Parallel & Distributed Computing: Lecture 12

Alberto Paoluzzi

November 15, 2017

Project: Add-ons modules to LARLIB.jl

Example of Python *to* Julia conversion

1. Step one: removal of dependencies from `larlib.largrid.py`
2. Step two: syntactic translation of functions
3. Step three: integration test
4. Step four: Debug, debug, debug

Introduction from largrid.pdf

This report aims to discuss the design and the implementation of the `largrid` module of the LAR-CC library, including also the Cartesian product of general cellular complexes. In particular, we show that both n-dimensional grids of (hyper)-cuboidal cells and their d-dimensional skeletons $(0 \leq d \leq n)$, embedded in $\mathbb{E}^n$, may be properly and efficiently generated by assembling the cells produced by a number n of either 0- or 1-dimensional cell complexes, that in such lowest dimensions coincide with simplicial complexes.

In Section 2 we give the simple implementation of generation of lower-dimensional (say, either 0- or 1-dimensional) regular cellular complexes with integer coordinates. In Section 3 a functional decomposition of the generation of either full-dimensional cuboidal complexes in $\mathbb{E}^n$ and of their d-skeletons $(0 \leq d \leq n)$ is given, showing in particular that every skeleton can be efficiently generated as a partition in cell subsets produced by the Cartesian product of a proper disposition of 0-1 complexes, according to the binary representation of a subset of the integer interval $[0, 2^n]$. In Section 5 we provide a very simple and general implementation of the topological product of *two* cellular complexes of any topology. When applied to embedded linear cellular complexes (i.e. when the coordinates of 0-cells of arguments are fixed and given) the algorithm produces a Cartesian product of its two arguments. In Section 6 the exporting of the module to different languages is provided. The Section 7 contains the unit tests associated to the various algorithms, that are exported by the used literate environment in the proper test subdirectory—depending on the implementation

Figure 1: Introduction to module

Step one: removal of dependencies from `larlib.largrid.py`

Look for function definitions

Get the significant subset

```
def larModelProduct(twoModels):
def grid_0(n):
def grid_1(n):
def larGrid(n):
def index2addr (shape):
def binaryRange(n):
def larVertProd(vertLists):
def filterByOrder(n):
def larCellProd(cellLists):
def larGridSkeleton(shape):
def larImageVerts(shape):
def larCuboids(shape, full=False):
def mergeSkeletons(larSkeletons):
```

From largrid.py To largrid-test.py 0/5

3 Cuboidal grids

More interesting is the generation of *hyper-cubical grids* of intrinsic dimension d embedded in n-dimensional space, via the Cartesian product of d 1-complexes and $(n-d)$ 0-complexes. When $d = n$ the resulting grid is said *solid*; when $d = 0$ the output grid is 0-dimensional, and corresponds to a grid-arrangement of a discrete set of points in $\mathbb{E}^n$.

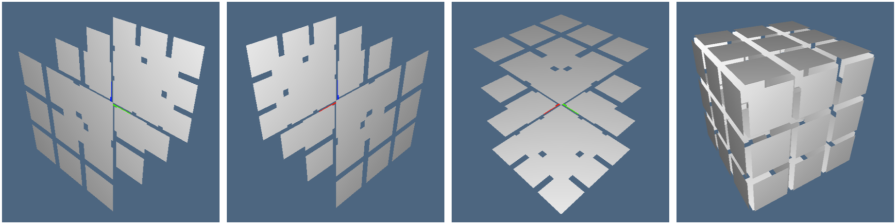

Figure 3: 2-skeleton of a 3-grid in $\mathbb{E}^3$

From largrid.py To largrid-test.py 1/5

Original code

```
def grid0(n):
    cells = AA(LIST)(range(n+1))
    return cells

def grid1(n):
    ints = range(n+1)
    cells = TRANS([ints[:-1],ints[1:]])
    return cells
```

