TACC Technical Report IMP-22

Implementing LULESH in IMP

Victor Eijkhout*

May 9, 2017

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

We describe the implementation of the LULESH proxy-app in the Integrative Model.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model (under construction)
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)
IMP-26 Load balancing in IMP (under construction)
IMP-27 Data analytics in IMP (under construction)

1 Introduction

'Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)' [3] (https://codesign.llnl.gov/lulesh.php) is a proxy-app that models a number of aspects of computationally important application at Lawrence Livermore National Laboratory (LLNL).

1.1 What are the interesting bits in Lulesh?

Since I don't have time to make a full reimplementation of Lulesh, I focus on what I find the interesting bits.

From the specification document:

> Many operations in the code involve loops over the nodes or elements of the domain. In production hydro codes, the collection of variables referenced in a single element calculation must be passed through many levels of function calls. Thus as good software engineering practice, it is common to gather those variables into a set of local data structures. We follow this practice in the reference code; computational blocks use the local data structures rather than the domain-level data. When the operations are complete, we scatter the data from the local arrays back to the arrays on the domain.

You can see this in action in code such as:

```
// get nodal coordinates from global arrays and copy into local arrays.
for( Index_t lnode=0 ; lnode<8 ; ++lnode )
{
  Index_t gnode = elemNodes[lnode];
  x_local[lnode] = mesh.x(gnode);
  y_local[lnode] = mesh.y(gnode);
  z_local[lnode] = mesh.z(gnode);
}
```

which gathers the eight nodes of an element into a local array.

This operation is interesting in parallel because the distribution of the nodes and and elements necessarily can not be conforming. Thus I focus on the communication that is needed to support this gathering – and its reverse scattering – and completely ignore the numerics.

1.2 Regular versus irregular computation

In this report I will focus on regular 2D and 3D meshes. This makes it easy to convey the ease of programming with Integrative Model for Parallelism (IMP). However,

irregular finite element meshes are possible too. For this we note that the above global-to-local copy code describes one row out of the adjacency matrix of this operation. This means that we accomodate this by constructing an explicit β-distribution through its adjacency matrix.

In some cases the index array in the original code corresponds to a *column* of the adjacency matrix. For this we need to have an auxiliary function for describing the β-distribution through its adjoint. This is currently not implemented yet.

1.3 Outline

In this report we describe the following transfer operations, which stand for, or include, the transfers happening in an actual Lulesh implementation:

1. Distribute element quantities to locally stored node.
2. Sum (or otherwise combine) these local node quantities to global.
3. Distribute global nodes back to elements, or:
4. Update local node information from global, and gather local node information to element quantities.

2 Transfer operations

The IMP model effects a separation of concerns between communication/synchronization of tasks and the task-local code. In this report, we describe the various transfer between distributed quantities in the case of a regular two-dimensional mesh. We will simplify the actual computations to simple replication or summation.

For the communication, in the IMP model it is enough to describe the algorithm-determined dependencies of output data on input data. (Note: this is not the same as indicating messages or other explicit communication mechanisms.) This will be the bulk of the discussion; the task-local code will be largely identical to the sequential code, as we will show.

We refer to [1] for the general theory of IMP.

2.1 Element to local node

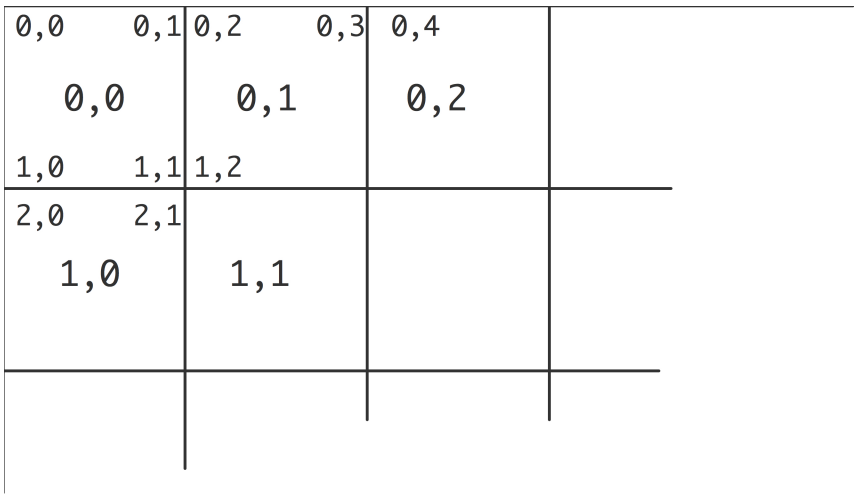

Figure 1: Element to local node broadcast, 2D case

Operation 1, distributing element information to the local copy of the node, is depicted in figure 1. The signature function describes the dependence of a (globally numbered) local node on the element number:

