VLSV File Format

This page contains information related to VLSV file format.

Table of Contents

VLSV File Format

Introduction

VLSV file format was created for writing domain-decomposed meshes, and variables defined on their zones, efficiently in parallel using message passing interface (MPI). Here is a list of features:

- Data written in parallel using collective MPI calls
- Parallel visualization with VisIt
- Scalability tested up to tens of millions of zones per mesh
- Multiple meshes per file
- Stretched Cartesian, Cylindrical, and Spherical coordinate systems
- Arbitrary coordinate scaling for visualization purposes (for example, from meters to Earth radii)
- User-defined axis labels and units

Multi-Domain Meshes

A multi-domain mesh is your vanilla uniform Cartesian mesh consisting of (Nx,Ny,Nz) zones and (Nx+1,Ny+1,Nz+1) nodes in (x,y,z) directions. The twist is that zones are decomposed into independent domains, as illustrated by the coloring in the figure below. Zones are written out in any specific order, and some of them may not even exist (you can have holes in the mesh).

Typically a parallel simulation is run with multiple MPI processes that are responsible for performing computations on variables defined in zones of their respective domains. This requires that MPI processes keep local copies of some zones defined in neighboring domains, as illustrated by the uncolored zones in the figure below. Here such zones are called ghost zones, or simply ghosts. Parallel visualization requires that each domain can be processes independently, thus each MPI process must write out its own domain plus ghost zones.

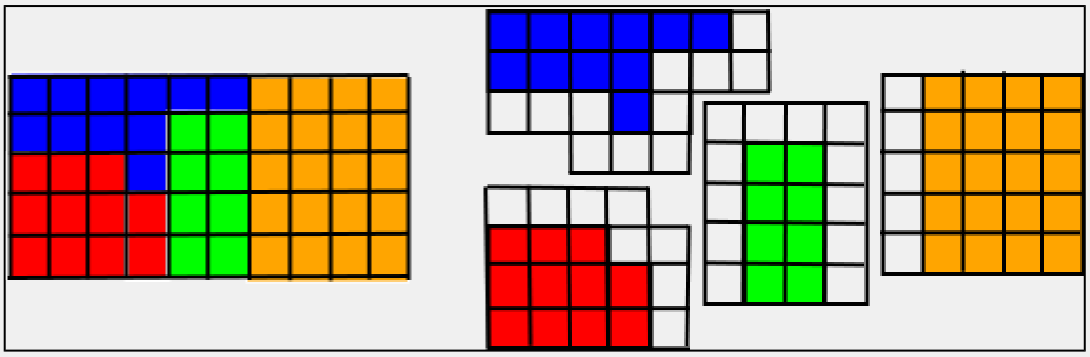

FIGURE: Example of a Cartesian mesh decomposed into four domains (red, blue, green, orange) that are each owned by different MPI processes. Pictures of the right show ghost zones (white).

Rationale: Strictly speaking mesh can be drawn without information on ghost zones. However, many plot types in VisIt are unable to cross domain boundaries correctly without variable data in ghosts.

A mesh is defined by its bounding box, i.e. the "global" simulation mesh. A domain is simply an unordered collection of zone (local+ghost) global ID numbers. Zones have unique global ID numbers that are calculated according to the C convention,

```
global ID = k*Ny*Nx + j*Nx + i,
```

where (i,j,k) are the zone indices, as illustrated in the figure below. Replace Nx by (Nx+1) and Ny by (Ny+1) in the formula above to get node global indices. Node and zone (i,j,k) indices are meshes logical coordinates. Mapping into physical coordinates depends on the coordinate system used.

When mesh is visualized node coordinates are obtained by a table lookup, i.e. you are required to provide three arrays of sizes Nx+1, Ny+1, and Nz+1, containing nodes' x, y, and z coordinate values. For example, assume that these arrays are called xcrds, ycrds, and zcrds. Assuming Cartesian coordinate system, (x,y,z) coordinates of a node having indices (i,j,k) are xcrds[i], ycrds[j], and zcrds[k]. This is how VLSV format implements stretched meshes -- zone sizes do not need to be uniform. Strictly speaking, values in arrays xcrds, ycrds, and zcrds depend on the coordinate system as discussed here.

