## TACC Technical Report IMP-18

# A tutorial introduction to IMP software

Victor Eijkhout*

November 7, 2016



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

**Abstract**

An introduction to the interface.

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 Introduction

In all of the prototypes and code examples, the string 'IMP' stands for 'put mpi or omp here, depending on how you want your program executed.

# 2 Concepts

## 2.1 Environments

With the environment you tell IMP how the program is going to be interpreted. If you specify an mpi_environment, IMP will call MPI_Init and see how many processors there are; with an omp_environment IMP will look at the OMP_NUM_THREADS environment variable; et cetera.

```
IMP_environment(int argc,char **argv);
```

## 2.2 Distributions

Distributions indicate how a set of $N$ indices is distributed over the processors. The easiest way is to use

```
IMP_distribution(environment *env,
    const char *type,index_int g)
```

where

- the environment was created as described in section 2.1;
- the type argument specifies the type of distribution; and
- the final parameter (of type long int) is the global size of the index set.

There are various more specific routines; for instance

```
IMP_distribution(environment *env,
    const char *type,index_int l,index_int g)
```

can specify the local size (per node or thread) through the l parameter.

## 2.3 Objects

After the relative intricacies of defining distributions, objects are easy, since the only thing they do is allocating the memory needed based on the distributions:

```
IMP_object( distribution *d );
IMP_object( distribution *d, double *x );
```

where the second call takes an array *you* supply.

## 2.4 Kernels

Kernels are the data parallel operations that take an object and make another object from it.

```
IMP_kernel( object *in,object *out );
```

Unfortunately, there is a lot more to this story: you need to supply the *signature* of the kernel, and the *local function* that is executed by each processor or thread.

### 2.4.1 Kernel signature

Let's say you have two objects `obj1,obj2` and you have defined

```
kernel *k = new IMP_kernel(obj1,obj2);
```

Now you need to specify the signature, which describes the data dependency structure of the kernel.

The easiest case is a 'conveniently parallel' function: index $i$ of the output is determined by only index $i$ of the input. In this case, you write

```
k->set_type_local();
```

However, in general index $i$ of the output will come from more than one index of the input. For instance, if it needs $i-1$ of the input, you write

```
k->add_beta_oper( new ioperator( "<<1" ) );
```

and similarly for $i+1$

```
k->add_beta_oper( new ioperator( ">>1" ) );
```

For sparse matrix multiplication, the beta distribution can be found from the actual sparse matrix:

```
d->set_index_pattern( mat );
```

The final case we mention is that of collectives. In that case the $\beta$ and $\gamma$ distribution are identical, so you would specify

```
k->set_explicit_beta_distribution( obj2->get_distribution() );
```

### 2.4.2 Local function

The local function is something you write without using any further IMP tools. It needs to have a prototype

```
void localfunction(int step,int p,object *in,object *out,void *ctx);
```

The opaque context pointer can be specified for the kernel:

```
k->set_localexecutectx( (void*) &your_data_structure );
```

The code for local functions can be tricky, but using the following as template one can actually write a single function for both MPI and OpenMP mode:

```
%% imp_functions.cxx
void vecshiftrightbump(int step,processor_coordinate *p,
       std::vector<object*> *invectors,object *outvector,double *flopcount) {
  object *invector = invectors->at(0);
  distribution
    *indistro = dynamic_cast<distribution*>(invector),
    *outdistro = dynamic_cast<distribution*>(outvector);
  double
    *indata = invector->get_data(p),
    *outdata = outvector->get_data(p);

  multi_indexstruct
    *pstruct = outvector->get_processor_structure(p);
  domain_coordinate
    pfirst = pstruct->first_index_r(), plast = pstruct->last_index_r();

  multi_indexstruct
    *in_nstruct = invector->get_numa_structure(),
    *out_nstruct = outvector->get_numa_structure(),
    *in_gstruct = invector->get_global_structure(),
    *out_gstruct = outvector->get_global_structure();
  domain_coordinate
    in_nsize = in_nstruct->local_size_r(), out_nsize = out_nstruct->local_size_r(),
    in_offsets = invector->offset_vector(),
    out_offsets = outvector->offset_vector();

  index_int pfirst0 = pfirst[0];
  if (pfirst0==0) pfirst0++;
  for (index_int i=pfirst0; i<=plast[0]; i++) {
    index_int Iout = INDEX1D(i,out_offsets,out_nsize), Iin = INDEX1D(i,in_offsets,in_nsize);
    outdata[Iout] = indata[Iin-1];
  }
  index_int len = plast[0]-pfirst0;
  *flopcount += len;
}
```