From largrid.py To largrid-test.py 1/5

Original code

```
def grid0(n):
    cells = AA(LIST)(range(n+1))
    return cells

def grid1(n):
    ints = range(n+1)
    cells = TRANS([ints[:-1],ints[1:]])
    return cells
```

Removed dependencies in python

```
def grid_0(n):
    return [[i] for i in xrange(n+1)]

def grid_1(n):
    return [[i,i+1] for i in xrange(n)]
```

From largrid.py To largrid-test.py 2/5

Multi-index to address transformation The second-order utility `index2addr` function transforms a `shape` list for a multidimensional array into a function that, when applied to a multindex array, i.e. to a list of integers within the `shape`'s bounds, returns the integer address of the array component within the linear storage of the multidimensional array.

The transformation formula for a d-dimensional array with `shape` $(n_0, n_1, ..., n_{d-1})$ is a linear combination of the 0-based[1] multi-index $(i_0, i_1, ..., i_{d-1})$ with `weights` equal to $(w_0, w_1, ..., w_{d-2}, 1)$:

$$addr = i_0 \times w_0 + i_1 \times w_1 + \cdots + i_{d-1} \times w_{d-1}$$

where

$$w_k = n_{k+1} \times n_{k+2} \times \cdots \times n_{d-1}, \qquad 0 \leq k \leq d-2.$$

Therefore, we get `index2addr([4,3,6])([2,2,0])` $= 48 = 2 \times (3 \times 6) + 2 \times (6 \times 1) + 0$, where `[2,2,0]` represent the numbers of (pages, rows, columns) indexing an element in the three-dimensional array of shape `[4,3,6]`.

Figure 5: Generation of address of elements in a linearized array

From largrid.py To largrid-test.py 2/5

Original code

```
def index2addr (shape):
    n = len(shape)
    shape = shape[1:]+[1]
    weights = [PROD(shape[k:]) for k in range(n)]
    def index2addr0 (multindex):
        return INNERPROD([multindex, weights])
    return index2addr0
```

From largrid.py To largrid-test.py 2/5

Original code

```
def index2addr (shape):
    n = len(shape)
    shape = shape[1:]+[1]
    weights = [PROD(shape[k:]) for k in range(n)]
    def index2addr0 (multindex):
        return INNERPROD([multindex, weights])
    return index2addr0
```

Removed dependencies

```
def index2addr (shape):
    n = len(shape)
    theShape = shape[1:]+[1]
    weights = [prod(theShape[k:]) for k in xrange(n)]
    def index2addr0 (multiIndex):
        return dot(multiIndex, weights)
    return index2addr0
```

From largrid.py To largrid-test.py 3/5

Original code

```
def larVertProd(vertLists):
    return AA(CAT)(CART(vertLists))
```

From largrid.py To largrid-test.py 3/5

Original code

```
def larVertProd(vertLists):
    return AA(CAT)(CART(vertLists))
```

Removed dependencies

```
using IterTools

def larVertProd(vertLists):
    return [[x[0] for x in v] for v in itertools.product(*vertLists)]
```

From largrid.py To largrid-test.py 4/5

Original code

```
def binaryRange(n):
def larCellProd(cellLists):
    shapes = [len(item) for item in cellLists]
    indices = CART([range(shape) for shape in shapes])
    jointCells = [CART([cells[k] for k,cells in zip(index,cellLists)])
                 for index in indices]
    convert = index2addr([ shape+1 if (len(cellLists[k][0]) > 1) else shape
                             for k,shape in enumerate(shapes) ])
    return [AA(convert)(cell) for cell in jointCells]
```

From largrid.py To largrid-test.py 4/5

Original code

```
def binaryRange(n):
def larCellProd(cellLists):
    shapes = [len(item) for item in cellLists]
    indices = CART([range(shape) for shape in shapes])
    jointCells = [CART([cells[k] for k,cells in zip(index,cellLists)])
                 for index in indices]
    convert = index2addr([ shape+1 if (len(cellLists[k][0]) > 1) else shape
                         for k,shape in enumerate(shapes) ])
    return [AA(convert)(cell) for cell in jointCells]
```

