LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-403175

Parallel and Streaming Generation of Ghost Data for Structured Grids

M. Isenburg, P. Lindstrom, H. Childs

April 25, 2008

IEEE Computer Graphics and Applications

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.

Parallel and Streaming Generation of Ghost Data for Structured Grids

Abstract

Parallel simulations decompose large domains into many blocks. A fundamental requirement for subsequent parallel analysis and visualization is the presence of *ghost data* that supplements each block with a layer of adjacent data elements from neighboring blocks. The standard approach for generating ghost data requires all blocks to be in memory at once. This becomes impractical when there are fewer processors—and thus less aggregate memory—available for analysis than for simulation.

We describe an algorithm for generating ghost data for structured grids that uses many fewer processors than previously possible. Our algorithm stores as little as one block per processor in memory and can run on as few processors as are available (possibly just one). The key idea is to slightly change the size of the original blocks by declaring parts of them to be ghost data, and by later padding adjacent blocks with this data.

Keywords: parallel algorithms, streaming processing, ghost data, structured grids, iso-contour extraction

1 Introduction

Visualization and analysis is an important part of the simulation process, where end users analyze simulation outputs to gain scientifi insights. A key issue for visualization algorithms is the amount of data they must process. This amount is often directly proportional to the memory of the originating machine, which—in a parallel setting—is the combined memory available to all compute nodes. For large-scale simulations that utilize a hundred thousand processors or more, visualization algorithms are faced with processing unprecedented amounts of data.

Many approaches, including those of the three major data-parallel, end user-oriented visualization tools (EnSight, ParaView, and VisIt) follow the "pure parallelism" paradigm where they load an entire time slice into primary memory. This mode of operation requires so much primary memory that the visualization algorithms must themselves run in a massively parallel setting. However, there is a widening gap between the number of nodes available to run simulations and the number of nodes available to visualize them [3]. As the community pushes towards petascale simulations, the number of processors required for visualization tools based on pure parallelism is becoming prohibitively large.

One alternative to loading all of the data into memory is to process it out-of-core [12]. The output for one time slice of a parallel simulation is often stored as many pieces—usually one per processor. One strategy is to stream each piece of data through several algorithms and then move on to the next piece. Of course, this approach is not without pitfalls when it comes to algorithms that require collective communication to obtain information about other portions of the data set when processing these pieces. Perhaps the most important algorithm that requires collective communication and that does not immediately lend itself to an out-of-core setting is that of ghost data generation.

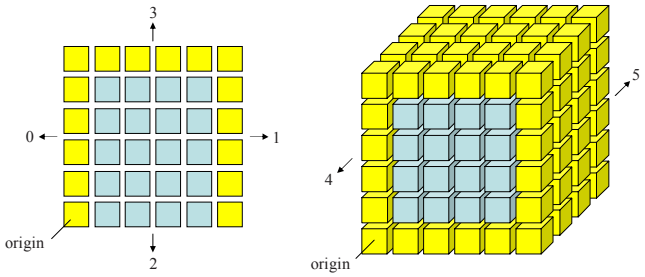

Figure 1: A two-dimensional and a three-dimensional block after adding ghost data. The blue boxes represent the data values, such as one scalar per cell, that are covered by this block. The yellow boxes represent one layer of ghost data. These data values are owned by a neighbor block (i.e. they exist there as a blue box) and are replicated here. This allows processing blocks independently from one another as computations along the block boundary can be carried out without having to access neighbor blocks. Note that there is no ghost data along the side where the block touches the domain boundary.

1.1 Ghost Data

Data-parallel visualization algorithms operate on a time slice partitioned across many processors and have each processor perform operations on its portion of the data set. Because visualization algorithms often need to incorporate the values of neighboring data points, special attention is needed for operating along the exterior of a given processor's data—unless we have ghost data. Replicated along the external boundaries, ghost data (see Figure 1) allows these algorithms to operate correctly and without further communication. A classic example for this is isocontouring.

The de-facto standard for contouring algorithms, Marching Cubes [10], assumes that the scalar fiel being contoured is define on the nodes (the cell corners). If node-centered data is stored without overlap, e.g. nodes from $x = [1 \ldots m]$ in one fil and $x = [m+1 \ldots 2m]$ in another (using a 1D analogue to illustrate our point), then a cell needs to be created between $x = [m, m+1]$ to form a complete, crack-free iso-contour, which is conveniently solved by replicating $x = m$ and $x = m+1$ as ghost nodes. Similarly, when isocontouring a fiel that is define at the center of each cell, the contouring algorithm firs interpolates the fiel to the nodes. Ghost data allows the interpolation to be consistent on both sides of the boundary, thereby ensuring a crack-free surface.

The uses of ghost data are not limited to isocontouring. Ghost data is also needed for gradient calculations, for reconstructing material interfaces in Eulerian hydrodynamics simulations, for smoothing out scalars for volume renderings with pseudocolors, and many other computations involving neighbor data.

1.2 Why Storing Ghost Data Is Impractical

The algorithm we propose in this paper is necessary because simulation codes rarely store out ghost data. The simulations themselves usually have ghost data, but they discard it when outputting file to reduce not only fil size but in particular I/O time. Although a single layer of elements seems small, it can make a significan difference. In this paper, we study a twenty-seven billion element Rayleigh-Taylor Instability calculation, which writes out blocks of size $12 \times 12 \times 3072$. Adding ghost data would result in blocks of

size $14 \times 14 \times 3072$, an increase of 36%. The authors have experience working with another turbulence simulation (not profile in this paper) that has blocks that are $6 \times 6 \times 6$. Adding ghost data in this case would result in blocks of size $8 \times 8 \times 8$, an increase of 137%. Further, as the trend in petascale computing is to have a small amount of primary memory on each processor, block sizes will continue to get smaller, meaning that the overhead for storing ghost data will be increasing.

