Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

High-Performance Mesh Partitioning and Ghost Cell Generation for Visualization Software

John Biddiscombe[1,2]

[1]Swiss National Supercomputing Centre, 6900 Lugano, Switzerland
[2]WMG, University of Warwick, UK.

Abstract

Post-processing large datasets efficiently in parallel requires good load balancing of geometry supplied to the visualization pipeline. When datasets are not pre-partitioned or cannot be read back from simulation output in well controlled pieces, it is necessary to perform a partitioning step before certain algorithms may be applied. Spatially sensitive operations such as resampling, smoothing or certain field advection/stencil algorithms require datasets/meshes to be contiguous and provide ghost cells so that artefacts do not occur at process boundaries where discontinuities occur.

This paper presents an integration of the mesh partitioning library Zoltan, into the Visualization Toolkit framework, VTK and the parallel visualization tool ParaView. The implementation allows seamless generation of well partitioned datasets using a user provided weighting and a selection of ghost cell generation options. The algorithms, and results obtained with the partitioning classes are presented with representative use cases that show an order of magnitude increase in performance compared to the off-the-shelf partitioning available previously, improving performance and reducing memory consumption/duplication.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.1]: Parallel Processing—Computer Graphics [I.3.2]: Distributed/network graphics—Software Engineering [D.2.2]: Software libraries—

1 Introduction and Motivation

The process of generating partitioned meshes in distributed applications is conceptually straightforward, though, in practice it can be difficult to do it efficiently and correctly, and for this reason, a number of dedicated partitioning packages exist that are widely used in the scientific community to create suitable data inputs for solvers [KK97,CP08]. In the field of visualization however, little attention has been paid to the process of efficiently generating clean, well partitioned meshes from data that has not been stored or prepared in advance in partitioned form. The two most widely used *off the shelf* visualization tools that specialize in parallel/distributed operation, ParaView [Hen05] and VisIt [CBW*12], provide their own utilities for partitioning and/or ghost cell generation, but they do not perform as well as tools designed and optimized specifically for the job (as we will show) and they do not provide as rich a set of options for customization as specialized tools do. In fact, although ParaView supports partitioning of data at runtime with generation of ghost cells, VisIt only supports generation of ghost cells from data that is loaded in partitioned form already.

Pipelines of the kind shown in figure 1, where load balancing is performed more than once, are particularly troublesome – in ParaView they are inefficient (particularly so with time dependent data on static meshes) and in VisIt they are not possible at all. One motivating example of a use case for such a pipeline is the selection of a region from a particle dataset (leaving data on a subset of processors and others empty), followed by an operation such as resampling that requires significant computation. In this case, the secondary filters will operate only on processors holding data whilst the empty ones go unused, wasting processing resources. Redistributing data a second time before performing the resampling allows all processors to participate in the calculation. To further improve load-balancing, we wish to support the weighting of data according to some user defined field (such as the mass or volume of particles).

DOI: 10.2312/pgv.20161181

Although the pipeline represented by figure 1 might seem uncommon, a second/final partitioning step is in fact added invisibly and automatically by ParaView when transparent rendering of geometry is enabled. The reason for this is that for correct alpha blending of geometry, it is necessary to sort objects according to their distance from the camera (the painter's algorithm [FvDFH90]) and when compositing scenes in parallel using a *sort-last* algorithm [MCEF94] one must ensure that regions from different processors do not intersect to avoid artefacts. A final partitioning step is therefore performed by the rendering engine to ensure that images are correct.

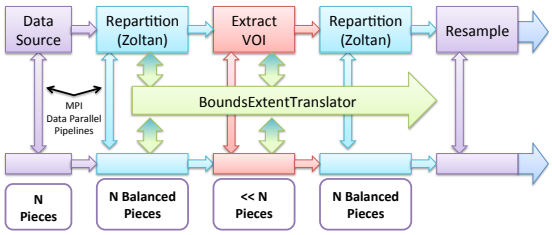


Figure 1: *Representation of data and information flow in a complex pipeline. Extracting a subset such as a VOI or feature from a dataset may leave some processes empty and further operations will not make use of all resources. Enabling multiple partitioning steps allows on demand selections to be redistributed for further processing using all available nodes.*

With the work presented in this paper we allow multiple partitioning steps when desirable, and eliminate them where possible by ensuring that meshes that have been previously partitioned by the user are not unnecessarily partitioned a second time by the rendering pipeline, this is achieved by adding partitioning metadata to the information passed along the pipeline. Additionally, we extend the types of dataset that can be partitioned to reduce data copies made by the visualization software – a dataset consisting of polygonal elements is transformed into an unstructured grid by ParaView's redistribution and must be transformed back to polygonal geometry prior to rendering, causing potentially a 2-fold increase in memory usage for the surface representation. The end of the *fixed-function* rendering pipeline is gradually removing the need to render only polygonal datasets, but for the time being this restriction remains for the majority of workflows.

The generation of ghost cells to ensure that gaps between dataset pieces are filled can be a relatively expensive operation that requires the computation of global node Ids and communication between processes to identify cells and cell neighbours. Ghost cell generation algorithms for grids (both unstructured or structured) work by identifying boundary cells and their neighbours and flagging them for exchange. This technique does not work for particle based datasets as there is no inherent cell connectivity (or neighbour information). Data loaded into a visualization tool generally loses any associated data structures that might have been used by the solver to accelerate boundary particle identification. Boundary information may also prove worthless when the number of nodes used for visualization is different from that used by the solver.