Removed dependencies

```
def larCellProd(cellLists):
    shapes = [len(item) for item in cellLists]
    indices = itertools.product(*[xrange(shape) for shape in shapes])
    def assemblies(index):
        return itertools.product(*[cells[k] for k,cells in zip(index,cellLists)])
    jointCells = [list(assemblies(index)) for index in indices]
    convert = index2addr([ shape+1 if (len(cellLists[k][0]) > 1) else shape
                         for k,shape in enumerate(shapes) ])
    return [map(convert, cell) for cell in jointCells]
```

From largrid.py To largrid-test.py 5/5

Original code

```
def larGridSkeleton(shape):
    n = len(shape)
    def larGridSkeleton0(d):
        components = filterByOrder(n)[d]
        componentCellLists = [AA(APPLY)(zip( AA(larGrid)(shape),(component) ))
                              for component in components]
        return CAT([ larCellProd(cellLists)  for cellLists in componentCellLists ])
    return larGridSkeleton0
```

From largrid.py To largrid-test.py 5/5

Original code

```
def larGridSkeleton(shape):
    n = len(shape)
    def larGridSkeleton0(d):
        components = filterByOrder(n)[d]
        componentCellLists = [AA(APPLY)(zip( AA(larGrid)(shape),(component) ))
                              for component in components]
        return CAT([ larCellProd(cellLists)  for cellLists in componentCellLists ])
    return larGridSkeleton0
```

Removed dependencies

```
def larGridSkeleton(shape):
    n = len(shape)
    def larGridSkeleton0(d):
        components = filterByOrder(n)[d]
        componentCellLists = [ [apply(f,[x]) for f,x in zip( [larGrid(dim) for dim i
                              for component in components ]
        print "componentCellLists =",componentCellLists
        out = [ larCellProd(cellLists)  for cellLists in componentCellLists ]
        return list(itertools.chain(*out))
    return larGridSkeleton0
```

Step two: syntactic translation of functions

Language translation 1/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def grid_0(n):
    return [[i] for i in xrange(n+1)]
def grid_1(n):
    return [[i,i+1] for i in xrange(n)]
def larGrid(n):
    def larGrid1(d):
        if d==0: return grid_0(n)
        elif d==1: return grid_1(n)
    return larGrid1
```

Language translation 1/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def grid_0(n):
    return [[i] for i in xrange(n+1)]
def grid_1(n):
    return [[i,i+1] for i in xrange(n)]
def larGrid(n):
    def larGrid1(d):
        if d==0: return grid_0(n)
        elif d==1: return grid_1(n)
    return larGrid1
```

Julia translation

```
function grid_0(n)
    return hcat([[i] for i in range(0,n+1)]...) end
function grid_1(n)
    return hcat([[i,i+1] for i in range(0,n)]...) end
function larGrid(n)
    function larGrid1(d)
        if d==0
            return grid_0(n)
        elseif d==1
            return grid_1(n)
        end end
    return larGrid1 end
```

Language translation 1/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def grid_0(n):
    return [[i] for i in xrange(n+1)]
def grid_1(n):
    return [[i,i+1] for i in xrange(n)]
def larGrid(n):
    def larGrid1(d):
        if d==0: return grid_0(n)
        elif d==1: return grid_1(n)
    return larGrid1
```

Julia translation

```
function grid_0(n)
    return hcat([[i] for i in range(0,n+1)]...) end
function grid_1(n)
    return hcat([[i,i+1] for i in range(0,n)]...) end
function larGrid(n)
    function larGrid1(d)
        if d==0
            return grid_0(n)
        elseif d==1
            return grid_1(n)
        end end
    return larGrid1 end
```

Language translation 2/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def index2addr (shape):
    n = len(shape)
    theShape = shape[1:]+[1]
    weights = [prod(theShape[k:]) for k in xrange(n)]
    def index2addr0 (multiIndex):
        return dot(multiIndex, weights)
    return index2addr0
```