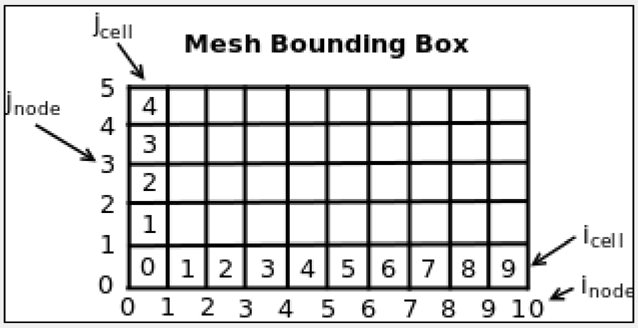

FIGURE: Two-dimensional example of a mesh bounding box. Zones are labeled with their (i_cell, j_cell) indices, and nodes with (i_node, j_node) indices. Note that the number of nodes per coordinate direction is number of zones plus one.

Your simulation may of course use a different numbering scheme for nodes and zones. When writing VLSV files, however, zones must be numbered as presented above.

Coordinate Systems

All meshes in VLSV format are logically Cartesian, i.e. mesh is defined by its Cartesian bounding box, and node coordinates can be calculated from (i,j,k) indices. With Cartesian coordinate system the logical coordinates coincide with physical coordinates,

```
float x = xcrds[i];
float y = ycrds[j];
float z = zcrds[k];
```

However, in case of Cylindrical and spherical coordinate systems the logical coordinates must map into (r,phi,z) and (r,theta,phi). VLSV visualization plugin maps these into Cartesian coordinates for Visit,

```
// Cylindrical
float r_cyl   = xcrds[i];
float phi_cyl = ycrds[j];
float z_cyl   = zcrds[k];
float x_cyl = r_cyl * cos(phi_cyl);
float y_cyl = r_cyl * sin(phi_cyl);
float z_cyl = z_cyl;

// Spherical
float r_sph     = xcrds[i];
float theta_sph = ycrds[j];
float phi_sph   = zcrds[k];
float x_sph = r_sph * sin(theta_sph) * cos(phi_sph);
float y_sph = r_sph * sin(theta_sph) * sin(phi_sph);float z_sph = r_sph *
cos(theta_sph);
```

HOWTO: Write a Multi-Domain Mesh

Here is a checklist (for each domain):

- Separate zones into locals and ghosts. Local zones are always written first, only mesh data needs ghost zones.
- Figure out an order in which to write out zones.
- At this point you have local IDs for all zones (=the order they are written to file).
- Synchronize local IDs with neighbor processes.
- Each process must know the domain (=MPI rank) where the ghost zone belongs to, and local ID of ghost zone in that domain.

Write Mesh Bounding Box

Write mesh bounding box to array MESH_BBOX (**master process only**):

```
long unsigned int bbox[6];
bbox[0] = <<number of blocks in x-direction>>;
bbox[1] = <<number of blocks in y-direction>>;
bbox[2] = <<number of blocks in z-direction>>;
bbox[3] = <<number of cells in x-direction in blocks>>;
bbox[4] = <<number of cells in y-direction in blocks>>;
bbox[5] = <<number of cells in z-direction in blocks>>;

// NOTE: If you are not using block-based mesh, use
//bbox[0] = <<number of zones in x-direction>>;
//bbox[1] = <<number of zones in y-direction>>;
//bbox[2] = <<number of zones in z-direction>>;
//bbox[3] = 1;
//bbox[4] = 1;
//bbox[5] = 1;

map<string,string> xmlAttributes;
xmlAttributes["mesh"] = <<name of your mesh>>;
bool success;
if (myRank == masterRank) success =
vlsv.writeArray("MESH_BBOX",xmlAttributes,6,1,bbox);
else success = vlsv.writeArray("MESH_BBOX",xmlAttributes,0,1,bbox);
```

Following XML attributes are supported:

Attribute name	Mandatory or optional?	Accepted values	Value datatype
mesh	mandatory	Name of the mesh.	string

