ParGrid Basics

March 21, 2012

1 Parallel Mesh

Computational meshes are often Cartesian rectangular cuboids ("shoeboxes"), although this needs not be the case. In parallel simulations the mesh (simulation domain) is somehow partitioned (decomposed) amongst N processes. Figure 1 shows an example of mesh partitioning in two-dimensional case.

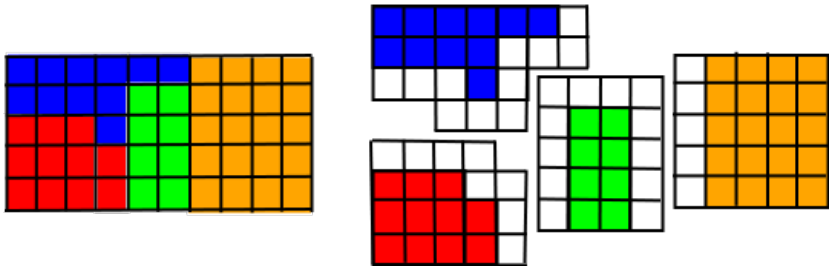

Figure 1: Left) Example of mesh partitioning to four processes (colours). Right) Processes need to allocate buffer cells (uncoloured) for cells stored on other processes.

In parallel simulations it is perhaps easiest to think of cells that are connected to other cells[1] instead of a mesh. For example, in Cartesian mesh each cell is connected to eight surrounding cells in two-dimensional case, and 26 cells in three-dimensional case (Figure 2). Some of the neighbouring cells are local to the process, the rest are remote neighbours hosted on other process(es) (uncoloured cells in Figure 1). Some of the neighbours may not exists – the cells at the boundary of the simulation domain have one or more missing neighbours. Cells with at least one missing neighbour are often flagged as "ghost cells" (Figure 1).

In ParGrid parlance ghost cells are called **exterior cells**, while cells whose neighbours exist are called **interior cells**. Each process has zero or more **local cells** (coloured cells in Figure 1), and keeps a copy of necessary amount of **remote cells** (uncoloured cells in Figure 1). Each process considers processes that

[1] Note: "cell" here should be understood as the smallest unit of parallelization. Parallel cell may actually represent a small Cartesian patch/block of cells.

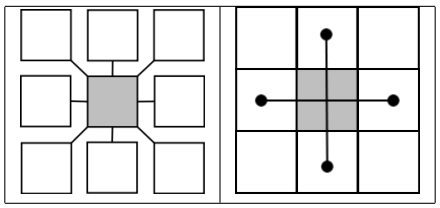

Figure 2: Left) Each cell is connected to eight (26) cells in two(three)-dimensional case. Some of the neighbours may not exist. Right) In finite difference advection equation data on marked (face)neighbours are needed to propagate the gray cell. The required cell data defines the transfer stencil of this particular algorithm.

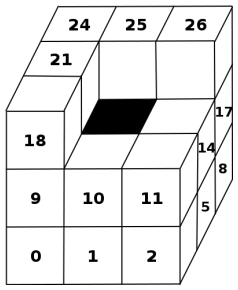

Figure 3: In ParGrid the neighbour type IDs are indices into a $3 \times 3 \times 3$ block of cells, centered at the considered cell (drawn in black). Type ID is calculated as $\mathrm{k} \cdot 9 + \mathrm{j} \cdot 3 + \mathrm{i}$, where (i,j,k) are the indices starting from the bottom lower left corner. The considered cell has indices (1,1,1), corresponding to neighbour type ID 13.

host one or more remote neighbours as their **neighbouring processes**. For example, in Figure 1 red and orange processes consider blue and green processes as their neighbouring processes. Red and orange process are not neighbours.

Cells are identified by unique **global ID** numbers, i.e. all processes agree on these numbers. For example, in a regular Cartesian grid cells are identified by their (i, j, k) indices. Each (i, j, k) tuple corresponds to a unique value (index into an array[2]), which can be used as global IDs. In ParGrid processes are only aware of their local cells, and of local cells' neighbours, some of which reside on other processes. Each process stores the cells (and associated data) in regular arrays that are accessed with **local IDs**; local IDs are simply indices into arrays. There is also a mapping between local and global IDs. Usually one only needs to care about the local IDs.

In order to correctly identify neighbour cells each cell has a neighbour list,

[2]In C/C++ index = k*ysize*xsize+j*xsize+i.

which is an array of size 27^3 that lists the local IDs of its neighbours[4]. Neighbours' local IDs are stored in particular order that tells which of the 27 cells the local ID refers to (see Figure 3)[5]. The indices into neighbour list can be computed with `pargrid::calcNeighbourTypeID` method that, instead of (i,j,k) index of the neighbour, takes the offset relative to the cell in question as parameters. For example, -x neighbour has offset $(-1, 0, 0)$, and +x,+y,+z neighbour has offset $(+1, +1, +1)$. A non-existing neighbour has local ID (and global ID) value equal to `pargrid::invalidCellID()`. Algorithm 1 shows how neighbour local IDs are obtained with ParGrid.