$$\sigma(i,j,k,\ldots) = \{i/2, j/2, k/2, \ldots\}.$$

This is implemented by the signature function code:

```
%% lulesh_functions.cxx
domain_coordinate *signature_coordinate_element_to_local( domain_coordinate *i ) {
  return i->operate_p( new ioperator("/2") );
};
std::shared_ptr<multi_indexstruct> signature_struct_element_to_local( std::shared_ptr<multi_indexs
  if (!i->is_contiguous())
    throw(std::string("signature_struct_element_to_local only for contiguous"));
  ioperator *divop = new ioperator("/2");
  return std::shared_ptr<multi_indexstruct>
    ( new contiguous_multi_indexstruct
      (i->first_index_r().operate(divop),i->last_index_r().operate(divop)) );
};
```

We note that this operation is likely to be local, but we do not indicate so either way.

The local code then operates only on input and output data that is avaible to a traditionally executed sequential program:

```
%% lulesh_functions.cxx
for (index_int i=qfirst[0]; i<=qlast[0]; i++)
  for (index_int ii=2*i; ii<2*i+2; ii++)
    for (index_int j=qfirst[1]; j<=qlast[1]; j++)
      for (index_int jj=2*j; jj<2*j+2; jj++) {
        index_int IJ = INDEX2D(i,j,in_offsets,in_nsize),
          IIJJ = INDEX2D(ii,jj,out_offsets,out_nsize);
        outdata[IIJJ] = indata[IJ];
      }
```

2.2 Local node to global

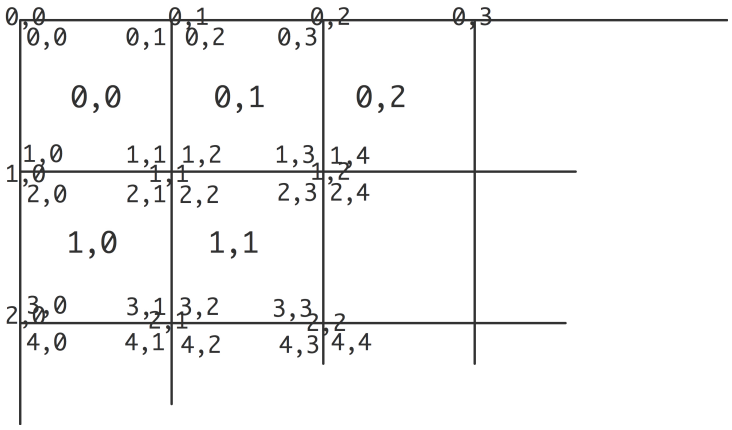

Figure 2: Correspondence between local and global node numbering

The mapping between local and global nodes (operation 2; see figure 2) is not trivially parallel, involving data transfer between processors since the two quantities can not be conformally partitioned.

The signature function, describing the dependence of a global node number on the local node numbers that constitute it, maps

$$g \mapsto \left(2g - \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}, 2g\right).$$

The signature code weeds out the negative numbers, and truncates beyond the far edge of the domain.

```
%% lulesh_functions.cxx
std::shared_ptr<multi_indexstruct> signature_local_from_global
    ( std::shared_ptr<multi_indexstruct> g,std::shared_ptr<multi_indexstruct> enc ) {
  int dim = g->get_same_dimensionality(enc->get_dimensionality());
  domain_coordinate_allones allones(dim);
  auto range = std::shared_ptr<multi_indexstruct>
    ( new contiguous_multi_indexstruct
      ( g->first_index_r()*2-allones,g->last_index_r()*2 ) );
  return range->intersect(enc);
};
```

The enclosing structure that we truncate against is passed in through a closure:

```
%% unittest_lulesh.cxx
local_to_global_nodes->last_dependency()->set_signature_function_function
( new multi_sigma_operator
  ( dim, [local_nodes] (std::shared_ptr<multi_indexstruct> g) -> std::shared_ptr<multi_indexstruct
    return signature_local_from_global( g,local_nodes->get_enclosing_structure() ); } ) )
```