Worse, the cost of storing ghost data is felt by all parties: during write (for the simulation code), read (for either the simulation code or the visualization code), and especially in the footprint on the disk. Furthermore, the discussion above assumes that a single layer of ghost data is already sufficien for all subsequent visualization and analysis needs. Storing additional layers (assuming they are available to the simulation) eventually becomes an even greater burden. Finally, as visualization tools can service many simulation codes, solving this problem once in the visualization tool (as opposed to many times in many simulation codes), and computing ghost data only when necessary, has merit.

1.3 Contribution

We describe a parallel and streaming out-of-core algorithm to generate ghost data for structured grids that is *very* scalable because we do not require all blocks to be in memory at once. This does not come at the expense of increased I/O: we read and write each block only one time. We assign to each processor a set of blocks which it processes one by one (while respecting certain dependencies when deciding which block to process next). The key idea is to slightly change the size of the original blocks by declaring their outermost data layers to be ghost data. These layers are then kept around for neighboring blocks to which they will be attached later. Hence, at any given time each processor stores at most one full block of data plus some layers from previous blocks. In theory, our algorithm handles the extreme case of generating ghost data on a *single* processor for the *hundreds of thousands* of blocks that are result of a typical peta-scale scale simulation. In practice, our algorithm makes efficien use of as many processors as there are available.

The input to our algorithm is a structured grid of one or several scalar components that is partitioned into m blocks. The algorithm outputs m blocks enhanced with ghost data. The output blocks, when stripped of ghost data, will differ slightly in size from the input blocks, but like the input will cover the domain exactly. When generating one layer of ghost data each output block will either grow or shrink by one layer of (non-ghost) data in each direction.

A number of systems have been developed for parallel streaming visualization [1, 2, 6, 9]. But none of these systems address the issue of generating ghost data out-of-core. They assume that ghost data is readily available (which we believe will be impractical at the petascale) or focus on the narrower set of use cases that can be accomplished without ghost data. We believe the algorithm presented in this paper is the fi st to consider the generation of ghost data in a streaming environment.

2 Ghost Data Generation

For reasons of simplicity we firs describe our algorithm for processing 2D data on a single processor; later we extend it to multiple processors and to the 3D case. We limit our description to one layer of ghost data and to data values stored with either grid cells or grid nodes. The extension to multiple layers of ghost data and to data values stored with both grid cells and grid nodes are conceptually identical and would unnecessarily complicate the details.

As input we expect a description of how the domain, a large structured grid, is partitioned into a grid of blocks (each a smaller structured grid), and methods to read and write a selected block. From the description of the domain we construct a data structure that describes the adjacency between blocks. We maintain a record for each block that points to the four (or in 3D six) neighbors it shares an edge (or in 3D a face) with. We also store the size and the origin (i.e. offset within the whole grid) with each block. Note that this block data structure does not store the actual data of the blocks.

2.1 Single Processor

Intuitively our algorithm works as follows: we select one block, load it from disk, possibly grow it along its edges using data we have in memory, declare data along the edges of the block to be ghost data, possibly copy some data along the edges for later use, output the block for immediate consumption to another process (rather than for storage on disk), discard it from memory, and continue with the next block. We illustrate this with a small example in Figure 2. Because at any time only one block and some edge data from previous blocks are in memory, we can process grids that are much larger than the available main memory.

We create ghost data simply by declaring the data along the edges of the block to be ghost data, which essentially shrinks the actual data content of this block. We only do this for edges along which we need to create ghost data, that is, where an edge-adjacent neighbor block exists. For the firs block processed in Figure 2, for example, these are the bottom edge and the right edge.

Before deallocating a block we make a copy of the data along the edges where we created ghost data and keep this copy around until the corresponding edge-adjacent neighbor block is processed. When creating one layer of ghost data we keep two layers of data: (1) The layer along the edge that was declared ghost data, which will later appear as actual data in a neighboring block, and (2) the layer adjacent to it, which will serve as ghost data for the neighboring block. We refer to these layers as the *edge data* or simply the *edge*.

After loading a block and before creating its ghost data we grow the block with edge data from any edge-adjacent, already processed neighbor block. This edge can subsequently be deallocated. Part of this edge was declared ghost data in the processed neighbor and becomes actual data in this block, and part of this edge was actual data in the neighbor and becomes ghost data here.

For each created edge we make a copy of its associated two layers of data and record the edge's dimensions and origin within the domain. The edge is accessible via a pointer stored with the corresponding block. The data at the corners of a block is sometimes duplicated in two edges, and is sometimes stored but not used. While this may seem wasteful in terms of memory use it simplifie the implementation by liberating us from having to maintain corners separately. For example, in the third step in Figure 2 our implementation grows the eight-column wide bottom-right block with an edge that is ten columns wide (i.e. with the two bottom columns of the block that was output in the second step). The leftmost 2×2 piece of this ten-column-wide edge corresponds to the bottom-right corner of the top-left block. This corner piece, while transmitted and stored, does not get attached to and further processed with the block. A similar inefficien y can occur when two received edges have an overlapping corner (because one copy of the overlap was redundant).

In summary, for each edge of the processed block we either keep a copy of the edge data for an edge-adjacent neighbor (i.e. for neighbors that have not been processed yet) or we grow the block using the edge data of an edge-adjacent neighbor (i.e. for neighbors that have already been processed). This means that edge data travels in only one direction between two adjacent blocks, and that the blocks shrink or grow correspondingly. Which direction the edge data travels depends on the order in which we select the blocks for processing. (Note: for the parallel algorithm we describe later there will be a few edges for which this direction is globally agreed upon.)

