TACC Technical Report IMP-15

Collectives in the Integrative Model for Parallelism

Victor Eijkhout*

April 23, 2017

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

Abstract

abstract

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 Discussion

Collectives are an example of an operation that falls entirely in our α, β, γ framework, but for which the β-distribution is nothing like a traditional 'halo region'.

There is a whole science of collectives [1] which is largely outside our purview. The IMP model describes the semantics of collectives in the sense of giving their input/output specification; the implementation is left to a different level of discussion. However, we can make the following remarks:

- Since the user code calls a collective kernel, we can put an optimized implementation under this. However, collectives induce a synchronization point, which is outside of the IMP's analysis of dependencies. Therefore, the following might be attractive.
- We can treat a collective as a general operation, and just let it generate all the individual tasks and dependencies. This may be attractive in the context of highly asynchronous applications. This is the strategy taken in the examples below.
- Under the strict definition of the IMP model, assigning a data index to more than one process means that these processes have duplicate copies, redundantly calculated. This means that IMP reduction-type collectives on shared address spaces behave differently from the naive interpretation where only a single result is computed. To achieve this effect we would need a processor mask on the replicated-scalar distribution.

2 Local stages

A collective working on a distributed array starts with doing a local reduction. Since we assume in the rest of this report (and in our implementations) that we have only a single value per process, we get the local reduction out of the way here.

- We assume that the input distribution is disjoint.
- The resulting distribution has a single index per process.
- Rather than specifying a signature function, we indicate that no communication is needed, by stating that $\beta = \alpha$.
- We then execute a local function on the input data.

For instance, for a norm kernel this local stage looks like:

```
%% imp_ops.h
norm_kernel( object *in,object *out )
  : kernel(in,out),entity(entity_cookie::KERNEL) {
  set_cookie(entity_cookie::SHELLKERNEL);
  int step = out->get_object_number();
  set_name(fmt::format("norm{}",get_out_object()->get_object_number()));

  {
    distribution *local_scalar = out->new_scalar_distribution();
```

```
    local_value = out->new_object(local_scalar);
  }
  local_value->set_name(fmt::format("local-norm-value{}",step));

  prekernel = out->kernel_from_objects(in,local_value);
  prekernel->set_name
    (fmt::format("local-norm-compute{}",get_out_object()->get_object_number()));
  prekernel->set_explicit_beta_distribution( in );
  prekernel->set_localexecutefn( &local_normsquared );
```

3 Implementations

For the moment we limit ourselves to 'all'-collectives. These are easier to describe than rooted ones, since they are more symmetric.

3.1 Allgather

The input for an allgather can be described as an array of P elements, one per processor, making the α-distribution

$$\alpha\colon p \mapsto p.$$

The result of the allgather leaves all P elements on each processor, so

$$\gamma\colon p \mapsto P$$

and

$$\sigma(p) = P \qquad \text{so} \quad \sigma(P) = P$$

and

$$\beta = \sigma\gamma = \gamma.$$

3.2 Reduction

For now we implement a reduction through an allgather followed by a local sum.

```
%% mpi_ops.h
class mpi_reduction_kernel : virtual public mpi_kernel {
private: // we need to keep them just to destroy them
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) {
    fmt::print("Mpi reduction of type {}\n",global_sum->strategy_as_string());
    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 );
```

We have various ways of implementing this.

3.2.1 Grouped reduction

The naive approach gives P^2 all-to-all message. As a first improvement, we do a grouped approach that gives $P\sqrt{P}$ messages:

```
%% mpi_ops.h
int
  P=decomp->domains_volume(), ntids=P, mytid=decomp->mytid(), g=P-1;
int groupsize = 4*( (sqrt(P)+3)/4 );

int
  mygroupnum = mytid/groupsize, nfullgroups = ntids/groupsize,
  grouped_tids = nfullgroups*groupsize, // how many procs are in perfect groups?
  remainsize = P-grouped_tids, ngroups = nfullgroups+(remainsize>0);
parallel_structure *groups = new parallel_structure(decomp);
try {
  for (int p=0; p<P; p++) {
auto pcoord = decomp->coordinate_from_linear(p);
index_int groupnumber = p/groupsize,
  f = groupsize*groupnumber,l=MIN(f+groupsize-1,g);
groups->set_processor_structure
  (pcoord,
   std::shared_ptr<multi_indexstruct>
   ( new multi_indexstruct
     ( std::vector<std::shared_ptr<indexstruct>>{
       std::shared_ptr<indexstruct>( new contiguous_indexstruct(f,l) ) } ) )
   );
  }
  groups->set_type( groups->infer_distribution_type() );
} catch (std::string c) {
```

```
  throw(fmt::format("Local grouping distribution failed: {}",c)); }

parallel_structure *partials = new parallel_structure(decomp);
try {
  for (int p=0; p<P; p++) {
index_int groupnumber = p/groupsize;
partials->set_processor_structure
  (p, std::shared_ptr<indexstruct>( new contiguous_indexstruct(groupnumber) ) );
  }
  partials->set_type( partials->infer_distribution_type() );
} catch (std::string c) {
  throw(fmt::format("Error <<{}>> in partial reduction",c)); }
```