The function does a simple averaging:

```
%% lulesh_functions.cxx
for (index_int i=pfirst[0]; i<=plast[0]; i++) {
  bool skip_first_i = i==out_gfirst[0], skip_last_i = i==out_glast[0];
  for (index_int j=pfirst[1]; j<=plast[1]; j++) {
    bool skip_first_j = j==out_gfirst[1], skip_last_j = j==out_glast[1];
    outdata[ INDEX2D(i,j,out_offsets,out_nsize) ] =
      ( !skip_first_i && !skip_first_j
        ? indata[ INDEX2D(2*i-1,2*j-1,in_offsets,in_nsize) ] : 0 )
      +
      ( !skip_first_i && !skip_last_j
        ? indata[ INDEX2D(2*i-1,2*j,in_offsets,in_nsize) ] : 0 )
      +
      ( !skip_last_i && !skip_first_j
        ? indata[ INDEX2D(2*i,2*j-1,in_offsets,in_nsize) ] : 0 )
      +
      ( !skip_last_i && !skip_last_j
        ? indata[ INDEX2D(2*i,2*j,in_offsets,in_nsize) ] : 0 )
      ;
  }
}
```

2.3 Global node to local

In reverse, to distribute global nodes back to local copies, we need a signature function that describes the dependency of a local node on its global counterpart. Referring to

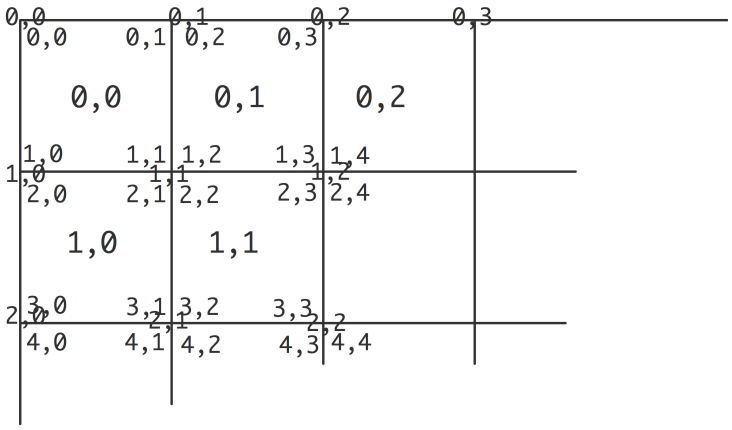

Figure 3: Correspondence between local and global node numbering

figure 3 we get this as follows:

$$g(\ell) = \left(\ell, \ldots, \ell + \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}\right)/2$$

with the obvious signature function

```
%% lulesh_functions.cxx
std::shared_ptr<multi_indexstruct> signature_global_node_to_local
    ( std::shared_ptr<multi_indexstruct> l ) {
  return l->operate
    ( new ioperator(">>1") )->operate( new ioperator("/2") );
};
```

There are no edge cases.

After the communication has been done, the local code implementation of this function can actually be fairly simple. Here is the 2D case:

```
%% lulesh_functions.cxx
for (index_int i=qfirst[0]; i<=qlast[0]; i++) {
  for (index_int j=qfirst[1]; j<=qlast[1]; j++) {
    bool
      left_i = i==0, right_i = i==global_nodes_sizes[0],
      left_j = j==0, right_j = j==global_nodes_sizes[1];
    index_int Iin = INDEX2D(i,j,in_offsets,in_nsize);
    double g = global_nodes_data[Iin];// [ i*global_nodes_sizes[0]+j ];
    if (!left_i && !left_j) {
      index_int Iout = INDEX2D( 2*i-1,2*j-1, out_offsets,out_nsize );
      local_nodes_data[Iout] = g; }
    if (!left_i && !left_j) {
      index_int Iout = INDEX2D( 2*i,  2*j-1, out_offsets,out_nsize );
      local_nodes_data[Iout] = g; }
    if (!left_i && !left_j) {
      index_int Iout = INDEX2D( 2*i-1,2*j,   out_offsets,out_nsize );
      local_nodes_data[Iout] = g; }
    if (!left_i && !left_j) {
      index_int Iout = INDEX2D( 2*i,  2*j,   out_offsets,out_nsize );
      local_nodes_data[Iout] = g; }
  }
}
```

2.4 Element from global node

Figure 4 depicts the relation between global nodes and elements in two dimensions.