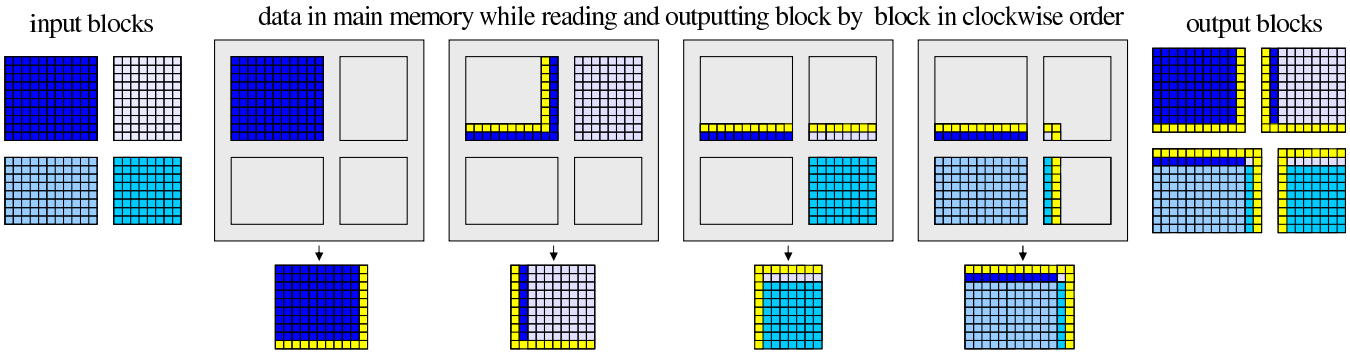

Figure 2: Streaming generation of ghost data on one processor for a domain partitioned into four blocks. At most one block, plus some edge data from previous blocks, is in memory at any time. The input data is shown in four shades of blue, the generated ghost data in yellow.

2.1.1 Row Ordering

On a single processor it is best to process the blocks row-by-row (zigzag or typewriter). This makes the algorithm especially simple to implement as it completely avoids the special cases that need to be considered when processing the blocks in other orderings (see next subsection). It is also the most memory-efficien way to process the data as it keeps the total number of edges that are in memory to a minimum. If each row contains r blocks then we need to keep maximally $r+1$ edges at any time in memory (i.e. parallel portions of two consecutive rows joined by an orthogonal edge).

2.1.2 Flexible Ordering

When we choose some other block ordering (as needed in the parallel case) we have to impose constraints on which blocks are eligible for processing. In particular, we need to check whether the block that is being considered passes the *corner rule* by examining how it is adjacent to already processed blocks. We disallow processing a block that is corner-adjacent to an already processed block if both of their common neighbor blocks (to which they are edge-adjacent) have not been processed. Such a configuratio would prevent us from properly growing blocks that arrive later.

We illustrate this scenario in Figure 3. Block A and block B are corner-adjacent and have—in violation of the corner rule—already been processed. This causes trouble when processing block C. We cannot grow block C with the edge data from both blocks A and B as there is not enough data to fil a block completely—one corner would be missing. Yet if we grow block C with the edge data only from A (or only from B) we leave behind two directly opposite edges that will never be used to grow a block. This would mean that two columns (or two rows) of data from the input blocks B and C (or A and C) will be missing in the generated output blocks.

2.1.3 Implementation

Given a data structure for querying edge-adjacent blocks and methods for reading and writing a block, looping over the following list of operations implements the algorithm:

select: Select a block that passes the corner rule for processing.

read: Read the selected block from disk.

grow: Absorb edge data from each edge-adjacent neighbor that has already been processed and deallocate those edges.

keep: Allocate and copy edge data for each edge-adjacent neighbor that has not yet been processed and keep it for later.

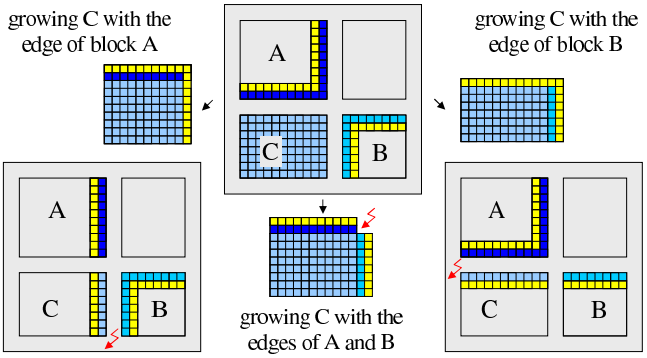

Figure 3: Processing the corner-adjacent blocks A and B first leads to an unresolvable situation when block C arrives (middle top). There is not enough data to grow C in both directions (middle bottom) while growing it only in one direction leaves two directly opposite edges behind that will never be used to grow a block (left and right).

write: Output the block for immediate consumption to another process (or–less preferred–for storage on disk) and discard it. The data along the boundary of the output block is now ghost data (unless it coincides with the domain boundary).

2.2 Multiple Processors

In the parallel case we have each processor partition the graph of edge-connected blocks into as many parts as there are processors. We do this in some deterministic manner so that each processor ends up with the exact same partitioning. We specify this partitioning by storing with each block the index of the part it belongs to. Each processor owns all the blocks whose index equals its rank. It is not necessary to keep the entire partitioned block data structure in memory; only the blocks owned by a processor and their immediate neighbors on other processors are needed.

The challenge is now that some blocks need to communicate edge data with a neighbor block that resides on another processor. This is the case for all blocks that have one or more edges on the partition boundary. The two processors owning the two blocks on either side of such an edge need to agree on the order in which the two blocks are processed to avoid processing the blocks simultaneously. If this were to happen neither block would utilize the edge data from the respective other and the processors would try to send each other edge data for an already processed block.