Algorithm 1 Example of how to access cell's neighbour(s).

```
// Get neighbours of a cell with local ID localID
const pargrid::CellID* const nbrs = pargrid.getCellNeighbourIDs(localID);

// Get local ID of (-1,0,0) neighbour
pargrid::CellID nbrID = nbrs[pargrid.calcNeighbourTypeID(-1,0,0)];
```

Before discussing data defined on cells let's consider a particular (bad) implementation of two-dimensional advection equation $\partial_t f + \mathbf{V} \cdot \nabla f = 0$

$$\frac{f_{i,j}^{n+1} - f_{i,j}^{n}}{\Delta t} = -V_x \frac{f_{i+1,j}^{n} - f_{i-1,j}^{n}}{2\Delta x} - V_y \frac{f_{i,j+1}^{n} - f_{i,j-1}^{n}}{2\Delta y}. \quad (1)$$

Here n refers to the time step, and i and j are cell indices in a Cartesian mesh. Figure 2 (right panel) illustrates the neighbour data that are needed to propagate a cell forward in time. In this example only the four out of eight neighbours are needed. Figure 2 (right panel) defines the **stencil** of this algorithm – in order to solve the gray cell correctly, the data on the marked cells must be up-to-date, i.e. in sync with neighbouring processes. It is usually quite inefficient to sync data on all neighbours if only, say, face neighbours are needed. Algorithm 2 shows how one can define a new stencil with ParGrid[6].

Algorithm 2 Example of how to define a (transfer) stencil with ParGrid. Here only data on x/y face neighbours is to be synchronized.

```
// define cells whose data needs to be received
vector<pargrid::NeighbourID> nbrTypeIDs;
nbrTypeIDs.push_back(pargrid.calcNeighbourTypeID(-1,0,0));
nbrTypeIDs.push_back(pargrid.calcNeighbourTypeID(+1,0,0));
nbrTypeIDs.push_back(pargrid.calcNeighbourTypeID(0,-1,0));
nbrTypeIDs.push_back(pargrid.calcNeighbourTypeID(0,+1,0));
int stencilID =
        pargrid.addStencil(pargrid::localToRemoteUpdates,nbrTypeIDs);
if (newStencilID < 0) cerr << "ERROR" << endl;
```

[3] The size of neighbour list is defined in `pargrid::N_neighbours`.

[4] ParGrid only stores cells' immediate neighbours. If one needs data from more distant neighbours, then each (parallel) cell must contain a patch/block of cells.

[5] A cell is also considered to be its own neighbour.

[6] ParGrid always has one stencil with ID 0 (zero) that transfers all data.

Algorithm 3 Example of how to define parallel data arrays with ParGrid. The example also shows how to access the created arrays.

```
// Define a double data array that has five values per cell
string name = "hydro";
unsigned int userDataID = pargrid.addUserData<double>(name,5);
if (userDataID == pargrid.invalidDataID())
        cerr << "ERROR" << endl;

// Two ways to access the user data
char* ptr1 = pargrid.getUserData(userDataID);
char* ptr2 = pargrid.getUserData(name);

// Set values for data on cell with local ID localID
double* data = reinterpret_cast<double*>(ptr1);
data[localID*5+0] = rho;
data[localID*5+1] = rhovx;
data[localID*5+2] = rhovy;
data[localID*5+3] = rhovz;
data[localID*5+4] = energy;
```

Algorithm 4 Example of how to connect a user-defined parallel data array to a stencil.

```
unsigned int transferID = 3;
bool result
   = pargrid.addUserDataTransfer(userDataID,stencilID,transferID,false);
if (result == false) cerr << "ERROR" << endl;
```

How to define cell data? In ParGrid there are two approaches to this, old and new. If possible one should use the (new) method described here. Typically one need several data values per cell. For example, in hydrodynamical simulations $(\rho, \rho v_x, \rho v_y, \rho v_z, U)$ need to be stored, i.e. five floating point values per cell. Algorithm 3 shows an example how to define parallel data arrays with ParGrid. Note that it is entirely up to the user how the data should be understood, i.e. is the data stored as cell average, or on particular cell node/face/edge.

Definition of stencil or data arrays is not enough to make the magic happen. One also needs to tell ParGrid which stencil(s) are used to transfer the data array(s)[7]. This is done by adding **transfers** to ParGrid.

Cells whose every neighbour in the stencil are local are called **inner cells**. Cells that have remote neighbours in the stencil are called **boundary cells**[8]. Boundary cells cannot be propagated until the data on (local copies of) remote cells has been synchronized with neighbouring processes. Typically one first starts the neighbour data sync, propagates inner cells while waiting for sync to complete, and finally propagates boundary cells. Algorithm 5 shows how this is done with ParGrid.

[7] It is also possible to sync the same data array using several different stencils.

[8] Unfortunately English language has quite limited vocabulary in this matter.

Algorithm 5 Example of how to start neighbour data sync, and how to propagate inner and boundary cells.

```
// Start data sync //
pargrid.startNeighbourExchange(stencilID,transferID);

// Propagate inner cells //
const vector<pargrid::CellID>& innerCells
                        = pargrid.getInnerCells(stencilID);
for (size_t c=0; c<innerCells.size(); ++c) {
        //** propagate cell with local ID innerCells[c] **//
}

// Wait for data sync to complete //
pargrid.wait(stencilID,transferID);

// Propagate boundary cells //
const vector<pargrid::CellID>& boundaryCells
                        = pargrid.getBoundaryCells(stencilID);
for (size_t c=0; c<boundaryCells.size(); ++c) {
        //** propagate cell with local ID boundaryCells[c] **//
}
```

2 Desing Considerations

The "old" way to define cell data is basically what one does as a first approximation: the cell data is defined in a header file which needs to be included everywhere. This presents a problem for code coupling and reusability. For example, let's consider an MHD simulation, and a particle simulation that basically just requires $\mathbf{E}, \mathbf{B}$-field to propagate the particles. Both codes are useful as they are, but it might be useful to couple them in some applications which presents some non-trivial issues. With data arrays defined above there is no need for header files that define cell data. The names of the data files, which are defined runtime, can be read from configuration file.