Language translation 2/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def index2addr (shape):
    n = len(shape)
    theShape = shape[1:]+[1]
    weights = [prod(theShape[k:]) for k in xrange(n)]
    def index2addr0 (multiIndex):
        return dot(multiIndex, weights)
    return index2addr0
```

Julia translation

```
function index2addr(shape)
    n = length(shape)
    theShape = append!(shape[2:end],1)
    weights = [prod(theShape[k:end]) for k in range(1,n)]
    function index2addr0(multiIndex)
        return dot(collect(multiIndex), weights) + 1
    end
    return index2addr0
end
```

Language translation 3/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def binaryRange(n):
    return [('{0:0'+str(n)+'b}').format(k) for k in xrange(2**n)]
def larVertProd(vertLists):
    return [[x[0] for x in v] for v in itertools.product(*vertLists)]
def filterByOrder(n):
    terms = [[int(elem) for elem in list(term)] for term in binaryRange(n)]
    return [[term for term in terms if sum(term) == k] for k in xrange(n+1)]
```

Language translation 3/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def binaryRange(n):
    return [('{0:0'+str(n)+'b}').format(k) for k in xrange(2**n)]
def larVertProd(vertLists):
    return [[x[0] for x in v] for v in itertools.product(*vertLists)]
def filterByOrder(n):
    terms = [[int(elem) for elem in list(term)] for term in binaryRange(n)]
    return [[term for term in terms if sum(term) == k] for k in xrange(n+1)]
```

Julia translation

```
using IterTools
function binaryRange(n)
    return [bin(k,n) for k in range(0,2^n)] end
function larVertProd(vertLists)
    coords = [[x[1] for x in v] for v in IterTools.product(vertLists...)]
    return sortcols(hcat(coords...)) end
function filterByOrder(n)
    terms = [[parse(Int8,bit) for bit in convert(Array{Char,1},term)]
                for term in binaryRange(n)]
    return [[term for term in terms if sum(term) == k] for k in range(0,n+1)] end
```

Language translation 4/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larCellProd(cellLists):
    shapes = [len(item) for item in cellLists]
    indices = itertools.product(*[xrange(shape) for shape in shapes])
    def assemblies(index):
        return itertools.product(*[cells[k] for k,cells in zip(index,cellLists)])
    jointCells = [assemblies(index) for index in indices]
    convert = index2addr([ shape+1 if (len(cellLists[k][0]) > 1) else shape
                         for k,shape in enumerate(shapes) ])
    return [map(convert, cell) for cell in jointCells]
```

Language translation 4/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larCellProd(cellLists):
    shapes = [len(item) for item in cellLists]
    indices = itertools.product(*[xrange(shape) for shape in shapes])
    def assemblies(index):
        return itertools.product(*[cells[k] for k,cells in zip(index,cellLists)])
    jointCells = [assemblies(index) for index in indices]
    convert = index2addr([ shape+1 if (len(cellLists[k][0]) > 1) else shape
                           for k,shape in enumerate(shapes) ])
    return [map(convert, cell) for cell in jointCells]
```

Julia translation