We call a neighbor block that resides on another processor a *remote neighbor* in contrast to a *local neighbor* that resides on the same processor. We say that a neighbor is "above" a block when it

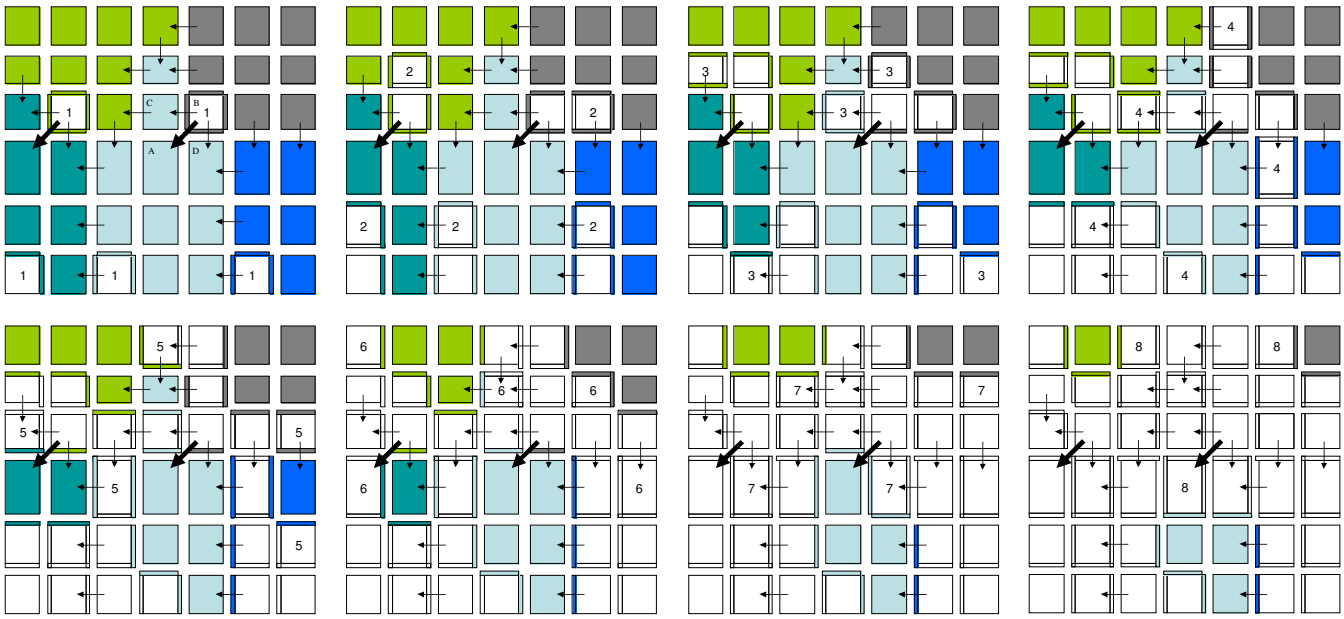

Figure 4: Example of parallel operation with five processors. The colors indicate different processors; the numbers indicate the order in which the blocks are processed. The skinny arrows indicate the global travel direction for edges along the partition boundary; the bold diagonal arrows are additional dependencies. The long skinny boxes correspond to the edges: Solid edges have been copied from the block they are adjacent to, and are currently in memory. Hollow edges have been used to grow the neighbor block they are adjacent to, and are no longer in memory.

is to the right (1), on top (3), or (in 3D) behind (5) that block. The numbers correspond to the directions shown with arrows in Figure 1. Correspondingly, we say that a neighbor is "below" a block when it is to the left (0), beneath (2), or (in 3D) in front of (4) that block. We call an edge on the partition boundary a *partition edge*.

2.2.1 Global Travel Direction

We impose a global travel direction for the edges along the partition boundary. Blocks send partition edge data to edge-adjacent remote neighbors that are below them. Correspondingly, blocks receive partition edge data from edge-adjacent remote neighbors that are above them. This define a partial order on the block selection along the partition boundary: Blocks that have one or more remote neighbors above them have to wait until these blocks are processed on other processors and their partition edge data has been received.

The global travel direction of the edge data assures two things. First, there are no circular dependencies that would cause a deadlock. And second, the resulting partial ordering that is imposed on the blocks is guaranteed to be compatible with the corner-rule. Other partial orderings, for example, letting edges travel towards the block with the smaller processor ID, will in general not be compatible with the corner rule.

We implement the partial order dictated by the global travel direction as follows: For each block that is owned by the processor we check whether it has any edge-adjacent remote blocks and, if so, increment counters stored with the block for the number of partition edges that need to be sent or received accordingly. Blocks that have a non-zero receive counter are not considered for processing until all their partition edges have arrived. While these blocks are waiting we keep the processor busy by processing other blocks. Blocks that have a non-zero send counter are prioritized for processing to shorten the wait time for blocks on other processors.

We need to introduce one additional dependency to prevent certain blocks from being processed too early. In the example in Figure 4 we illustrate these extra dependencies with bold arrows. In these scenarios a block A has to wait until a corner-adjacent block B on a different processor has been processed. This is necessary whenever the two blocks C and D that are edge-adjacent to both A and B are on the same processor as A and each wait for an edge from B. If we were to process A before B we would create a deadlock: C and D cannot be processed because they are waiting for an edge from B, while B cannot be processed next because it would violate the corner-rule. We fin such configuration during initialization and add an additional dependency to these blocks by giving them a pointer to the block they must wait for. Hence, block A has to wait until the off-processor block B has been processed, which is signaled through the arrival of edge data from B.

2.2.2 Implementation

The main difference of the multi-processor implementation over the single processor case are the initial partitioning, the additional constraints and heuristics for selecting the next block, and the communication of edges that are on the partition boundary.

select: Select for processing a block that is owned by this processor, has a nonzero receive counter, has no additional dependency (either because its dependency pointer is zero or because the dependency is already resolved because the block it points to is already processed), and passes the corner rule.

read: Read the selected block from disk.

grow: Absorb from each edge-adjacent neighbor (local and remote) edge data that has already been processed to grow the block and then discard these edges.

keep: Allocate and copy edge data for each edge-adjacent neighbor that has not yet been processed. If the neighbor is local, keep the edge in memory for later; if the neighbor is remote, send the edge to the corresponding processor.

write: Output the block for immediate consumption to another process (or–less preferred–for storage on disk) and discard it.

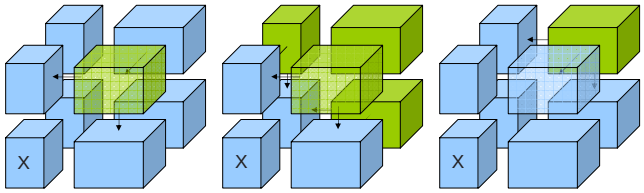

Figure 5: Additional dependencies in the 3D case for the block marked with 'X'. Blocks with the same color are on the same processor. Arrows indicate the global travel direction of faces: (a) dependent on one edge-adjacent block, (b) dependent on all three edge-adjacent blocks, (c) dependent on the corner-adjacent block.