3.2.2 Recursive reduction

```
%% mpi_ops.h
for (nlevels=1; nlevels<100; nlevels++) {
  index_int lsize = cur_level->outer_size();
  fmt::print("recursive level {}: {}\n",nlevels,cur_level->as_string());
  if (lsize==1) break; // we're done; discard the operate distribution and put the outobject
  levels.push_back( cur_level );
  objects.push_back( cur_object );
  try {
cur_level = cur_level->operate(div2);
cur_object = cur_level->new_object(cur_level);
  } catch (std::string c) {
fmt::print("Error <<{}>> on dist <<{}>> at level {}\n",c,cur_level->as_string(),nlevels);
  throw(std::string("Setup of recursive reduction failed")); }
}
if (levels.size()==0) { // this happens with one processor: we already have a scalar
  fmt::print("recursive reduction, degenerate case\n");
  levels.push_back( cur_level );
  objects.push_back( cur_object );
}
levels.push_back( dynamic_cast<distribution*>(global_sum) );
objects.push_back( global_sum );
```

3.2.3 MPI collective reduction

3.3 Inner product

For now we implement an inner product as an allgather followed by a local sum.

3.4 Outer product

We call 'outer product' the operation

$$A[i,j] = B[i] \cdot C[j].$$

In its simplest form, C is redundantly stored, and B is properly distributed. We then store A similar to B, with the dimension of C as the 'orthogonal dimension'.

```
mpi_distribution(environment *env,const char *type,
  index_int ortho, index_int local, index_int global);
```

An outer product implementation is available for this case:

```
%% mpi_ops.h
mpi_outerproduct_kernel( object *in,object *out,
    object *replicated,
    void (*f)(int, processor_coordinate*, std::vector<object*>*, object*, double*)
  )
  : mpi_kernel(in,out),entity(entity_cookie::KERNEL) {
  last_dependency()->set_explicit_beta_distribution(in);
  add_in_object(replicated);
  last_dependency()->set_explicit_beta_distribution(replicated);

  set_name(fmt::format("outer product{}",get_out_object()->get_object_number()));
  set_localexecutefn(f);
};
```

3.5 Broadcast

Broadcast is an interesting case because it starts with a replicated distribution, and goes to a non-replicated one. In the single scalar case:

$$\alpha\colon p \mapsto \{0\}$$
$$\gamma\colon p \mapsto \{p\}$$

One easily sees that this is not possible. So we fake a solution by having $\beta = \alpha$ and using a simple copy as local function.

```
%% imp_ops.h
class bcast_kernel : virtual public kernel {
public:
  bcast_kernel( object *in,object *out ) : kernel(in,out),entity(entity_cookie::KERNEL) {
    if (in==nullptr) throw(std::string("Null in object in copy kernel"));
    if (out==nullptr) throw(std::string("Null out object in copy kernel"));

    set_name(fmt::format("bcast{}",get_out_object()->get_object_number()));
    dependency *d = last_dependency();
```

```
    d->set_explicit_beta_distribution(out);
    set_localexecutefn( &crudecopy );
  };
};
%% mpi_ops.h
class mpi_bcast_kernel : public mpi_kernel,public bcast_kernel {
public:
  mpi_bcast_kernel( object *in,object *out )
    : kernel(in,out),bcast_kernel(in,out),mpi_kernel(in,out),
      entity(entity_cookie::KERNEL) {
  };
};
%% mpi_functions.cxx
void crudecopy(int step,processor_coordinate *p,std::vector<object*> *invectors,object *outvector,void
{
  object *invector = invectors->at(0);
  double *indata = invector->get_data(p);
  double *outdata = outvector->get_data(p);

  int ortho = 1;
  if (ctx!=nullptr) ortho = *(int*)ctx;

  index_int n = outvector->volume(p);

  for (index_int i=0; i<n*ortho; i++) {
    outdata[i] = indata[i];
  }
  *flopcount += n*ortho;
}
```

References

[1] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 2007.