TACC Technical Report IMP-13

Sparse Operations in the Integrative Model for Parallelism

Victor Eijkhout*

February 22, 2016

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

Abstract

We consider the sparse matrix vector product.

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
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 generation (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)

1 Introduction

In the formal treatment of Integrative Model for Parallelism (IMP) (see IMP-03) we introduced the 'indirect' function Ind mapping output indices to required input indices.

> Let $x, y \in$ Array and let the function f compute each element of y from certain elements of x. We use a function σ (or σ_f to indicate its provenance explicitly) called the 'signature function' to determine the mapping from output indices to input sets of indices:

Datatype Signature: Signature of data parallel functions

$$\text{Signature} \equiv N \rightarrow \text{Ind}$$

As remarked, this function can be represented as a boolean sparse matrix, and for the case where f computes a matrix-vector product, that boolean matrix corresponds to the sparsity pattern of f.

2 Implementation

We define the sparse matrix-vector product as inheriting from a regular kernel, by setting the sparse matrix as both the pattern that will determine the beta distribution, and as context for the local function:

```
class mpi_spmvp_kernel : virtual public mpi_kernel {
public:
  mpi_spmvp_kernel( object *in,object *out,mpi_sparse_matrix *mat)
    : kernel(in,out),mpi_kernel(in,out) {
    set_name(fmt::format("sparse-mvp{}",get_out_object()->get_object_number()));
    dependency *d = last_dependency();
    d->set_index_pattern( mat );
    set_localexecutefn( &local_sparse_matrix_vector_multiply );
    localexecutectx = (void*)mat;
  };
  virtual void analyze_dependencies() override {
    mpi_kernel::analyze_dependencies();
    mpi_sparse_matrix *mat = (mpi_sparse_matrix*)localexecutectx;
    if (mat->get_trace())
      fmt::print("[{}] {}\n",get_out_object()->get_architecture()->mytid(),mat->as_string());
  };
};
```

The pattern is stored as as the local part of the beta distribution:

```
      for (auto dom : gamma->get_domains()) {
indexstruct *base = gamma->get_pstruct(dom);
indexstruct *columns = pattern->all_columns_from(base);
```

```
beta_struct->set_pstruct( dom,columns );
      }
```

which means that dependency analysis can be done as for an explicit beta:

```
%% imp_base.cxx
std::vector<message*> *distribution::messages_for_segment
        ( int mytid,int skipself,indexstruct *beta_block,indexstruct *halo_struct ) {
  index_int g,g_last; int P = this->global_ndomains();
  std::vector<message*> *messages = new std::vector<message*>;
  messages->reserve(P);
  indexstruct *buildup = new empty_indexstruct();
  int pstart = mytid;
  if (get_architecture()->has_random_sourcing())
    pstart += rand()%P;
  for (int ip=0; ip<P; ip++) {
    int p = (pstart+ip)%P; // start with self first
    if (!lives_on(p)) continue; // deal with masks
    std::shared_ptr<indexstruct> intersect { beta_block->intersect( processor_structure(p) ) };
    if (intersect->local_size()>0 && !(skipself && p==mytid) && !buildup->contains(intersect.get())) {
      message *m = new message(p,mytid,intersect);
      if (get_architecture()->halo_has_local_addres_space) m->relativize(halo_struct);
      messages->push_back( m );
      buildup = buildup->struct_union(intersect.get());
      if (buildup->contains(beta_block)) goto covered; //->equals(buildup)) goto covered;
    }
  }
```

3 Remapping and caching

In the class definition of `mpi_spmvp_kernel` above we did not yet note the overloaded definition of `analyze_dependencies`: this includes a call to remap the matrix:

```
%% mpi_base.cxx
void mpi_sparse_matrix::remap( distribution *alpha,distribution *beta,int mytid ) {
  if (has_been_remapped) return;

  indexstruct *column_indices = beta->processor_structure(mytid);
  sparse_matrix::remap( 0,alpha->local_size(mytid)-1, column_indices );

  has_been_remapped = 1;
};
```

where MPI remapping is based on an internal routine:

```
void sparse_matrix::remap( index_int first, index_int last,indexstruct *column_indices ) {
  for (index_int irow=first; irow<=last; irow++) {
    indexstruct *oldrow  = rows[irow], *newrow;
```

```
    if (!rows[irow]->is_indexed())
      oldrow = rows[irow]->convert_to_indexed();

    newrow = oldrow->make_clone();
    for (index_int j=0; j<oldrow->local_size(); j++) {
      index_int col = oldrow->get_ith_element(j);
      index_int l;
      try { l = column_indices->find(col); }
      catch (std::string c) { fmt::printf("Error <<{}>>\n",c);
      newrow->set_ith_element(j,l);
    }
    rows[irow] = newrow;
  }
};
```