The data along the boundary of the output block is now ghost data (unless it coincides with the domain boundary).

receive: Check for edges received from other processors. For each such edge update the block data structure: associate the edge with the remote neighbor that it is from, mark the neighbor as already processed, and decrement the receive counter of the block that was waiting for this edge.

We implement inter process communication of edges using non-blocking MPI `Isend` and `Irecv` calls. On each processor we post one (or several) non-blocking receive calls for each processor we expect to receive edge data from. We use non-blocking send calls when edge data is sent to a remote neighbor. After processing a block we firs check whether any edges were received before processing the next block. A process will temporarily stall only if there are no allowable blocks for processing because all remaining blocks wait (directly or indirectly) for some edge to arrive.

2.3 Extension to 3D Grids

The algorithm for ghost data creation in 3D is similar to the 2D version, with the main difference being that we copy data in units of faces rather than edges when processing a block. Each block has up to six face-adjacent neighbor blocks. If a face-adjacent neighbor has not yet been processed we keep a copy of the face data around that will later be used to grow that neighbor block. If a face-adjacent neighbor has already been processed we use the face data that was copied earlier to grow the block and deallocate the face data.

On a single processor the simplest way to process the grid is layer-by-layer and row-by-row as all special cases are avoided. For other block orderings we need to check again whether the selected block is allowed for processing. In 3D a block has to pass the *edge rule* and the *corner rule*. Intuitively these two rules state that the selected block cannot touch already processed blocks *only* along an edge or at a corner. The block must also be adjacent to at least one face incident to such an edge or corner. The reasoning behind this is the same as for the corner rule in 2D (see Figure 3).

On multiple processors there is one more global travel direction for faces along the partition boundary. Yet the basic principle remains the same: blocks send their face data to any remote neighbor that is "below" them and blocks receive faces from any remote neighbor that is "above" them (see Section 2.2).

As in the 2D case we need to introduce additional dependencies to prevent certain blocks from being processed too early. More precisely, some blocks have to wait until an edge-adjacent or a corner-adjacent block owned by a different processor has been processed. These situations are illustrated in Figure 5. In the firs scenario block X has to wait until the edge-adjacent block has been processed. The reasoning is the same as in the 2D case: processing block X firs would result in a deadlock as its face-adjacent blocks wait to receive faces from the edge-adjacent remote neighbor, which in turn could not be processed without violating the edge rule. In the second scenario block X has to wait for three edge-adjacent blocks. In the third scenario block X has to wait for the corner-adjacent block. We fin these configuration during set-up and add explicit dependency pointers to the respective blocks in the block data structure.

By choosing an appropriate assignment of blocks to processors we can avoid such additional dependencies altogether. For example, when partitioning the blocks with planar axis-aligned cuts of the domain we cannot create any of the scenarios depicted in Figure 5. Because additional dependencies increase the likelihood that processors have to wait for one another we prefer to use such assignments (such as the "cut" assignment in the next section).

3 Experimental Results

We integrated our parallel streaming ghost data algorithm into VisIt [4] and evaluated its performance on two data sets containing 8 and 27 billion elements, respectively. We use the smaller data set to investigate how different assignments affect the performance and the larger data set to demonstrate scalability—especially compared to the standard non-streaming approach. The data sets are:

RM8: $2048 \times 2048 \times 1920 \approx 8$ billion single-byte scalar values stored as $16 \times 16 \times 30 = 7,680$ gzipped blocks of size $128 \times 128 \times 64$. This is the entropy fiel of a Richtmyer-Meshkov instability simulation [11].

RT27: $3072 \times 3072 \times 3072 \approx 27$ billion four-byte floating-poin scalar values stored as $256 \times 256 = 65,536$ uncompressed blocks of size $24 \times 24 \times 3072$. This is the density fiel of a Rayleigh-Taylor instability simulation [5].

For each data set we run two different computations that require ghost data: histogramming of the gradient magnitude field and iso-contour extraction. Computing a histogram represents a scenario where the computation load per block is balanced but very light, which can lead to I/O bound operation. Iso-contour extraction highlights the problems of a load-imbalance where processing some blocks takes significantl longer than processing others as the amount of iso-contour per block varies drastically. Moreover, significan memory is needed to store the generated iso-contour.

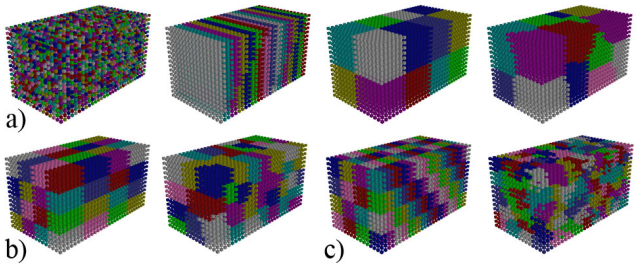

Figure 6: Different assignments of blocks (shown as small cubes) to processors (indicated by color) for the 7,680 blocks of RM8: (a) the random, slice, cut, and metis assignment for 16 processors, (b) cut and metis for 64 processors, and (c) cut and metis for 240 processors.

On the smaller RM8 data set we evaluate the performance of different strategies for assigning blocks to processors. The four different assignments we used are also illustrated in Figure 6:

random: Blocks are randomly assigned to processors.

slice: Consecutive blocks ordered layer-by-layer and row-by-row.

cut: Cutting planes partition the domain into rectangular parts.

metis: The MeTiS software [8] partitions a graph representing the face-adjacency between the blocks.

All experiments were run on LLNL's ASC Prism machine: a 128-node Linux visualization cluster with dual 2.4 GHz AMD Opteron CPUs, 16 GB of main memory per node, a 4x InfiniBan interconnect, and a Lustre parallel f le system.

