TACC Technical Report IMP-25

Dense Linear Algebra in IMP

Victor Eijkhout*

September 27, 2016

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

Abstract

Given how much IMP has been motivated by stencil and grid operations, the extension to dense linear algebra could use some explanation.

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)

1 Tools for Cartesian parallelism

The initial Integrative Model for Parallelism (IMP) theory was described in terms of linearized indices. While it is possible to treat matrices and higher dimensional objects that way, doing so becomes increasingly forced. Hence we have implemented tools for multi-dimensional parallelism.

We have the following classes:

- `processor_coordinate`: a multi-dimensional processor number. The number of dimensions is a user input.
  ```
  %% imp_base.cxx
  processor_coordinate::processor_coordinate(int dim) {
    for (int id=0; id<dim; id++)
      coordinates.push_back(-1);
  };
  ```
- `decomposition`: a Cartesian decomposition of the computational domains and their assignment to the processors.
  ```
  %% imp_base.cxx
  decomposition::decomposition( architecture *arch,processor_coordinate *sizes )
    : architecture(arch),entity(entity_cookie::DECOMPOSITION) {
    int dim = sizes->get_dimensionality();
    if (dim<=0)
      throw(std::string("Non-positive decomposition dimensionality"));
    domain_layout = new processor_coordinate(sizes);
  };
  ```
- `multi_indexstruct`: for now we assign to each processor a structure that is the product of a `indexstruct` in each dimension.
- `parallel_structure`: description of the `multi_indexstruct` objects of all processors.

2 Parallel transpose

Rather than describing the signature function of the operation, we construct the beta distribution explicitly from the alph:

```
%% unittest_struct.cxx
parallel_structure *transstruct;
REQUIRE_NOTHROW( transstruct = new parallel_structure(mdecomp) );
CHECK( transstruct->get_dimensionality()==dim );
transstruct->set_name("transposed structure");
for (int p=0; p<ntids; p++) {
  fmt::print("\n");
  processor_coordinate *pcoord;
  REQUIRE_NOTHROW( pcoord = mdecomp->coordinate_from_linear(p) );
```

```
  processor_coordinate *pflip;
  REQUIRE_NOTHROW( pflip = new processor_coordinate
     ( std::vector<int>{pcoord->coord(1),pcoord->coord(0)} ) );
  REQUIRE_NOTHROW
    ( transstruct->set_processor_structure(pcoord,d->get_processor_structure(pflip)) );
}
```

References