To address this problem, we have therefore implemented a ghost particle exchange based purely on the bounding boxes around process boundaries after the initial partitioning phase has identified the regions. We extend this algorithm to handle arbitrary cell types for any unstructured grid or polygonal mesh and thereby make ghost cell generation possible for all mesh types. Significantly, by making use of the initial partitioning information to identify ghost point and cells, we require no additional communication phase and ghost cell exchange can be performed during the same pass as the main partitioning step. This improves performance and does not impact the majority of visualization algorithms – when more robust ghost cell identification with full neighbour information is required the software can fall back to the implementation provided in VTK as explained later.

The contributions of this work are VTK filters that:

- Improve the efficiency of partitioning by integrating a dedicated tool into the pipeline.
- Add the ability to provide weighting of nodes to modify the load balancing step.
- Extend pipeline metadata to bypass unnecessary render driven partitioning operations.
- Reduce memory usage by handling polygonal dataset types as well as general unstructured grids.
- Provide simple ghost particle and cell generation for arbitrary cell types using a single pass bounding box tests only.
- Improve time dependent partitioning of static dataset by caching MPI send information for subsequent time steps.

2 Related Work

In the VTK library the workhorse of data partitioning is the `vtkDistributedDataFilter` (known as the D3 filter, for *distributed data decomposition*) which takes datasets of unstructured type (`vtkPolyData` or `vtkUnstructuredGrid`, see [SML98] for further information about VTK dataset types) and outputs data of type `vtkUnstructuredGrid` only. The class performs an MPI distributed Recursive Coordinate Bisection (RCB) algorithm on the input data and redistributes points/cells based on the algorithm described in [BB87], which is also used by the Zoltan [BDH*07, DBH*02] library itself (with some improvements). The algorithm is a spatial subdivision of the dataset in parallel with the emphasis being on generating a well balanced tree that minimizes communication costs. This differs from single node spatial decomposition where the emphasis is usually on producing a structure that enables quick location of an individual cell such as in [GJ10] or by using standard BSP or OctTrees. The performance of the D3 filter is

generally acceptable *when no other options exist*, but compared to modern well maintained libraries it compares badly and lacks features that are available in dedicated partitioning tools. There are several other widely used partitioning libraries available other than Zoltan, most notably, ParMETIS [KK97] and PT-Scotch [CP08], these tools are principally designed for graph/hypergraph partitioning and use a dual-graph representation of a mesh minimizing the cost of edge cuts representing communication between nodes. For the purposes of visualization, it is less important to minimize communication (between cells across process boundaries) as the algorithms are different from those in the solvers and for the most part we are visualizing/analyzing lower order (polygonal or tetrahedral) cells and the cost function of interest is simply a function of the number of cells/vertices in the algorithms used during post processing. It is therefore acceptable to partition the vertices of the mesh directly and consider the cells as additional properties to be carried along with data migration.

Zoltan was selected because it provides a geometric RCB partitioner and an API that is well suited to integration into toolkits like VTK. Zoltan operates by providing user defined callbacks to pass in the data to be partitioned, and once the partitioning is complete, further callbacks to pack individual elements into buffers so that they can be migrated to their destination process. This lends itself to the structure of VTK datasets where points, point fields, cells and cell fields are all stored in separate structures which must be traversed in order to pack data for sending. An additional consideration is that Zoltan provides an interface to both the METIS and Scotch libraries so that they can be used if the functionality were ever to be extended to graph partitioning (VTK supports datasets of type `vtkTable` and `vtkGraph` that are not currently supported, but could be included in some future implementation).

Ghost cells in VTK can be generated during partitioning by the D3 filter, however if a dataset is pre-partitioned by a reader/source, the filter `vtkUnstructuredGrid-GhostCellsGenerator` can be used to perform an extraction of bounding cells on each process and distribute them accordingly (but only for unstructured grids). In VisIt, a much more complex and comprehensive method of representing AMR, multi-block or multi-piece datasets exists, in the form of a Subset Inclusion Lattice (SIL) data structure that encodes the relationships between data blocks/pieces in the form of a graph/tree structure (see [BP89] for a further explanation), this allows a dedicated ghost generation filter to walk the tree of datasets and efficiently extract exactly the cells necessary for each adjoining dataset block to make up the missing data it requires for interpolation between blocks, though this can only be done for datasets that contain the SIL information (provided usually by the parallel reader).

Recent work in [HWB*15] describes an algorithm for computing connected components in parallel by using a union-find algorithm on the cells of interest for each of the distributed pieces and then combining this information to build a final set of connectivities. They describe both partitioning and ghost cell generation, but do not provide a general purpose implementation that can be reused for other filters. In their connected components implementation, ghost cells are used to identify those cells on the edges that abut other processes so that pieces overlapping boundaries can be matched up to test for connectivity. It is essential in this algorithm that cell Ids match across processes (global Ids) otherwise the same cell on two processes will be flagged as two different cells and a connected piece will be lost. Other algorithms (such as resampling operations) are generally indifferent to the presence of global Ids (cell or vertex) and so it is not always necessary to generate them.

It is worth noting that structured grids are not considered in this paper; the regular nature of grids makes ghost cell generation a trivial operation as one can find boundary cells and neighbouring ones by direct lookup. For out of core generation of structured grid ghost cells during streaming, the reader is referred to [ILC10] and for an excellent demonstration of the generation and use of ghost cells from AMR structured grids to generate crack free isosurfaces see [WCM12].