*Rationale: Plugin needs to know how to compute zone (i,j,k) indices from global IDs using the formula given above. Nx is equal to bbox[0]*bbox[3] etc.*

Write Bounding Box Node Coordinates

Write bounding box node (x,y,z) coordinates to arrays MESH_NODE_CRDS_X, MESH_NODE_CRDS_Y, and MESH_NODE_CRDS_Z (**master process only**). NOTE that a mesh always has more nodes than zones:

```
unsigned int xCells = <<number of zones in x-direction>>;
unsigned int yCells = <<number of zones in y-direction>>;
unsigned int zCells = <<number of zones in z-direction>>;

float* xNodeCrds = new float[xCells+1];
float* yNodeCrds = new float[yCells+1];
float* zNodeCrds = new float[zCells+1];
for (unsigned int i=0; i<xCells+1; ++i) xNodeCrds[i] = <<x-coordinate of node i>>;
for (unsigned int j=0; j<yCells+1; ++j) yNodeCrds[j] = <<y-coordinate of node j>>;
for (unsigned int k=0; k<zCells+1; ++k) zNodeCrds[k] = <<z-coordinate of node k>>;

map<string,string> xmlAttributes;
xmlAttributes["mesh"] = <<name of your mesh>>;
bool success = true;
if (myRank == masterRank) {
   if (vlsv.writeArray("MESH_NODE_CRDS_X",xmlAttributes,xCells+1,1,xNodeCrds) ==
false) success = false;
   if (vlsv.writeArray("MESH_NODE_CRDS_Y",xmlAttributes,yCells+1,1,yNodeCrds) ==
false) success = false;
   if (vlsv.writeArray("MESH_NODE_CRDS_Z",xmlAttributes,zCells+1,1,zNodeCrds) ==
false) success = false;
} else {
   if (vlsv.writeArray("MESH_NODE_CRDS_X",xmlAttributes,0,1,xNodeCrds) == false)
success = false;
   if (vlsv.writeArray("MESH_NODE_CRDS_Y",xmlAttributes,0,1,yNodeCrds) == false)
success = false;
   if (vlsv.writeArray("MESH_NODE_CRDS_Z",xmlAttributes,0,1,zNodeCrds) == false)
success = false;
}
```

Following XML attributes are supported:

Attribute name	Mandatory or optional?	Accepted values	Value datatype
mesh	mandatory	Name of the mesh.	string

Rationale: Plugin calculates zone's (i,j,k) indices from its global ID using the formula given above. Each zone has eight nodes: (i,j,k), (i+1,j,k), (i+1,j+1,k), ..., (i+1,j+1,k+1). Node (x,y,z) coordinates are obtained by indexing the arrays written above.

Write Zone Global ID numbers

Write global IDs of local zones, followed by global IDs of ghost zones, to an array MESH.

```
unsigned int localID=0;
vector<unsigned int> globalIDs;
for all local zones z {
   // Current value of localID is the local ID of zone z
   unsigned int gID = <<global ID of z>>;
   globalIDs.push_back(gID);
   ++localID;
}
for all ghost zones g {
   // Current value of localID is the local ID of zone g
   unsigned int gID = <<global ID of g>>;
   globalIDs.push_back(gID);
   ++localID;
}
unsigned int N_zones = globalIDs.size();

map<string,string> xmlAttributes;
attributes["name"] = <<name of your mesh>>;
attributes["type"] = "multi_ucd";
bool success = vlsv.writeArray("MESH",xmlAttributes,N_zones,1,&(globalIDs[0]));
```

The example above also shows how local ID values are obtaines -- it is simply zone's order in vector globalIDs. Local IDs need to be syncronized between neighboring MPI processes, they are needed in writing array MESH_GHOST_LOCALIDS below. **Important**: the order of zones in vector globalIDs defines the order in which a process writes out its variable data.