3.1 Gradient Histograms on RM8

Figure 7 shows for all four assignments the maximal memory footprint and the efficien y (speedup divided by processor count) for computing gradient histograms on RM8 running with 1 to 240 processors. The differences in memory footprint are insignifican in practice and can be attributed to the different maximum number of faces stored simultaneously. This number drops as the blocks are distributed over more processors where fewer faces are buffered per processor due to fewer interior faces and more dependencies.

As expected, the efficien y is overall best for the cut assignment. It drops from 92 percent on 64 processors to 83 percent on 240 processors, not because processors are waiting for one another, but because the load time for the blocks varies. Some of the processors are assigned blocks with highly varying data content that does not compress well with gzip and whose size on disk is much larger than average. This is illustrated by the varying length of the blue segments (load time) and the absence of yellow segments (wait time) in the top right Gantt chart in Figure 7. The vertical axis in this chart corresponds to the processor ID (0 at the top) and the horizontal axis to time. Blue means loading a data block from disk, green processing a block, yellow waiting for face data from another processor, and red idling for other processors to finish

The efficien y of the slice assignment drops at 32 processors where the slices are only one block wide. In this case the processors are essentially serialized as each slice but the firs waits for faces from the slice above, which creates a lot of idle time (see Figure 7). The efficien y is especially poor for the metis assignment on 240 processors—it drops to 16 percent—due to excessive dependencies between processors. The reason is that the boundaries between parts become so "ragged" (as illustrated in Figure 6) that nearly all blocks are tied up in corner and edge constraints and wait for faces from other processors. In contrast, the cut assignment does—by construction—not have any corner or edge constraints.

3.2 Iso-contours on RM8

The performance of parallel iso-contour extraction is illustrated by the graphs in Figure 8. This iso-contour (see [7, 11]) is dense, practically space-filling and cuts horizontally across the center of the domain. Several of the 30 layers of blocks below and above the surface are empty, however, so we can expect load balancing issues.

We should point out that the reported memory footprint includes the extracted iso-contour that accumulates as blocks are processed. For on-the-fl rendering or output to disk this could be avoided using a *streaming mesh* representation for the iso-contour [7]. However VisIt does not yet support such a representation. As seen in Figure 8, the random assignment has by far the lowest maximal memory footprint: each processor is assigned a more or less balanced mix of empty and full blocks, and hence produces roughly the same amount iso-contour. Metis and Slice have a plateau going from 2 to 4 processors. This is because of an unbalanced division of blocks containing iso-contour. Going from two parts to four by inserting horizontal partitions creates two parts that are empty of iso-contour, so that the memory footprint of the other two parts remains unchanged. The same happens when going from 8 to 16 processors for the cut assignment. Eventually each additional cut halves the memory footprint. This happens when the part that is responsible for reporting the maximal memory footprint consists entirely of blocks full of iso-contour.

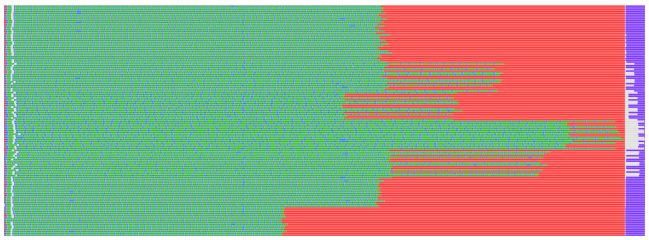

Figure 9: Gantt chart for isocontouring the RM8 data set with the cut assignment on 128 processors. The decrease in efficiency neither comes from waiting for processors (i.e. there is no yellow) nor from loading of blocks (i.e. no long blue lines) but has to do with the uneven load balance of the cut assignment: the processors in the center have most of the contouring work to do.

CPUs	total time [sec]	% of time spent on load	process	wait	idle	other
1	1780.4	28.1	61.6	0.0	0.0	10.2
2	875.1	23.6	63.2	0.0	2.7	10.5
4	438.8	26.3	62.8	0.0	2.6	8.4
8	217.6	24.5	63.4	0.0	3.3	8.9
16	141.6	22.5	48.6	0.0	20.7	8.2
32	74.3	23.9	46.4	0.0	21.7	7.9
64	34.4	15.7	50.2	0.0	25.1	9.1
128	19.5	13.9	44.2	0.0	32.7	9.1
240	10.8	13.7	42.7	0.1	33.7	9.8

Table 1: Timing results for isocontouring the RM8 data set with the cut assignment on 1 to 240 processors. We also report the percentage of time the processors spend **load**ing blocks, **process**ing them, **wait**ing for face data from other processors to arrive, and **idle**ing until all other processor are done. The time spent on miscellaneous tasks (e.g. rendering) is reported as **other**.

Again and as expected, the cut assignment generally outperforms the other assignments in terms of efficien y, and we report detailed timing results in Table 1. The reason for the drop in efficien y is the lack of load balancing: some processors have only blocks without iso-contour, while others have iso-contour in every block. The Gantt chart for running the cut assignment on 128 processors (see Figure 9) clearly shows this. The efficien y does not decrease because processors are waiting: there is no yellow in this figure But, as indicated by the long red lines, many processors are idle in the end waiting for other processors to finis their contouring work. The random assignment is surprisingly good when running on fewer, say up to 64, processors; the better load balance for iso-contour extraction makes up for the increasingly excessive dependencies and communication between processors.

When running on a single processor we process the blocks row-by-row and layer-by-layer so that the faces simultaneously allocated correspond to the interface between two layers of blocks. On multiple processors, surprisingly the random assignment leads to the fewest number of simultaneously allocated faces. This is because faces do not get buffered very long but get immediately used by the blocks that have been waiting for them. However, the random assignment needs to communicate the largest number of faces by far, as illustrated by the rightmost graph in Figure 8.