As previously mentioned, parallel sort-first compositing (with transparency) requires front to back sorting of data both locally on a node, and when compositing images from each renderer. The process is the same as that for traversing a BSP tree (see [GC91]) to perform hidden surface removal and is used by IceT [MKPH11], the engine responsible for combining images from render-nodes in ParaView. Zoltan provides a mechanism to query the BSP cuts made during partitioning and IceT provides a mechanism to pass in the BSP tree for ordering the compositing operations. We make use of these facilities to direct ParaView's rendering: an early version of the implementation described in this paper was used in [HBB*13] to compare the performance of transparent parallel rendering in ParaView with a custom rendering tool on large neuron datasets of polygonal meshes, however no treatment of ghost cells was possible (or necessary) and only a single datatype was handled by the software. Another use case for large scale transparent rendering comes from the Astrophysics community where large particle datasets are rendered using a photorealistic rayrace/splat algorithm such as found in [JKR*10]. The algorithm used assumes that gaseous absorption and emission are equal which simplifies the blending operation to make it associative and therefore particles may be composited out of order. To correctly model gaseous absorption (dust clouds etc), particles need to be sorted, not just on a node, but across nodes. Currently no large scale renderers correctly implement absorption and it is a motivating example for future use of this work.

The Zoltan2 library, a rewritten C++ implementation of Zoltan contains new algorithms such as the Multi-Jagged (MJ) partitioner that has even better scalability and performance (see [DRDC16]) than the RCB algorithm and we

provide options to select/use this algorithm. Unfortunately, Zoltan2 does not provide a method (at the time of writing) to expose the BSP tree cuts and so we cannot use the MJ implementation when compositing transparent renderings and it has not therefore been included in this paper. An alternative method for compositing transparent images in parallel would be a distributed implementation of a fragment sorting depth peel algorithm such as the one in [LWXW09], but to date this has not been attempted as the network traffic would be prohibitive for complex scenes.

3 Mesh Partitioning with Zoltan

Partitioning using Zoltan is broken into a sequence of operations as follows (ignoring ghost cells for the time being)

1. Passing a description of geometry to Zoltan
2. Executing the domain decomposition/partitioning algorithm by calling `_LB_Partition` †
3. Migrating data from the source to destination nodes according to the information returned by the partitioning step

where the migration step may be performed by Zoltan automatically and immediately after partitioning, or manually by the user (by calling `_Migrate`). It is important to note that the partitioning (step 2) does not transfer any of the user's data between nodes (other than that required for the computation of the domain decomposition itself). Data transfer only takes place during the migration step using lists of objects (vertices/cells/other) that are to be exchanged between processors.

Although VTK consists of many dataset types, we are concerned here only with polygonal and unstructured grids, a special case of which is particle data, since it contains only 0 dimensional points and may be held in `vtkPolyData` or `vtkUnstructuredGrid` container datasets. A specialized `vtkParticlePartitionFilter` has therefore been created to handle data of this kind - the method for particle partitioning is presented first and then generalized to other cell types.

3.1 Particle Partitioning

When no ghost particles are required, the steps involved are as outlined in 1-3 above, with the result of the partition information coming in the form of a pair of lists (on each process), one containing the point Ids that need to be exported from the process, the other being the destination rank for each of the aforementioned Ids. Given this pair of *export* lists {Ids,Ranks}, Zoltan provides a convenience function `_Invert_Lists`, a collective operation that must be called by all processes that takes the *export* list pair from each process and generates a count of the inverse operation, how many Ids will be received by each node. On completion of this call, each rank knows not only how many Ids it will export but also how many it will import, making it possible to correctly allocate the right amount of space for the final set of points on each node, a crucial optimization that improves performance and memory use.

3.1.1 Ghost Particles

The pair of *export* lists returned from `_LB_Partiton` does not have to be passed to `_Invert_Lists` directly, or indeed at all, the user is free to create completely new lists and use them instead. This provides an opportunity to insert new steps between steps 2 and 3 to locate ghost particles and modify the *export* lists before inverting them and migrating data. Resampling/interpolating particle data to produce contours as shown in figure 2 requires neighbour lists that include ghost particles for smooth boundaries between processors. The thickness of the ghost region may be determined from some property such as the Kernel size in the case of Smoothed Particle Hydrodynamics (SPH) data, and the RCB bounding boxes for each partition are available after the `_LB_Partition` step completes, so one can easily add the desired ghost layer thickness to each process bounds to find the region in which ghost particles must be identified.

The following (straightforward) algorithm is used to identify ghosts and track ownership of points. An important point being that each point should be *owned* by one rank only and exist as a ghost on any others it intersects the $(bounds + halo)$ region of.

Algorithm 1 iterates over all points that are on the current process and tests them against the bounding boxes of other processes using a BSP locator built from the halo regions of each process, if they lie inside the halo region of another process (or processes) and have not already been marked as belonging to the remote process during `_LB_Partition`, they must be ghosts for that rank. If they are flagged as ghosts, but were *not* previously marked as belonging to this rank, then they must be added to a list of local points to *keep* (as well as send). The total space required for all points is given by

$$N_{Final} = N_{Original} + N_{Import} - (N_{UniqueExport} - N_{Keep}),$$

where $N_{UniqueExport}$ is given by the sum of the number of points designated for export by Zoltan and those flagged as ghost exports – after duplicate Ids have been removed – since a point exported as a ghost to multiple processes and/or exported to be *owned* by another *must only be counted once*. Once the total space required is known, it can be allocated, then points to *keep* are copied into their new output locations and `_Migrate` can be called using the space offset by the kept points and using the *export* list created by combining the original partition and the new ghost assignments. Before exchanging data one must ensure that any processes with 0 points have arrays setup for receiving field data, so an extra

† For brevity, and to reduce repetition, function names such as `Zoltan_LB_Partition` have been abbreviated to `_LB_Partition` with a single leading *underscore* in place of `Zoltan_`