### 2.5 Algorithms

Finally, kernels are stored in a data structure called an `algorithm`.

```
%% template_heat.cxx
algorithm *heat;
heat = new IMP_algorithm(decomp);

IMP_kernel *initialize = new IMP_origin_kernel(xs->at(0));
domain_coordinate *deltaloc = new domain_coordinate( std::vector<index_int>{0} );
initialize->set_localexecutefn
```

```
  ( [deltaloc] (int step,processor_coordinate *p,std::vector<object*> *in,object *out,double *flopcount
    vecdelta(step,p,in,out,flopcount,*deltaloc); } );
heat->add_kernel( initialize );
for (int step=0; step<nsteps; step++) {
  heat->add_kernel( new IMP_diffusion_kernel( xs->at(step),ys->at(step) ) );
  if (trace) {
    object *nrm = new IMP_object(scalar);
    kernel *xnrm = new IMP_norm_kernel( xs->at(step+1),nrm );
    heat->add_kernel( xnrm );
    heat->add_kernel( new IMP_trace_kernel(nrm,fmt::format("Norm at step {}",step+1)) );
  }
  heat->add_kernel( new IMP_copy_kernel( ys->at(step),xs->at(step+1) ) );
}
heat->analyze_dependencies();
heat->execute();
```

By having a separate analyze and execute stage, the IMP system can optimize the task graph, for instance to effect latency hiding.

## 3 Operations

Using the above mechanisms it is possible to write new kernels, but a number of kernels are already given as ready-to-use building blocks.

### 3.1 Base kernels

With the current program structure, only certain kernels are available as base classes, with the specific kernels derived from them:

```
class copy_kernel : virtual public kernel {
public:
  copy_kernel( object *in,object *out ) : kernel(in,out);
}
class axpy_kernel : virtual public kernel {
public:
  axpy_kernel( object *in,object *out,double *x ) : kernel(in,out);
}
class scale_kernel : virtual public kernel {
public:
  scale_kernel( double *a,object *in,object *out ) : kernel(in,out);
}
class scaledown_kernel : virtual public kernel {
public:
```

```
  scaledown_kernel( double *a,object *in,object *out ) : kernel(in,out);
}
class sum_kernel : virtual public kernel {
public:
  sum_kernel( object *in1,object *in2,object *out ) : kernel(in1,out);
}
class scalar_kernel : virtual public kernel {
public:
  scalar_kernel( object *in1,const char *op,object *in2,object *out ) : kernel(in1,out);
}
class axbyz_kernel : virtual public kernel {
protected:
public:
  axbyz_kernel( char op1,object *s1,object *x1,
char op2,object *s2,object *x2,object *out )
    : kernel(x1,out);
}
```

Given these base kernels, the derived kernels are easy to define, for instance:

```
%% mpi_ops.h
class mpi_copy_kernel : public mpi_kernel,public copy_kernel {
public:
  mpi_copy_kernel( object *in,object *out )
    : kernel(in,out),copy_kernel(in,out),mpi_kernel(in,out),
      entity(entity_cookie::KERNEL) {
  };
};
```