Following XML attributes are supported:

Attribute name	Mandatory or optional?	Accepted Values	Value datatype
domains	optional	Total number of domains in mesh.	integer
geometry	optional	Mesh coordinate system, either of "cartesian", "cylindrical", or "spherical". Defaults to "cartesian".	string
name	mandatory	Name of the mesh (must be valid for VisIt).	string
type	mandatory	"multi_ucd".	string
xlabel	optional	Label for x-axis. Defaults to "x-coordinate".	string
xperiodic	optional	Is mesh periodic in x-direction, "yes" or "no". Defaults to "no".	string
xunits	optional	Units for x-coordinate. Defaults to "".	string
ylabel	optional	Label for y-axis. Defaults to "y-coordinate".	string
yperiodic		Is mesh periodic in y-direction, "yes" or "no". Defaults to "no".	string
yunits	optional	Units for y-coordinate. Defaults to "".	string
zlabel	optional	Label for z-axis. Defaults to "z-coordinate".	string
zperiodic		Is mesh periodic in z-direction, "yes" or "no". Defaults to "no".	string
zunits	optional	Units for z-coordinate. Defaults to "".	string

Rationale: In order for VisIt to visualize data in parallel it must be able to compute each domain independently, including ghost zones. Plugin uses global IDs to compute node coordinates of each zone.

Write Domain Sizes

Write a vector of size two containing (total number of zones, number of ghosts) in the domain to an array MESH_DOMAIN_SIZES:

```
int domainSize[2];
domainSize[0] = <<total number or zones in this domain>>;
domainSize[1] = <<number of ghost zones in this domain>>;

map<string,string> xmlAttributes;
attributes["mesh"] = <<name of mesh>>;bool success =
vlsv.writeArray("MESH_DOMAIN_SIZES",xmlAttributes,1,2,domainSize);
```

Following XML attributes are supported:

Attribute name	Mandatory or optional?	Accepted values	Value datatype
mesh	mandatory	Name of the mesh.	string

Rationale: Plugin needs to know the number of local and ghost zones in each domain -- ghost zones need to be flagged so that VisIt does not draw them. Number of ghost zones given here also tells how to index the two arrays written below.

Write Ghost Zone Domain and Local ID Numbers

Write ghost zone domain IDs to array MESH_GHOST_DOMAINS, and local IDs to array MESH_GHOST_LOCALIDS:

```
unsigned int N_ghosts = 0;
vector<unsigned int> ghostDomainIDs;
vector<unsigned int> ghostLocalIDs;
for all ghost zones g {
   int domainID = <<MPI process rank owning ghost zone g>>;
   int localID = <<local ID of ghost zone g in domain domainID>>;
   ghostDomainIDs.push_back(domainID);
   ghostLocalIDs.push_back(localID);
   ++N_ghosts;
}

map<string,string> xmlAttributes;
attributes["mesh"] = <name of mesh>;
bool success1 =
vlsv.writeArray("MESH_GHOST_DOMAINS",xmlAttributes,N_ghosts,1,&(ghostDomainIDs[0]));
success2 =
vlsv.writeArray("MESH_GHOST_LOCALIDS",xmlAttributes,N_ghosts,1,&(ghostLocalIDs[0]));
```

Following XML attributes are supported:

Attribute name	Mandatory or optional?	Accepted values	Value datatype
mesh	mandatory	Name of the mesh.	string

Rationale: This data is used when VisIt is requesting variable data. Each domain must give valid variable values for all its zones (local + ghosts), these two arrays tell plugin how to fetch data that belongs to other domains.

HOWTO: Write Multi-Domain Variables

VLSV file format accepts all types of variables. Variables are defined as tuples that exist of all zones in the domain. VisIt, however, only supports scalars (tuple size 1), vectors (tuple size 2 or 3), and tensors (tuple size 9). Variable data must be written in the same order as zones were written to array MESH above. Variables are written to arrays called VARIABLE. Example below shows how to write a three-component vector variable to VLSV file:

```
// Copy variable data to temporary buffer:
int arraySize=<<number of local zones in domain>>;
int vectorSize=3;
double* buffer = new double[arraySize*vectorSize];
int localID=0;
for all local zones z {
   for (i=0; i<vectorSize; ++i) {
      buffer[localID*vectorSize+i] = <<value of i:th component of vector in zone
z>>
   }
   ++localID;
}

// Write buffer to file:
map<string,string> xmlAttributes;
xmlAttributes["mesh"]=<<name of your variable>>;
xmlAttributes["mesh"]=<<name of your mesh>>;
bool success=vlsv.writeArray("VARIABLE",xmlAttributes,arraySize,vectorSize,buffer);
delete [] buffer;
```

Rationale: each domain only writes variable data on its local zones. Plugin uses MESH_GHOST_DOMAINS and MESH_GHOST_LOCALIDS arrays to fetch values to domain's ghost zones.