Input: ExportList: List of exports {Ids,Ranks}
Output: GhostList: List of exports ghost {Ids,Ranks}
Output: LocalIdsToKeep: List of Ids marked for export, but also to be kept locally

```
IdToProcessList ← {myRank,myRank,myRank ... }
foreach {Id,Rank } in ExportList do
 | IdToProcessList [Id] ← Rank
end
foreach Rank ≠myRank do
 | HaloBox ← Inflate (PartitionBox,GhostSize)
end
Build BSP tree from Halo Boxes
foreach {Point,Id } do
 | foreach Rank with Overlaps (Point,HaloBox) do
 | | if IdToProcessList [Id ]≠Rank then
 | | | GhostList ← Add {Id,Rank }
 | | | if IdToProcessList [Id ]=myRank then
 | | | | LocalIdsToKeep ← Id
 | | | end
 | | end
 | end
end
```

Algorithm 1: Identifying ghost points and points to keep. Note that the *foreach* loop orders can be swapped for better cache coherency

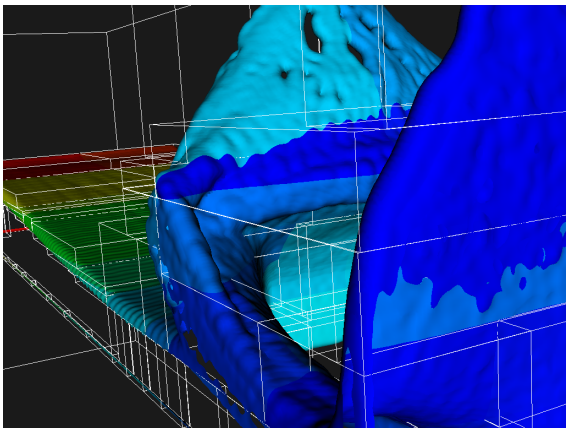
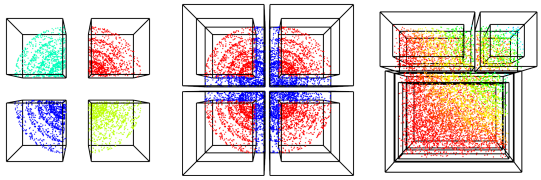


Figure 2: *Resampled and contoured SPH particle dataset with bounding boxes of process regions shown. Crack free contours between processes are ensured by setting the ghost region overlap to the SPH kernel size.*

`AllGather` is performed for this initialization. Field data array pointers are then cached so that Pack and Unpack operations triggered by the Zoltan callbacks during migration are efficient.

Figure 3 (c) shows the result of partitioning a uniform distribution of particles when a user supplied weight array is specified. This requires no additional logic in the filter, passing an extra pointer to `_LB_Partition` is sufficient; the returned *export* lists and bounding boxes) reflect the weighted partition and the effects are propagated through automatically. Providing weights via a user supplied field allows finer control of load balanacing and can be used to place fewer large particles and more smaller ones per process (or vice versa).

Figure 3 (a) and (b) show results of partitioning with and without ghost particles enabled – blue ghost points around the centre of 3 (b) must be marked as `DUPLICATEPOINT` on all ranks that have copies as a ghost and be unmarked on the rank that *owns* it. (Implementation note: points are marked as ghosts prior to sending (where applicable), and then unmarked afterwards if they are *owned* by the sending process). The implementation of algorithm 1 guarantees that every point will appear once and only once as a non ghost and $N-1$ times as a ghost or duplicate point – providing points are unique on the input. If duplicated points are present in the input, they will be passed through (the excpetion being points already marked as ghosts on the input which can be dropped).



(a) Simple (b) Ghosts (c) Weights

Figure 3: *Partitioning point based data (partitions separated spatially for illustration). (a) Random points in shells partitioned onto 4 processes. (b) The same points with a halo region around each process where ghost points are duplicated. (c) A Cube of uniformly distributed points weighted towards the upper right corner.*

3.2 Cell Partitioning

Partitioning of meshed datasets follows the same basic outline as for particle data with the major difference being that after points have been partitioned cells that are made up from them must be distributed. Cells along process boundaries may have their vertices assigned to multiple processes but must be *owned* by one and only one and if necessary, duplicated as ghosts on other processes. Fortunately, any cell whose vertices belonging to multiple ranks *must* be a boundary cell by definition which makes the task of identifying them straightforward.

A simple classifiction of cells can be made based on the partition assignments of their points

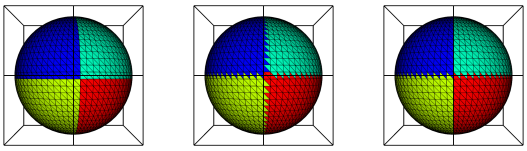
- `Local`: All cell points remain on the local process
- `Remote`: All points assigned to a single remote process
- `Split`: One (or more) local, rest on remote process(es)
- `Scattered`: All remote, on multiple processes

This classification is used to help decide which process should *own* the cell and if it needs to be exported. Cells of type `Local` are not exported whilst `Remote` cells must

be, with no additional checks for either necessary. Cells that are `Split` can either be exported or kept – but if kept, any points marked for *export* must be added to a list of points to *keep* and if exported any points marked as local must be marked for *export* too – otherwise the cell will be incomplete on the receiving (*owning*) process. Likewise, cells of type `Scattered` must have *all* points sent to the receiving process (not just those already marked for the destination). Note that points that are *unused* when a cell is sent to one process but other points go elsewhere cannot be dropped completely as they will be part of a neighbouring cell.

Cell ownership produces subtle differences that can normally be ignored, but might be of interest, particularly if ghost cells are unwanted for some reason (many existing algorithms in VTK do not correcly handle ghost cells/points and their presence can affect results). The effects of cell ownership possibilites can be seen in figure 4 and are:

- `First`: The process that owns the first point
- `Most`: The process owning the most points
- `Centroid`: The process overlapping the cell centroid



(a) First (b) Most (c) Centroid

Figure 4: *Cell ownership assignment. The choice of mode produces subtle effects which can usually be ignored, but might be desirable in certain circumstances.*

Usually assigning the cell ownership based on the first point produces a somewhat random ownership of boundary cells, but as can be seen in figure 4 (a), when data comes from a structured source, it can give a good distribution. Assignment based on centroid is the default mode of operation.