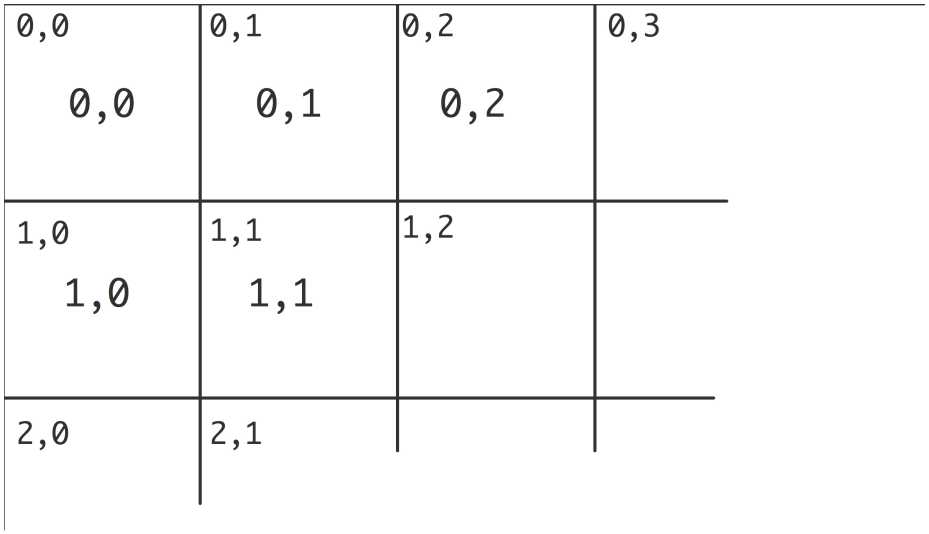

Figure 4: Element and global node numbering, 2D case

We see that

$$\sigma(e) = \left[e, \ldots, e + \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}\right]$$

2.5 Element from local node

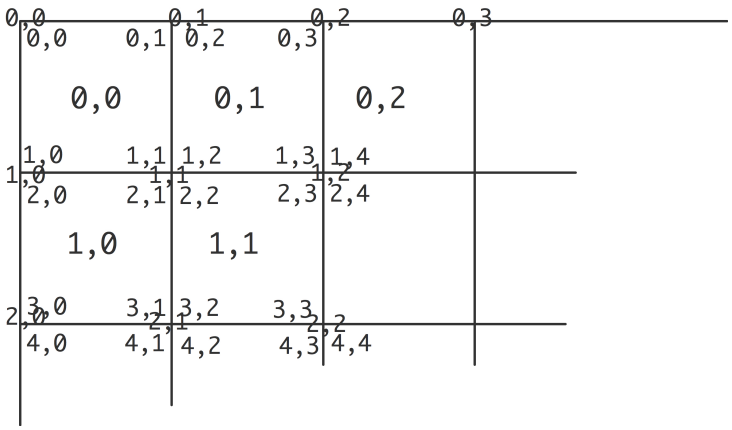

Figure 5: Correspondence between local and global node numbering

Operation 4 uses the inverse mapping

$$e \mapsto \left[2*e,\ldots,2*e+\begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}\right]$$

The signature function that implements this:

```
%% lulesh_functions.cxx
std::shared_ptr<multi_indexstruct> signature_local_to_element
    ( int dim,std::shared_ptr<multi_indexstruct> i ) {
  domain_coordinate_allones allones(dim);
  ioperator *times2 = new ioperator("*2");
  return std::shared_ptr<multi_indexstruct>
    ( new contiguous_multi_indexstruct
      ( i->first_index_r()*2,i->last_index_r()*2+allones ) );
}
```

and the local execution:

```
%% lulesh_functions.cxx
for (index_int i=pfirst[0]; i<=plast[0]; i++)
  for (index_int j=pfirst[1]; j<=plast[1]; j++) {
    index_int IJ = INDEX2D(i,j,out_offsets,out_nsize);
    outdata[IJ] =
      ( indata[INDEX2D(2*i,  2*j,   in_offsets,in_nsize)] +
        indata[INDEX2D(2*i,  2*j+1, in_offsets,in_nsize)] +
        indata[INDEX2D(2*i+1,2*j,   in_offsets,in_nsize)] +
```

```
        indata[INDEX2D(2*i+1,2*j+1, in_offsets,in_nsize)]
        ) / 4.;
  }
```

2.6 Discussion

We conclude that, in the Cartesian case, all data transfers in Lulesh have a signature that is easily given through a function recipe. Functionalitywise this means our problem is solved. Performancewise, we expect no difference between this an a regular MPI implementation.

2.6.1 Even easier description?

It is worth investigating if operations on distributions should be implemented: 'multiply' for step 4, 'inversion' for step 1, and 'composition' for step 2.

2.6.2 Multiple materials

A domain can host multiple materials; typically a material lives only on a subset of elements, though individual elements can have more than one material. We realize this by putting a mask on the element distribution.

2.6.3 Performance

Sometimes the best setup is architecture dependent [2]. This seems mostly to be a matter of the amount of overdecoposition in threading. We can easily accomodate this.

References

[1] Victor Eijkhout. IMP distribution theory. Technical Report IMP-01, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014. (Included in source code repository.).

[2] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[3] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.