Following XML attributes are supported:

Attribute name	**Mandatory or optional?**	**Accepted values**	**Value datatype**
centering	optional	"node" or "zone", defaults to "zone".	string
dir	optional	Name of sub-menu in VisIt where variable appears.	string
name	mandatory	Name of the variable.	string
mesh	mandatory	Name of the mesh.	string

VLSV Library

Configuration

Architecture-dependent compiler, linker, and archiver flags are set in `Makefile.arch` file(s), where "`.arch`" is file suffix for your architecture. For example, in server called meteo flags are set in `Makefile.meteo`. You need to make your own copy of `Makefile.arch` file and set correct flags. Currently you need to set C++ compiler and linker flags, define the name of archiver, and set include and library paths to `SILO` library (needed by `vlsv2silo`). Note that you should not touch `Makefile` file.

Here is an example architecture Makefile here:

```
##############################################
###          ARCHITECTURE-DEPENDENT        ###
### MAKEFILE FOR VLSV PARALLEL FILE FORMAT ###
###                                        ###
###    Make a copy of this file and edit   ###
###     include and library paths below    ###
##############################################

# Compiler and its flags. Leave FLAGS empty, it is
# intended to be used to override the options given here.
# For example,
# make "FLAGS=-O0 -g" "ARCH=arch"
# would re-set optimization level to 0 and define debugging flag -g.

# Name of MPI compiler and its flags:
CMP = mpic++
CXXFLAGS = -O3 -std=c++0x -Wall
FLAGS =

# Name of archiver:
AR = ar

# SILO include and library paths:
INC_SILO=-I<path to SILO include dir>
LIB_SILO=-L<path to SILO library dir> -lsilo
```

Compilation

After you have created your own architecture `Makefile`, for example `Makefile.myarch`, compile VLSV library using make:

```
> make "ARCH=myarch"
```

Upon successful compilation VLSV library file `libvlsv.a` should appear in the directory.

VisIt Plugin

Configuration

VisIt handles it's compilation environment. Open the `vlsv.xml` file with `xmledit` and set include and library paths point to vlsv directory. Note that **you need to download VisIt source file distribution** in order to build the plugin!

```
> xmledit vlsv.xml
```

Open tab `CMake` and set correct include and library paths to `CXXFLAGS` and `LDFLAGS` fields.

Compilation

VisIt comes with several programs that are meant to aid the writing and compilation of user-defined plugins. Information needed to autogenerate Makefiles is in `vlsv.xml` file, which must be processes with xml2cmake and `CMake` before compiling with `Make`:

```
> xml2cmake vlsv.xml
> cmake .
> make
```

Note that you may need to manually open `CMakeList.txt` file and edit the required `CMake` version number on line 2

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.8 FATAL_ERROR)
```

to correspond to the version installed on your system.

Development

Serial vs. Parallel Engine

Separate plugin libraries are generated for serial and parallel database engines. VisIt seems to use pure MPI parallelization (unconfirmed). In practice the same source code file(s) can be used for both, but MPI includes and function calls must be guarded by PARALLEL preprocessor macros:

```
#ifdef PARALLEL
   #include <mpi.h>
#endif
```

Makefile is autogenerated by cmake. When plugin code is compiled with make, only the parallel plugin version uses mpic++ with -DPARALLEL compiler flag, serial plugin is compiled with g++. In other words, compiler error(s) will be generated if MPI code is not inside PARALLEL preprocessor macros.

Order of Execution

Basically any MPI process can request any domain in the mesh.