Assignment of cells for *export* from their points is outlined in algorithm 2. Using this algorithm, tracking the assignment of points and cells in a single pass through the cell list is possible. This makes it possible to build up the final allocation of all points and cells in one go prior to migration – though the algorithm does require a considerable number of internal flags to note if a cell is being kept/sent or both and checks to ensure that all points belonging to all cells are sent where they belong.

3.3 Ghost Cell Generation

Extending algorithm 2 to handle ghost cells varies slightly depending on the ghost cell mode selected. The types that are available are:

- `None`: no ghost cells are generated
- `Boundary`: cells that are `SPLIT` or `SCATTERED` must straddle a boundary and be copied on all proesses that share the points.

Input: ExportPoints: List of exports {Ids,Ranks}
Input: BoundaryMode: enum {First, Most, Centroid}
Output: ExportCells: List of exports {Ids,Ranks}
Output: LocalPointsToKeep: Points for export, but also keep

```
// PointDest is process owning point Id
Setup IdToProcessList as in algorithm 1
foreach Cell do
    PointProcessCount ← Count points per process
    // cell local/remote/split/scattered
    CellType ← Classify (Cell, PointProcessCount)
    // which process will own cell
    CellDest ← Assign (Cell, BoundaryMode)
    Mark cell for export to CellDest
    foreach Point in Cell do
        if PointDest≠CellDest then
            Mark point for export to CellDest
            if PointDest=LocalRank then
                Mark point for keep
            end
            if CellType SPLIT or SCATTERED then
                foreach Rank in PointProcessCount do
                    Mark point for export to Rank
                    Mark cell for export to Rank
                    if CellDest=LocalRank then
                        Mark cell for keep
                    end
                end
            end
        end
    end
    // for point and cell export/keep lists
    Remove duplicates {Id,Rank} from export list
    Remove duplicates from keep list
end
```

Algorithm 2: Algorithm for distributing Cells and their points from the export lists supplied by partitioning. The bookkeeping of points to send and keep is rather complex and simplified here to the main concept: all points for exported cells must be marked and any shared by cells not exported must have copies kept

- `BoundingBox`: The cell is copied to all processes whose Halo region overlaps the cell.

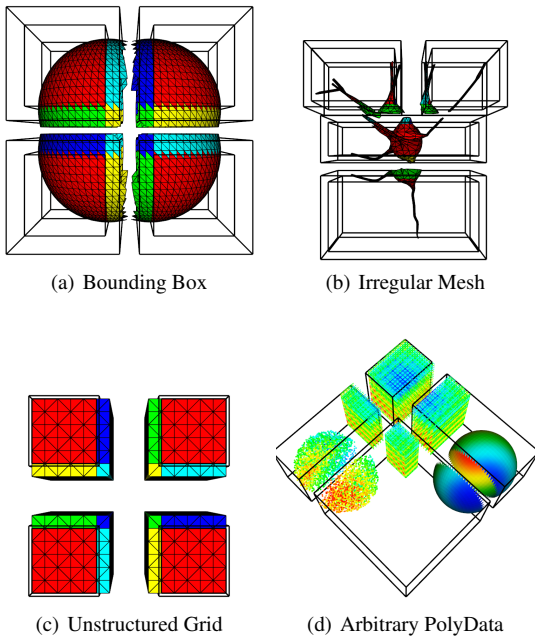
Note again that like points, cells are *owned* by one and only one process and copies are sent to others as ghosts where they are marked as `DUPLICATECELL`. When `Boundary` mode is selected, only cells of type `SPLIT` or `SCATTERED` need to be flagged as ghosts, but only on the processes that are not receiving the cell already. Since these cell types are already handled by algorithm 2 there is just one extra check to make a copy of the cell on any processes not yet receiving it.

For `BoundingBox` mode, things are more complicated. For each rank, we inflate the bounds by an amount equal

to the requested halo size and must then test *all* cells against the bounds (since we do not know how large the halos might be – though usually of the order of the maximum cell size). To accelerate this process, as before, we use a BSP locator created from the halo bounds of each rank and then classify cells as ghosts if they overlap a rank bounds and are not already destined for that rank (as they might be due for migration anyway as non ghost cells of type `Remote`). As can be seen in figure 5 (a), large halo regions can lead to the same cells being sent to several processes. All points of each ghost cell must be flagged for sending (or keeping) to all overlapping ranks in order to build up the final *export* list. However it is possible to do this in a single pass through of all the cells. The advantage of this approach is that all point exchanges can be done in one migration step and then all cells in a second. This is not the case for a true neighbour based ghost exchange as neighbour cells may originate on different processes and there is no way of knowing which are truly neighbours until a first exchange of points or global Ids has taken place and comparisons made. Figure 5 illustrates a range of dataset types and ghost cell options that can be handled by the implementation. The algorithm/implementation guarantess that if the input data has no duplicated cells, then each cell will appear once and only once in the output as a normal cell and $N-1$ times as a ghost (where N represents the number of halo regions overlapping the cell). Points may be duplicated in the output if they come from different ranks as we make no comparison of global Ids. When this matters, then the Mesh partition filter may have ghost cells disabled and the VTK ghost cell generator may be used (see section 5.3).