### 3.2 Derived kernels

The following kernels are more or less written from the ground up:

```
%% mpi_ops.h
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),entity(entity_cookie::KERNEL) {
    set_name(fmt::format("sparse-mvp{}",get_out_object()->get_object_number()));
    dependency *d = last_dependency();
    d->set_index_pattern( mat );
    set_localexecutefn
      ( [mat] (int step,processor_coordinate *p,std::vector<object*> *inobjects,object *outobject,doubl
      return local_sparse_matrix_vector_multiply(step,p,inobjects,outobject,(void*)mat,cnt); } );
  };
  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()->mytid(),mat->as_string());
  };
};
%% mpi_ops.h
class mpi_sidewaysdown_kernel : virtual public mpi_kernel {
private:
  distribution *level_dist,*half_dist;
  mpi_object *expanded,*multiplied;
  mpi_kernel *expand,*multiply,*sum;
public:
  mpi_sidewaysdown_kernel( object *top,object *side,object *out,mpi_sparse_matrix *mat )
    : kernel(top,out),mpi_kernel(top,out),entity(entity_cookie::KERNEL) {
%% mpi_ops.h
class mpi_centerofmass_kernel : virtual public mpi_kernel {
public:
  mpi_centerofmass_kernel(object *bot,object *top,int k)
    : kernel(bot,top),mpi_kernel(bot,top),entity(entity_cookie::KERNEL) {
    set_localexecutefn
      ( [k] (int step,processor_coordinate *p,std::vector<object*> *invectors,object *outvector,double
scansumk(step,p,invectors,outvector,k,cnt); } );
    last_dependency()->set_signature_function_function( &doubleinterval );
  };
  mpi_centerofmass_kernel(object *bot,object *top) :
    mpi_centerofmass_kernel(bot,top,1) {};
};
%% mpi_ops.h
class mpi_reduction_kernel : virtual public mpi_kernel {
private: // we need to keep them just to destroy them
  distribution *local_scalar; //,*gathered_scalar;
public:
  mpi_reduction_kernel( object *local_value,object *global_sum)
    : kernel(local_value,global_sum),mpi_kernel(local_value,global_sum),
      entity(entity_cookie::KERNEL){
    if (!global_sum->has_type_replicated())
      throw(fmt::format
    ("Reduction output needs to be replicated, not {}",global_sum->type_as_string()));
%% mpi_ops.h
    sumkernel = new mpi_kernel(local_value,global_sum);
    sumkernel->set_name
      (fmt::format("reduction:one-step-sum{}",get_out_object()->get_object_number()));
    sumkernel->last_dependency()->set_explicit_beta_distribution
      ( new mpi_gathered_distribution(decomp) );
```

```
    sumkernel->last_dependency()->set_is_collective
      ( decomp->has_collective_strategy(collective_strategy::MPI) );
    sumkernel->set_localexecutefn( &summing );
%% omp_ops.h
class omp_innerproduct_kernel : virtual public omp_kernel {
private:
  distribution *local_scalar,*gathered_scalar; // we need to keep them just to destroy them
  object *local_value;
protected:
  omp_kernel *prekernel,*sumkernel;
public:
  omp_innerproduct_kernel( object *v1,object *v2,object *global_sum)
    : omp_kernel(v1,global_sum),kernel(v1,global_sum),
      entity(entity_cookie::KERNEL)  {
    decomposition *decomp = global_sum;
    set_name("inner-product");

    local_scalar = new omp_scalar_distribution(decomp); //(v2,v2->global_ndomains());
    local_value = new omp_object(local_scalar);
    local_value->set_name("local-inprod-value");

    dependency *d;
    prekernel = new omp_kernel(v1,local_value);
    prekernel->set_name("local-innerproduct");
    d = prekernel->last_dependency();
    d->set_explicit_beta_distribution(v1);
    d->set_name("inprod wait for in vector");
    prekernel->add_in_object(v2);
    d = prekernel->last_dependency();
    d->set_explicit_beta_distribution(v2);
    d->set_name("inprod wait for second vector");
    prekernel->set_localexecutefn( &local_inner_product );

    sumkernel = new omp_reduction_kernel(local_value,global_sum);
  };
  virtual void split_to_tasks() override {
    split_contained_kernels(prekernel,sumkernel);
  };
  virtual void analyze_dependencies() override {
    analyze_contained_kernels(prekernel,sumkernel);
  }
  virtual void execute() { // synckernel->execute();
    prekernel->execute(); sumkernel->execute(); };
  kernel *get_prekernel() { return prekernel; };
};
```

# 4 Obscure stuff

## 4.1 Processor masks

Sometimes a kernel need not apply to all processors. Thus we have a mask to blank out processors. Note: the mask is actually the set of active processors.

- Class objects of type `processor_mask` can have processors added or subtracted.
- Masks can be added to a distribution by `dist->add_mask(m)`. Objects created on a masked distribution has no storage on excluded processors.
- You can also set a mask on a dependency object, which applies the mask to the beta object created.
- The task execution routine returns immediately if there is a mask on the output or the first halo object.