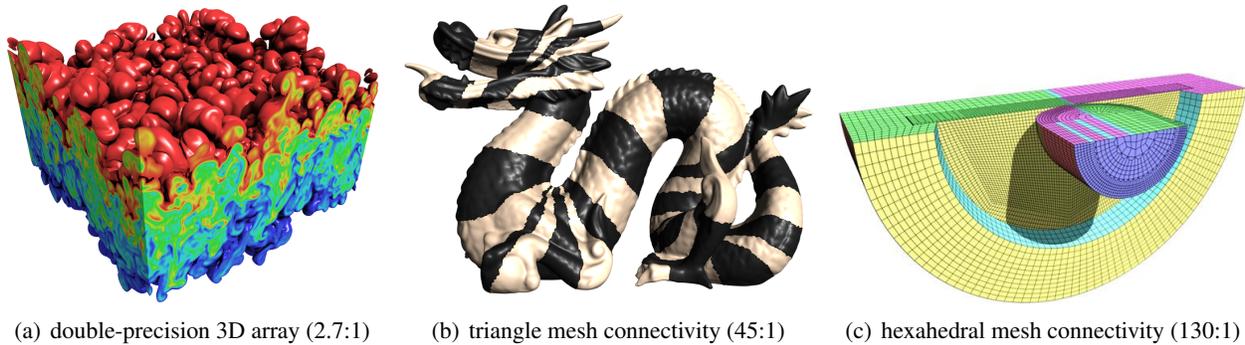


Figure 3: Typical lossless compression ratios for various types of data.

Being the first of their kind, our streaming compressors support triangular [8], tetrahedral [6], and hexahedral unstructured meshes [10]. We have also developed fast and effective compressors for various types of floating-point and semi-structured data [2, 3, 7, 9]. Finally, for situations in which random rather than sequential access is needed, we designed a triangle mesh (de)compressor that can selectively decode small portions of the compressed mesh, and which fetches, decodes, and caches the data entirely transparently to the application [16]. In several experiments in parallel I/O settings, we have shown that the CPU overhead of performing compression is virtually negligible, and that I/O time is reduced in proportion to the compression ratio; typically 40:1 for triangular and tetrahedral meshes and 100:1 to 1,000:1 for hexahedral meshes.

Summary

The successful conclusion of this project has enabled new efficient out-of-core and parallel methods for analyzing, processing, and visualizing unstructured data sets of unprecedented scale through order-of-magnitude reduction in memory and bandwidth use. Our techniques are being adopted by the programs via common I/O libraries and through direct interactions with code teams. For example, our hexahedral mesh and floating-point compressors are now supported by LLNL's *Silo* I/O API used by many simulation codes, and data readers and streaming techniques have been integrated with LLNL's VisIt visualization tool. Software implementations of our methods have been released to the general public [11]. Along with our five conference [3, 4, 6, 8, 10] and eleven journal publications [1, 2, 7, 9, 12, 13, 14, 15, 16, 17, 18], which to date have collectively been cited more than 250 times (source: *Google Scholar*), our work has also been well received by the academic community.

Acknowledgements

Several researchers, both from LLNL and academic institutions, contributed to the success of this project, including current LLNL employees Peer-Timo Bremer, Jonathan Cohen, Mark Duchaineau, Martin Isenburg, and Peter Lindstrom; Valerio Pascucci (formerly at LLNL, currently at University of Utah); Sung-Eui Yoon (formerly a postdoc with the PI at LLNL, currently at Korea Advanced Institute of Science and Technology); Dinesh Manocha and Jack Snoeyink (University of North Carolina at Chapel Hill); Jarek Rossignac (Georgia Institute of Technology); Jonathan Schewchuk (University of California, Berkeley); and Cláudio Silva (University of Utah).

References

- [1] S. P. Callahan, L. Bavoil, V. Pascucci, and C. T. Silva. Progressive Volume Rendering of Large Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1307–1314, 2006. UCRL-CONF-208680.
- [2] L. Ibarria, P. Lindstrom, and J. Rossignac. Spectral Interpolation on 3×3 Stencils for Prediction and Compression. *Journal of Computers*, 2(8):53–63, 2007. UCRL-JRNL-232201.
- [3] L. Ibarria, P. Lindstrom, and J. Rossignac. Spectral Predictors. *IEEE Data Compression Conference*, 163–172. 2007. UCRL-CONF-226261.
- [4] M. Isenburg and P. Lindstrom. Streaming Meshes. *IEEE Visualization*, 231–238. 2005. UCRL-CONF-211608.
- [5] M. Isenburg, P. Lindstrom, and H. Childs. Parallel and Streaming Generation of Ghost Data for Structured Grids. *Tech. Rep. UCRL-TR-403175-DRAFT*, Lawrence Livermore National Laboratory, 2008. Submitted to IEEE Transactions on Visualization and Computer Graphics.
- [6] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Shewchuk. Streaming Compression of Tetrahedral Volume Meshes. *Graphics Interface*, 115–121. 2006. UCRL-CONF-217274.
- [7] M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless Compression of Predicted Floating-Point Geometry. *Computer-Aided Design*, 39(7):869–877, 2005. UCRL-JRNL-208490.
- [8] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming Compression of Triangle Meshes. *Symposium on Geometry Processing*, 111–118. 2005. UCRL-CONF-210481.
- [9] P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006. UCRL-JRNL-220406.
- [10] P. Lindstrom and M. Isenburg. Lossless Compression of Hexahedral Meshes. *IEEE Data Compression Conference*, 192–201. 2008. UCRL-CONF-236522.
- [11] P. Lindstrom, S.-E. Yoon, and M. Isenburg. LOCAL Toolkit, 2007. UCRL-CODE-232243.
- [12] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust On-line Computation of Reeb Graphs: Simplicity and Speed. *ACM Transactions on Graphics*, 26(3):58, 2007. UCRL-JRNL-218501.
- [13] H. Vo, S. Callahan, P. Lindstrom, V. Pascucci, and C. Silva. Streaming Simplification of Tetrahedral Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):145–155, 2007. UCRL-JRNL-208710.
- [14] S.-E. Yoon, C. Lauterbach, and D. Manocha. R-LODs: Fast LOD-based Ray Tracing of Massive Models. *The Visual Computer*, 9(9–11):772–784, 2006. UCRL-JRNL-219092.
- [15] S.-E. Yoon and P. Lindstrom. Mesh Layouts for Block-Based Caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–1220, 2006. UCRL-JRNL-229656.
- [16] S.-E. Yoon and P. Lindstrom. Random Accessible Compressed Triangle Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1536–1543, 2007. UCRL-JRNL-229656.
- [17] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-Oblivious Mesh Layouts. *ACM Transactions on Graphics*, 24(3):886–893, 2005. UCRL-JRNL-211774.
- [18] S.-E. Yoon and D. Manocha. Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum*, 25(3):507–516, 2006. UCRL-JRNL-219070.