4 Pipeline Extensions

The pipeline mechanism of VTK allows filters to add meta-data to the outputs they produce and to request it from data they consume (see [BGMT07] for more information). When the rendering engine composites images from parallel sources, it must sort them from back to front. For structured grids, the sort order is known and internally, VTK/-ParaView creates a KdTree object to represent the spatial arrangement of the pieces. For unstructered data, a distribution phase is used to generate the KdTree on demand, but if the tree is already known it could be used if the information was available to the renderer. The particle and mesh partition filters both inherit from a ZoltanBasePartitionFilter and this adds an information key to the data output which holds a `vtkBoundsExtentTranslator` object that is passed downstream (refer to figure 1). The Bounds translator holds a reference to the KdTree generated from information provided by Zoltan and the translator also provides a mechanism to compute structured indices from the bounding boxes of each piece of data (this can for example be used by a resampling filter to compute the correct structured dimensions of a piece when each piece is a different size). By modifying the `vtkGeometryRepresentation` class (that is responsible for rendering unstructured data) to look for a Bounds translator, we can detect when data has been partitioned upstream and bypass the domain decomposition step, passing the KdTree directly to the rendering engine.



(a) Bounding Box (b) Irregular Mesh

(c) Unstructured Grid (d) Arbitrary PolyData

Figure 5: *Partitions of various mesh types and ghost cell options. In (a) ghost cells are coloured according to the rank they are owned by. In (b) a compex mesh is partitioned with regions of denser polygons receiving smaller bounds. (c) shows Unstructured grid partitioning with only boundary cells as ghosts and (d) demonstrates a mixture of points/lines/polygons partitioned (without loss of data type).*

4.1 Time Dependent Data

When animating scalar/vector fields on a dataset that has fixed geometry (such as the neuron circuit described below), the pipeline will update all filters that receive modified datasets. This includes partitioning filters – both user added and those from the rendering stage. The abstraction of the partition filter to make use of zoltan `_Migrate` for point/-cell migration steps makes it easy to cache the final *export* lists in the partition filter and uses these to resend field arrays on timesteps without recomputing the main partition. In section 5.2 we show the benefits of this and the use of KdTree information to accelerate rendering.

5 Performance

To test the performance of the new `vtkParticle-PartitionFilter` (PPF) and `vtkMeshPartition-Filter` (MPF), comparisons are made with the D3 filter from VTK and the `vtkUnstructuredGridGhostCells-Generator` (vUGGCG). As we are principally interested in the raw performance (ignoring multithreading optimizations

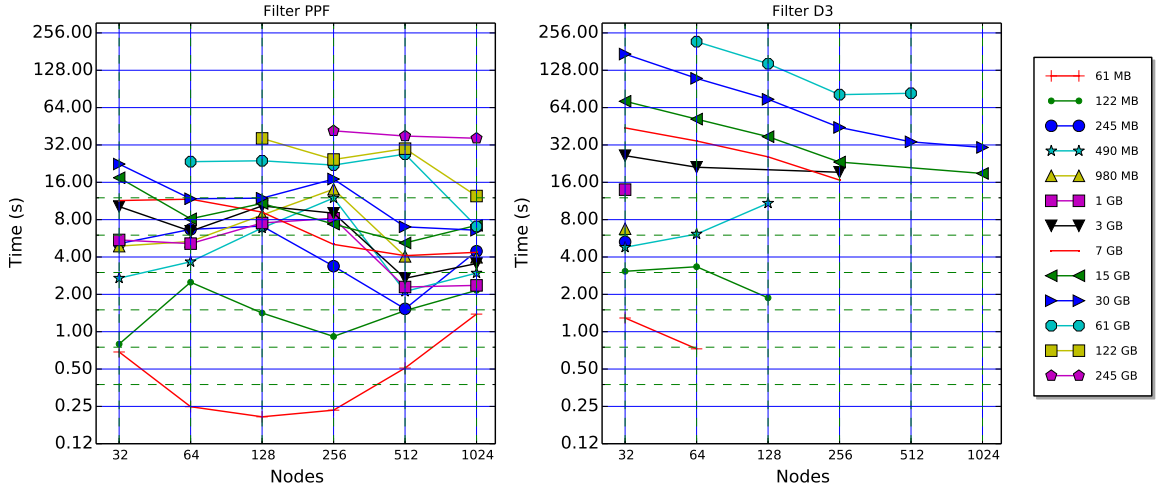


Figure 6: *Timing comparison between the Particle Partition Filter and D3 filters on time to partition datasets of different sizes on different node counts. Note that the D3 filter was unable to produce valid output for smaller datasets on larger node counts, and larger data on any node counts. Results for the larger datasets are only available on larger node counts due to memory pressures.*

and network effects), all tests have been run as single threaded single rank per node processes (the Zoltan2 version of the new filters supports multi-threading but this has not been included here).

5.1 Strong-Scaling : Particles

Figure 6 shows the time taken to partition a series of particle datasets ranging from 61MB to 245GB. The difference in performance between the PPF and D3 is obvious. Unfortunately, results for D3 are missing for a number of test combinations due to failures and timeouts of the filter (undiagnosed). The PPF was unable to partition the largest datsets on smaller node counts, but other than that completed virtually all tests in times that are an order of magnitude faster than the D3 filter. Note also that the PPF filter supports ghost particles where D3 does not.