3.3 Gradient Histograms and Contouring on RT27

The Gantt charts in Figure 10 give a good illustration of the difference between our new streaming approach and the non-streaming operation already supported in VisIt. The non-streaming approach

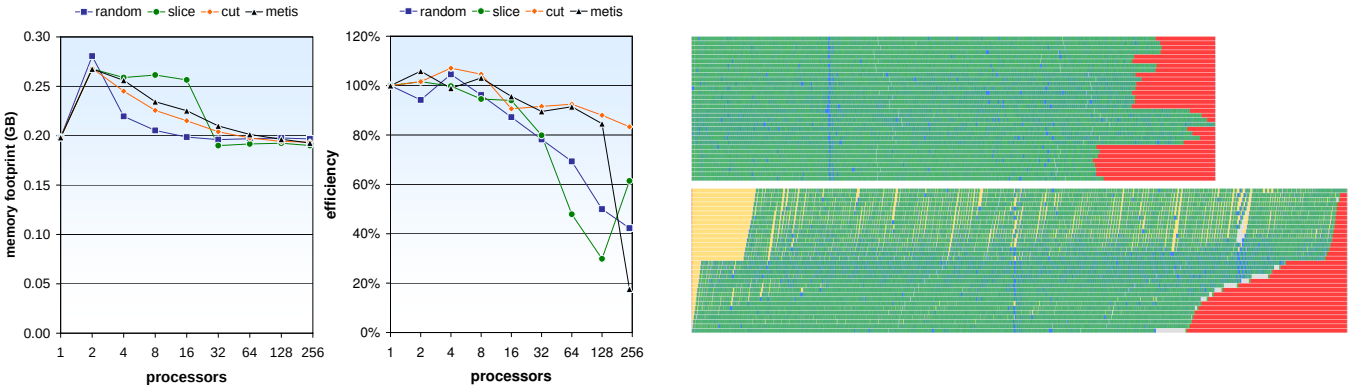

Figure 7: Results of computing gradient histograms on RM8 for different processor counts and block assignments. Left: The maximal memory footprint on any one processor. Middle: Parallel efficiency (speedup divided by processor count). Bottom right: Gantt chart showing task breakdown on 32 processors using the slice assignment. The yellow (especially in the top left corner) illustrates the serialization of processors that have to wait until results (i.e. block faces) trickle in from other processors, resulting in 80% efficiency. Top right: Corresponding Gantt chart for the cut assignment, illustrating a 13% shorter total processing time and a higher efficiency of 92%. The red bars indicate load imbalance due to some processors being delayed by longer load and gunzip times (blue) because the blocks they are assigned contain more varying data.

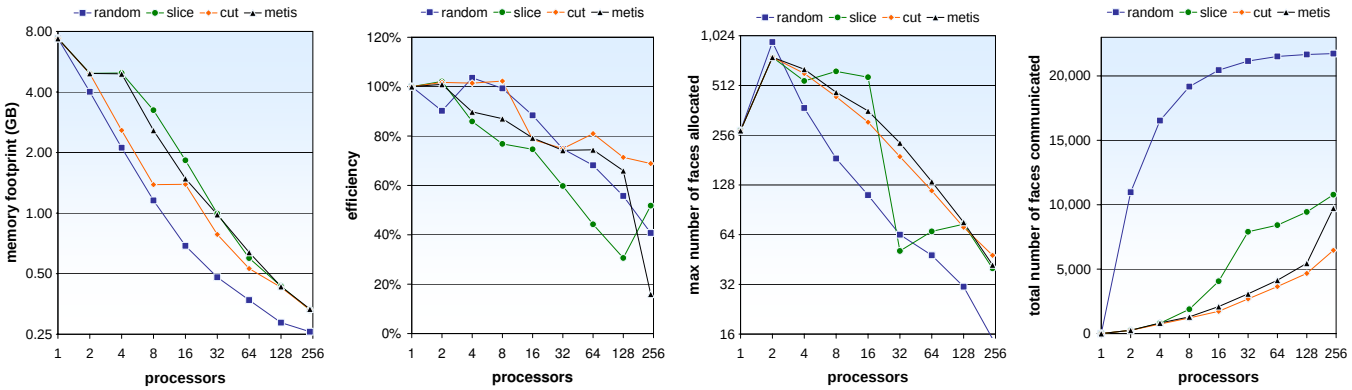

Figure 8: Performance curves for isocontouring the RM8 data set. From left to right: the maximal memory usage on any processor, the efficiency, the maximal number of simultaneously allocated faces on any processor, and the total number of faces communicated between processors.

firs loads all blocks, then communicates all ghost data, before it starts the contouring, whereas the streaming approach interleaves loading, communicating, and contouring per block.

The gradient histogram computation on RT27 illustrates the scalability of our method in terms of memory use (leftmost graph in Figure 10). The streaming approach uses less than half a gigabyte per node independent of the number of processors. In contrast, the non-streaming approach requires a minimum of 128 processors, each using over 6 GB of RAM, to handle this large 108 GB data set. Hence, the streaming approach to ghost data creation enables us to process larger data sets with fewer processors, in this case as few as one.

Also for iso-contouring our streaming approach makes it possible to perform the computation on many fewer processors than previously possible. The non-streaming approach needs at least 64 processors to be able to run, whereas the streaming approach needs only four. Again, if the iso-contour would be rendered (or signifi cantly simplified on-the-fl or piped to disk instead of being accumulated in memory our method could run on a single processor.

4 Discussion

This paper is about streaming generation of ghost data in a parallel setting. Our algorithm can run on a single processor and scales with good efficien y to as many processors as may be available. For data sets that do not fi in aggregate memory, our method is to our knowledge the firs to solve the problem of generating ghost data; a fundamental precondition in parallel visualization tools for many downstream processing tasks [1,2,6,9]. Ghost data is needed for isocontouring, for gradient calculation, for material interface reconstruction, and for many other calculations. We focus on contouring because it is a well known problem for the community and it demonstrates well known (and consistent) performance properties. But we emphasize that we have contributed a solution for ghost data creation that is distinct from contouring.

One feature of our algorithm is simplicity. Because we establish an agreed-upon global travel direction for the block faces, we make all dependencies implicit to the block assignment such that each processor can decide on its own which block to process next. Alternate approaches, such as a master-slave approach, are considerably more complex. Another feature is that our method is enabling in the sense that it allows completing downstream computation tasks on many fewer processors than previously possible.