```
function larCellProd(cellLists)
    shapes = [length(item) for item in cellLists]
    cart = IterTools.product([range(0,shape) for shape in shapes]...)
    indices = hcat([collect(tuple) for tuple in sort(collect(cart))]...)
    h = 1
    index = indices[:,h]
    inCart = [collect(cells[k+1]) for (k,cells) in collect(zip(index,cellLists))]
    jointCells = sort(collect(IterTools.product(inCart...)))
    for h in range(2,size(indices,2))
        index = indices[:,h]
        inCart = [collect(cells[k+1]) for (k,cells) in collect(zip(index,cellLists))]
        jointCells = hcat( jointCells, sort(collect(IterTools.product(inCart...))) )
    end
    convertIt = index2addr([ shape+1 for shape in shapes ])
    [hcat(map(convertIt, jointCells[:,j])...) for j in range(1,size(jointCells,2))]
```

Language translation 5/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larImageVerts(shape):
   def vertexDomain(n):
      return [[k] for k in xrange(n)]
   vertLists = [vertexDomain(k+1) for k in shape]
   vertGrid = larVertProd(vertLists)
   return vertGrid
```

Language translation 5/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larImageVerts(shape):
   def vertexDomain(n):
      return [[k] for k in xrange(n)]
   vertLists = [vertexDomain(k+1) for k in shape]
   vertGrid = larVertProd(vertLists)
   return vertGrid
```

Julia translation

```
function larImageVerts(shape)
    function vertexDomain(n)
        return [[k] for k in range(0,n)]
    end
    vertLists = [vertexDomain(k+1) for k in shape]
    vertGrid = larVertProd(vertLists)
    return vertGrid
end
```

Language translation 6/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larGridSkeleton(shape):
    n = len(shape)
    def larGridSkeleton0(d):
        components = filterByOrder(n)[d]
        componentCellLists = [ [apply(f,[x]) for f,x in zip( [larGrid(dim) for dim i
                              for component in components ]
        print "componentCellLists =",componentCellLists
        out = [ larCellProd(cellLists)  for cellLists in componentCellLists ]
        return list(itertools.chain(*out))
    return larGridSkeleton0
```

Language translation 6/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larGridSkeleton(shape):
    n = len(shape)
    def larGridSkeleton0(d):
        components = filterByOrder(n)[d]
        componentCellLists = [ [apply(f,[x]) for f,x in zip( [larGrid(dim) for dim 
                                for component in components ]
        print "componentCellLists =",componentCellLists
        out = [ larCellProd(cellLists)  for cellLists in componentCellLists ]
        return list(itertools.chain(*out))
    return larGridSkeleton0
```

Julia translation

```
function larGridSkeleton(shape)
    n = length(shape)
    function larGridSkeleton0(d)
        components = filterByOrder(n)[d]
        mymap(arr) = [arr[:,k]  for k in range(1,size(arr,2))]
        componentCellLists = [ [ mymap(f(x)) for (f,x) in zip( [larGrid(dim) for di
                                for component in components ]
        out = [ larCellProd(cellLists)  for cellLists in componentCellLists ]
```

Language translation 7/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larCuboids(shape, full=False):
   vertGrid = larImageVerts(shape)
   gridMap = larGridSkeleton(shape)
   if not full:
      cells = gridMap(len(shape))
   else:
      skeletonIds = xrange(len(shape)+1)
      cells = [ gridMap(id) for id in skeletonIds ]
   return vertGrid, cells
```

Language translation 7/7

largrid.py-test.py → largrid-test.jl

transformed code

```
def larCuboids(shape, full=False):
   vertGrid = larImageVerts(shape)
   gridMap = larGridSkeleton(shape)
   if not full:
      cells = gridMap(len(shape))
   else:
      skeletonIds = xrange(len(shape)+1)
      cells = [ gridMap(id) for id in skeletonIds ]
   return vertGrid, cells
```

Julia translation

```
function larCuboids(shape, full=false)
    vertGrid = larImageVerts(shape)
    gridMap = larGridSkeleton(shape)
    if ! full
        cells = gridMap(length(shape)+1)
    else
        skeletonIds = range(1,length(shape)+1)
        cells = [ gridMap(id) for id in skeletonIds ]
    end
```

Step three: integration test

Execute the functions to be exported (API)

```
print(larCuboids(shape, true})
```

Step four: Debug, debug, debug