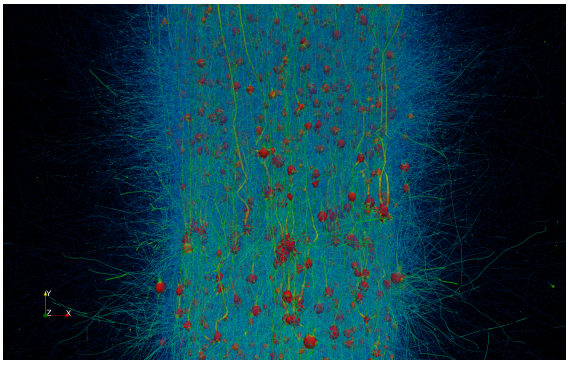


Figure 7: *5K neuron circuit of* 10^9 *triangles rendered with transparency on 1024 cores.*

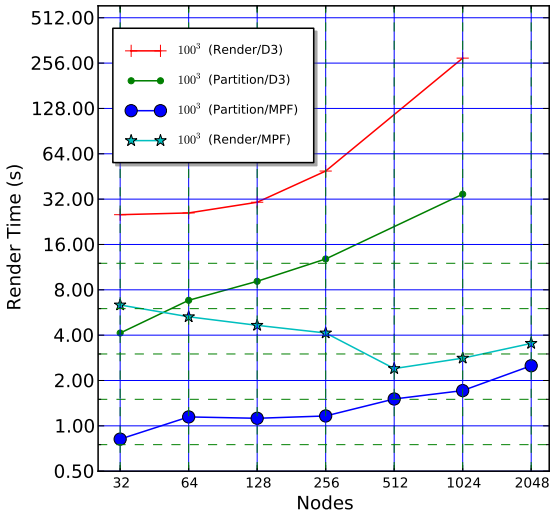


Figure 8: *Render time for transparent surface geometry extracted from a distributed dataset of* 100^3 *hexahedra triangulated (per node) and then shrunk to produce* $6 * 100^3$ *disconnected tetrahedra. The MPF passes KdTree information to the renderer and avoids an additional partitioning step.*

5.2 Mesh Partitioning and Rendering

A comparison of the D3 and MPF yields results that are the same as for D3/PPF with the MPF outperforming D3 by a significant margin. Rather than present a similar graph, we instead present timings of a render pipeline that is motivated by the use case illustrated by figure 7. This image shows a neuron circuit of 5K cells comprising 10^9 triangles with a very high depth complexity as the neurons form a dense

interwoven mesh. Visualizing fields in the centre is made simpler by controlling the transparecy of uninteresting items that should be hidden. A pipeline that mimics this workflow is as follows

Wavelet→Tetrahedralize→Partition→Shrink→Render

with transparency enabled for the final render stage. The `Wavelet` source generates strucured image data of arbitrary dimension in parallel, passing this through `Tetrahedralize` converts the data to an unstructured grid (with 6 times as many cells), the `Partition` filter redistributes the cells in parallel as we would do after reading a mesh/dataset from file. A `Shrink` filter is added to separate the cells so that the final render stage has many surface polygons to handle. Enabling transparency causes the render pipeline to redistribute data a second time if necessary. We can control the number of cells by varying the resolution of the wavelet source.

Figure 8 shows the time to partition and render the wavelet using the MPF and D3 filters. The MPF filter partitions the largets datasets ($2048 * 100^3$) in a few seconds, passes the KdTree downstream and rendering takes slightly longer. The D3 filter takes several minutes to perform the same actions – for 1024 nodes the first partition time is 30 seconds, but after the shrink filter operates, the second partition takes around 2 minutes. Even though the Shrink filter occurs after partitioning, this does not invalidate the decomposition and the renderer could still use the KdTree. We can generalize this by saying that any filter that does not alter the domain decomposition (i.e. most filters in VTK) can pass the KdTree downstream without invalidating the decomposition and triggering a re-execution of the MPF/PPF filters.

Figure 9 shows the peak memory usage on a node during the render benchmark of figure 8. Wherease the MPF pipeline uses almost constant memory per node as the job size increases (but data size per node remains constant), the D3 pipeline memory usage grows quickly as the job size increases. At 1024 nodes the memory use is 4 times larger for D3 than MPF and at 2048 nodes, the D3 filter is unable to operate. It is clear that the MPF pipeline not only operates faster but can handle much larger datasets.

When combining the KdTree information optimization with caching of *export* lists for time dependent field data, the time to animate the neuron models was reduced from over 3 minutes per time step to under 5 seconds as scalar values change.

5.3 Ghost Generation

In order to benchmark the ghost cell generation a pipeline of

Wavelet→Tetrahedralize→Partition

with ghost cells enabled using `BoundingBox` mode in MPF, and

Wavelet→Tetrahedralize→Partition→vUGGCG

with ghost cells disabled in MPF, but ghost cells generated instead by the standard VTK ghost cell generator. We time

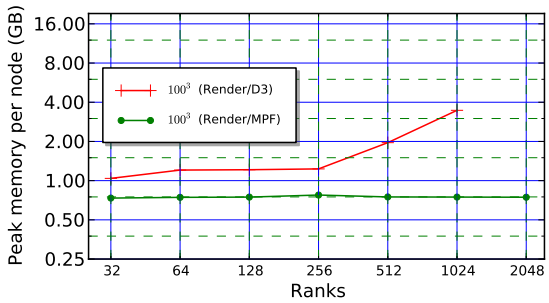


Figure 9: *Peak memory use per node during rendering surface geometry desribed in figure 8*

the MPF when generating ghosts and again when not generating them and use the differnce as a reasonable approximation to the ghost cell time (since the ghost cells are generated and exchanged during the migration step, separating out the times exactly is not possible).

The results in figure 10 show that the VTK ghost cell filter performs extremely well, generating ghosts for $6 * 2048 * 100^3$ nodes in seconds and the MPF filter taking roughly half the time. The VTK ghost cell filter performs point merging of duplicated points and is therefore expected to take longer. Importantly, we see here that it is quite possible to mix the two classes and benefit from both fast partitioning and robust ghost cells when needed.