When running on a single processor our algorithm is practically optimal. We do not explicitely demonstrate this in the paper but we can generate ghost data for hundreds of thousands of blocks even on a single processor. Some memory math should convince you quickly: On a single processor we store at most one original data block plus one "sheet" of ghost data through the entire domain. For a domain consisting of 1 trillion double words decomposed into $100 \times 100 \times 100 = 1$ million blocks where each block

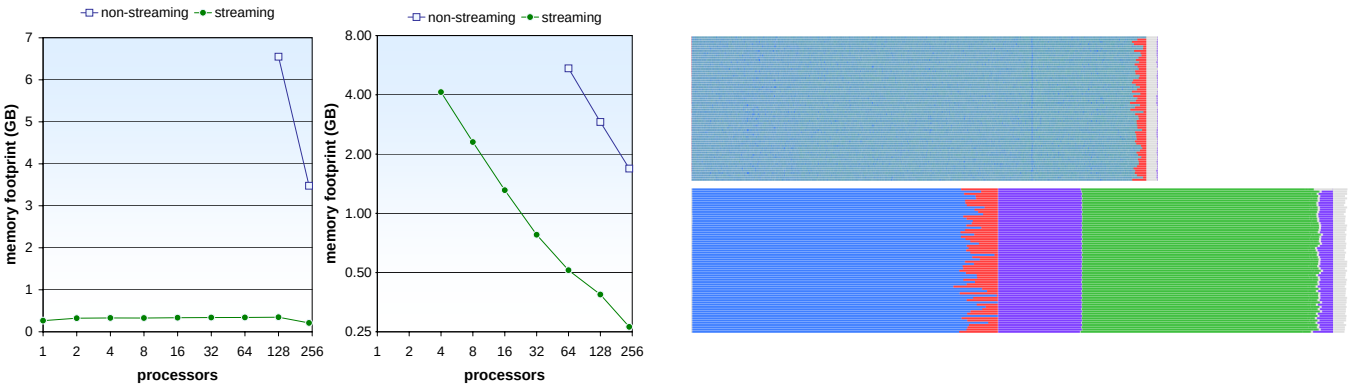

Figure 10: Comparison of streaming and non-streaming execution on the RT27 data set. Left: Per-processor maximum memory footprint for computing the gradient histogram. Middle: Memory footprint for isocontouring. Note that VisIt's rendering module accumulates the whole isocontour in-core before rendering, which hampers both the streaming and non-streaming approach. Top right: Gantt chart for streaming isocontouring on 64 processors. Bottom right: The non-streaming approach (349 sec) first loads all blocks (blue), communicates all ghost data (purple), and then does all the contouring (green), whereas the streaming approach (250 sec) interleaves these tasks block for block.

contains $100 \times 100 \times 100 = 1$ million double words this "sheet" of ghost data will roughly be $2 \times 10,000 \times 10,000$ double words. Hence, the total amount of memory for one block and the "sheet" will be around 771 MB.

When running on multiple processors our algorithm works best with cut-like assignments that result in few dependencies between processors and that are load balanced. A cut-like assignment is particularly efficien because it does, by construction, not violate an edge or corner rule. If we were guaranteed such assignments we could simplify the implementation significantl by removing the parts of the code dedicated to enforcing the edge and corner rules. However, even for random assignments with lots of dependencies or extremely unbalanced loads, our algorithm will complete the computation.

We certainly acknowledge that improvements could be made to VisIt to make its non-streaming mode demonstrate better results. For example, its non-streaming ghost zone communication routine could be modifie to not create two copies of the data set while it executes. And its derived quantity generator could be more efficien about discarding intermediate results. But we chose to take a "warts and all" approach. The two previous examples reflec the capability of the tool and are not atypical for production visualization tools. In addition, there are likely many more examples of changes that could make VisIt's memory footprint more efficient We believe that fact is actually a strength of our approach. Rather than auditing VisIt's well over one million lines of code and making many changes, we can make a single change (streaming data through the pipeline), and obviate the inefficiencies

The streaming and parallel algorithm we have described creates ghost data for regular structured grids. It would be desirable to have the same capability for block-structured or unstructured grids. However, this may require a different approach since it does not seem possible to establish an agreed-upon global travel direction for the faces that avoids communication between processors while avoiding deadlocks in the dependencies. A master-slave approach may be more suitable here.

References

[1] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, 2001.

[2] John Biddiscombe, Berk Geveci, Ken Martin, Kenneth Moreland, and David Thompson. Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1376–1383, 2007.

[3] Hank Childs. Architectural challenges and solutions for petascale postprocessing. *Journal of Physics, Conference Series*, 78, June 2007.

[4] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *IEEE Visualization*, pages 191–198, October 2005.

[5] Andrew W. Cook, William H. Cabot, Peter L. Williams, Brian J. Miller, Bronis R. de Supinski, Robert K. Yates, and Michael L. Welcome. Tera-scalable algorithms for variable-density elliptic hydrodynamics with spectral accuracy. In *ACM/IEEE Supercomputing*, page 60, November 2005.

[6] Andreas Gerndt, Bernd Hentschel, Marc Wolter, Torsten Kuhlen, and Christian Bischof. VIRACOCHA: An efficien parallelization framework for large-scale CFD post-processing in virtual environments. In *ACM/IEEE Supercomputing*, page 50, November 2004.

[7] M. Isenburg and P. Lindstrom. Streaming meshes. In *IEEE Visualization*, pages 231–238, October 2005.

[8] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[9] C. Charles Law, Kenneth M. Martin, William J. Shroeder, and Joshua Temkin. A multi-threaded streaming pipeline architecture for large structured data sets. In *IEEE Visualization*, pages 225–232, October 1999.

[10] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 163–169, July 1987.

[11] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimitis, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the IBM-SP system. In *ACM/IEEE Supercomputing*, page 70, November 1999.

[12] C. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientifi visualization and computer graphics. In *Visualization 2002 Course Notes*, 2002

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.