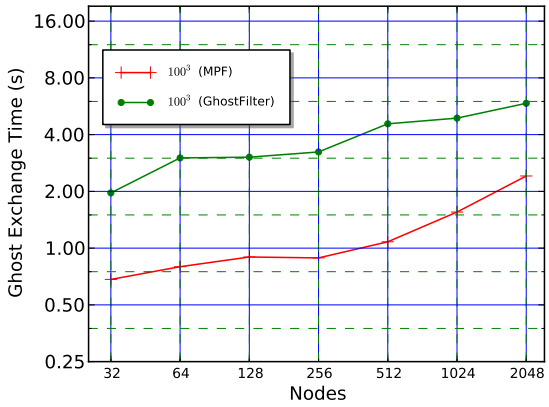


Figure 10: *Comparison of ghost exchange times using MPF BoundingBox method and ParaView's dedicated GhostCellsFilter on $6*100^3$ tetrahedra per node.*

6 Conclusion

The partitioning software presented improves performance compared to the existing VTK implementation by an order of magnitude for all cases of dataset partitioning, reduces memory use by preserving polygonal datatypes as well as unstructured grids and adds support for particle datasets with ghosts. Time dependent dataset processing can be significantly accelerated and datasets of large size

that were not previously possible to visualize with transparency can now be handled with relative ease. The filters described in this work may be integrated with any existing packages that make use of parallel VTK and enable better load-balancing for arbitrary pipelines. A fast ghost cell generation algorithm has been presented that handles all unstructured cell types and the partitioning can interoperate with the more robust existing ghost cell implementations when required. The software developed is made freely available under a permissive Open Source license.

Software

The software described in this paper is available for download from https://github.com/biddisco/pv-zoltan.

Acknowledgements

The authors would like to thank Abhishek Kumar and the Google Summer of Code project for providing a valuable stimulus to this work and Jean-Guillaume Piccinali for supplying data. The work received funding from the Human Brain Project from the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement no. 604102 (HBP).

References

[BB87] BERGER M. J., BOKHARI S. H.: A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput. 36*, 5 (May 1987), 570–580. 2

[BDH*07] BOMAN E., DEVINE K., HEAPHY R., HENDRICKSON B., LEUNG V., RIESEN L. A., VAUGHAN C., CATALYUREK U., BOZDAG D., MITCHELL W., TERESCO J.: *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; Developer's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4749W. 2

[BGMT07] BIDDISCOMBE J., GEVECI B., MARTIN K., THOMPSON D.: Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1376–1383. 7

[BP89] BUTLER D. M., PENDLEY M. H.: A visualization model based on the mathematics of fiber bundles. *Computers in Physics 3*, 5 (1989), 45–51. 3

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RÜBEL O., DURANT M., FAVRE J. M., NAVRÁTIL P.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Oct 2012, pp. 357–372. 1

[CP08] CHEVALIER C., PELLEGRINI F.: Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput. 34*, 6-8 (July 2008), 318–331. 1, 3

[DBH*02] DEVINE K., BOMAN E., HEAPHY R., HENDRICKSON B., VAUGHAN C.: Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering 4*, 2 (2002), 90–97. 2

[DRDC16] DEVECI M., RAJAMANICKAM S., DEVINE K., CATALYUREK U.: Multi-jagged: A scalable parallel spatial partitioning algorithm. *Parallel and Distributed Systems, IEEE Transactions on 27*, 3 (March 2016), 803–817. 3

[FvDFH90] FOLEY J. D., VAN DAM A., FEINER S. K., HUGHES J. F.: *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. 2

[GC91] GORDON D., CHEN S.: Front-to-back display of bsp trees. *IEEE Comput. Graph. Appl. 11*, 5 (Sept. 1991), 79–85. 3

[GJ10] GARTH C., JOY K.: Fast, memory-efficient cell location in unstructured grids for visualization. *Visualization and Computer Graphics, IEEE Transactions on 16*, 6 (Nov 2010), 1541–1550. 2

[HBB*13] HERNANDO J. B., BIDDISCOMBE J., BOHARA B., EILEMANN S., SCHÜRMANN F.: Practical parallel rendering of detailed neuron simulations. In *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2013), EGPGV '13, Eurographics Association, pp. 49–56. 3

[Hen05] HENDERSON A.: *ParaView Guide, A Parallel Visualization Application*. Kitware Inc., 2005. 1

[HWB*15] HARRISON C., WEILER J., BLEILE R., GAITHER K., CHILDS H.: *Topological and Statistical Methods for Complex Data: Tackling Large-Scale, High-Dimensional, and Multivariate Data Spaces*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, ch. A Distributed-Memory Algorithm for Connected Components Labeling of Simulation Data, pp. 3–19. 3

[ILC10] ISENBURG M., LINDSTROM P., CHILDS H.: Parallel and streaming generation of ghost data for structured grids. *Computer Graphics and Applications, IEEE 30*, 3 (May 2010), 32–44. 3

[JKR*10] JIN Z., KROKOS M., RIVI M., GHELLER C., DOLAG K., REINECKE M.: High-performance astrophysical visualization using Splotch. *Procedia Computer Science 1*, 1 (2010), 1775 – 1784. {ICCS} 2010. 3

[KK97] KARYPIS G., KUMAR V.: A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, March 14-17, 1997, Hyatt Regency Minneapolis on Nicollet Mall Hotel, Minneapolis, Minnesota, USA* (1997). 1, 3

[LWXW09] LIU B., WEI L.-Y., XU Y.-Q., WU E.: Multi-layer depth peeling via fragment sort. In *Computer-Aided Design and Computer Graphics, 2009. CAD/Graphics '09. 11th IEEE International Conference on* (Aug 2009), pp. 452–456. 4

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 4 (July 1994), 23–32. 2

[MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 25:1–25:10. 3

[SML98] SCHROEDER W., MARTIN K. M., LORENSEN W. E.: *The Visualization Toolkit (2Nd Ed.): An Object-oriented Approach to 3D Graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. 2

[WCM12] WEBER G., CHILDS H., MEREDITH J.: Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (amr) data. In *Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on* (Oct 2012), pp. 31–38. 3