TACC Technical Report IMP-11

A gentle introduction to the Integrative Model for Parallelism

Victor Eijkhout[*]

May 9, 2017

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

Abstract

In this report we give the philosophy, the basic concepts, and some demonstration, of the Integrative Model for Parallelism (IMP). We show that a judicious design of programming abstractions can lead to a system that accomplishes the holy grail of parallel programming:

1. High level expression,
2. translating to efficient use of low level primitives,
3. in a wide range of applications.

It does this through a new theoretical formulation of parallel computing.

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

In this report we describe the Integrative Model for Parallelism (IMP). This model purports to be a better solution to parallel programming in HPC than currently existing models.

IMP promises the following:

- Ease of programming: an IMP program is essentially sequential, with the user specifying algorithm steps in high level terms operating on distributed objects.
- Mode-independent parallel programming: the same program runs with MPI on distributed memory, task models on shared memory, or hybrid combinations of these.
- Asynchronous execution: despite the sequential user language, IMP generates a realization of the code in terms of asynchronously executing tasks, only governed by their dependencies.
- Latency-tolerance: the IMP system can analyze codes to realize overlap of communication and computation, or even do a 'communication avoiding' analysis.

In the remainder of this introduction we motivate some of the choices made in the Integrative Model, while in later sections we discuss the basic concepts and some practical results.

1.1 Motivation from programmability

We argue that many parallel programming systems are too general, and closely inspired by the target hardware. For instance message passing is made necessary by the presence of distributed memory, while Directed Acyclic Graph (DAG) models require shared memory and are naturally attractive in threaded environments. Both these models have practical disadvantages: the whole program behaviour is a non-trivial combination of the behaviours of the strands of execution that the user codes. Larus and Sutter phrase it as follows [5]:

> [H]umans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations.

On the other hand, on a conceptual level, a parallel program can be fairly simple. For instance, a matrix-vector or matrix-matrix multiplication is conceptually a single, even data parallel, operation. All the intricacies of programming the data distributions needed for a scalable execution of such operations are really implementation details, and ought not to concern the user.

The conceptual attraction of such an approach is eloquently formulated in [4]:

> [A]n HPF program may be understood (and debugged) using sequential semantics, a deterministic world that we are comfortable with. Once again, as in traditional programming, the programmer works with a single address space, treating an array as a single, monolithic object, regardless of how it may be distributed across the memories of a parallel machine. The programmer does specify data distributions, but these are at a very high level and only have the status of hints to the compiler, which is responsible for the actual data distribution[.]

Thus we argue for a programming model that is on an essentially higher level than the execution model. The programming model should be more than syntactic sugar around the execution primitives: it should be expressed on a level closer to application terms.

1.2 Realization

Of course, designing a system with sequential semantics is not easy. The above mentioned HPF was a notable failure [2] for a variety of reasons, and several other proposed languages do not seem to scale well to irregular codes on clusters without further provisions.

IMP successfully realizes this goal, based on the following idea.

- IMP uses dataflow as an Intermediate Representation. Dataflow can serve as an expression of algorithms that enables a wide range of parallelism modes; however, instead of programming the dataflow formulation explicitly the IMP system derives it from the programmed description.
- Data in the IMP system is organized by distributions. However, our notion of distribution more generally than generally used. In particular distribution can be dynamically generated by the system. We will argue later that this enables the essential theoretical innovation behind IMP.
- In common with several programming systems based on task DAGs, we use an approach where program steps are separately declared, analyzed, and executed. This is also known as 'inspector-executor', and it used in the sparse components of libraries such as PETSc and Trilinos.

We will motivate the IMP concepts in section 2 with a sketch of a formalization. The realization of this in software is described in section 3.

2 A motivating example for the basic concepts

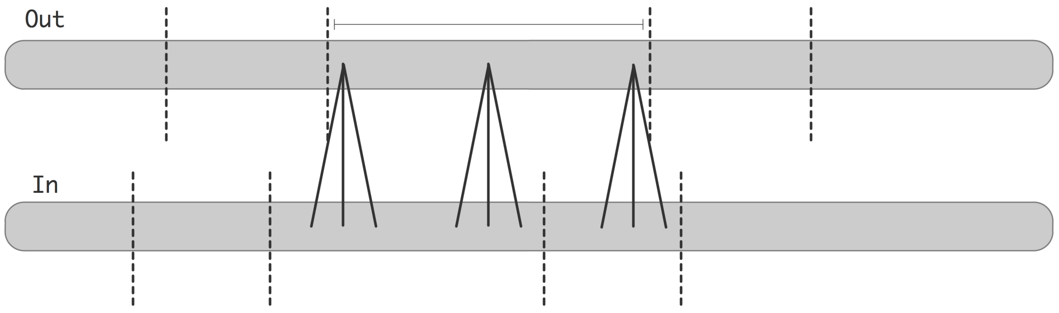
We consider a simple data parallel example, and show how it leads to the basic distribution concepts of IMP: the three-point operation

$$\forall_i \colon y_i = f(x_i, x_{i-1}, x_{i+1})$$

which describes for instance the 1D heat equation

$$y_i = 2x_i - x_{i-1} - x_{i+1}.$$

(Stencil operations are much studied; see e.g., [6] and the polyhedral model, e.g., [1]. However, we claim far greater generality for our model.) We illustrate this graphically by depicting the input and output vectors, stored distributed over the processors by contiguous blocks, and the three-point combining operation:

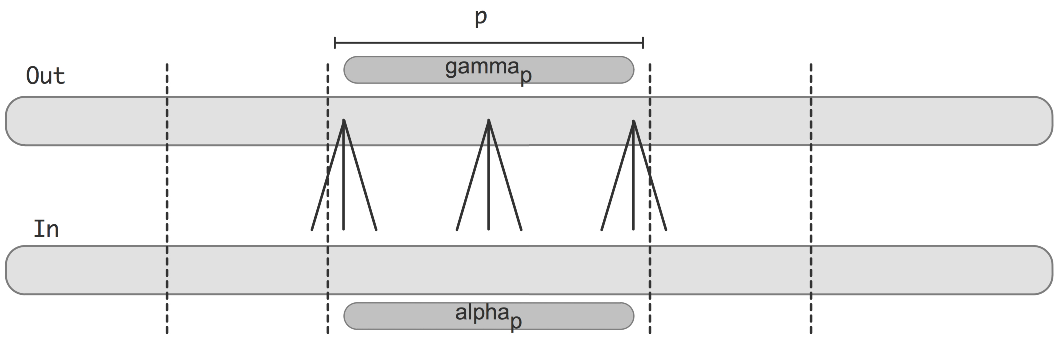


The distribution indicated by vertical dotted lines we call the α-distribution for the input, and the γ-distribution for the output. These distributions are mathematically given as an assignment from processors to sets of indices:

$$\alpha\colon p \mapsto [i_{p,\min},\ldots,i_{p,\max}].$$

The traditional concept of distributions in parallel programming systems is that of an assignment of data indices to a processor, reflecting that each index 'lives on' one processor, or that that processor is responsible for computing that index of the output. We turn this upside down: we define a distribution as a mapping from processors to indices. This means that an index can 'belong' to more than one processor. (The utility of this for redundant computing is obvious. However, it will also seen to be crucial for our general framework.)

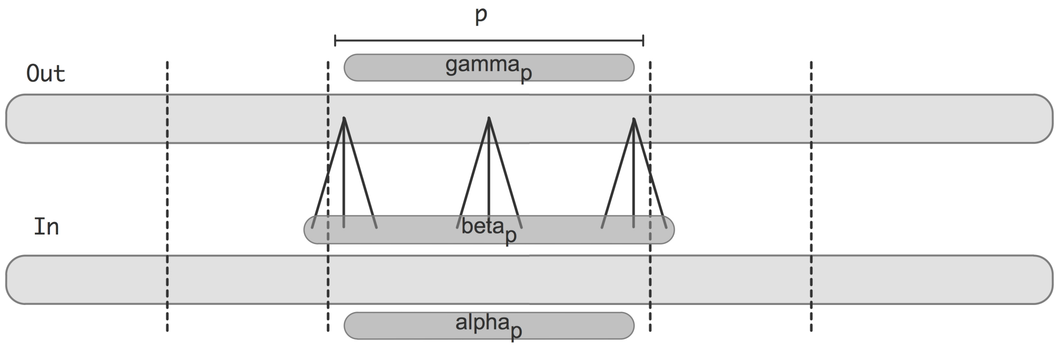
For purposes of exposition we will now equate the input α-distribution and the output γ-distribution, although that will not be necessary in general.



This picture shows how, for the three-point operation, some of the output elements on processor p need inputs that are not present on p. For instance, the computation of y_i for $i_{p,\min}$ takes an element from processor $p-1$. This gives rise to what we call the β-distribution:

$\beta(p)$ is the set of indices that processor p needs to compute the indices in $\gamma(p)$.

The next illustration depicts the different distributions for one particular process:



Observe that the β-distribution, unlike the α and γ ones, is not disjoint: certain elements live on more than one processing element. It is also, unlike the α and γ distributions, not specified by the programmer: it is derived from the γ-distribution by applying the shift operations of the stencil. That is,

The β-distribution brings together properties of the algorithm and of the data distribution.

We will formalize this derivation below.

2.1 Deriving the dataflow formulation

This gives us all the ingredients for reasoning about parallelism. Defining a *kernel* as a mapping from one distributed data set to another, and a *task* as a kernel on one particular process(or), all data dependence of a task results from transforming data from α to β-distribution. By analyzing the relation between these two we derive at dependencies between processors or tasks: each processor p depends on some predecessors q_i, and this set of predecessors can be derived from the α,β distributions: q_i is a predecessor if

$$\alpha(q_i)\cap\beta(p)\neq\emptyset.$$

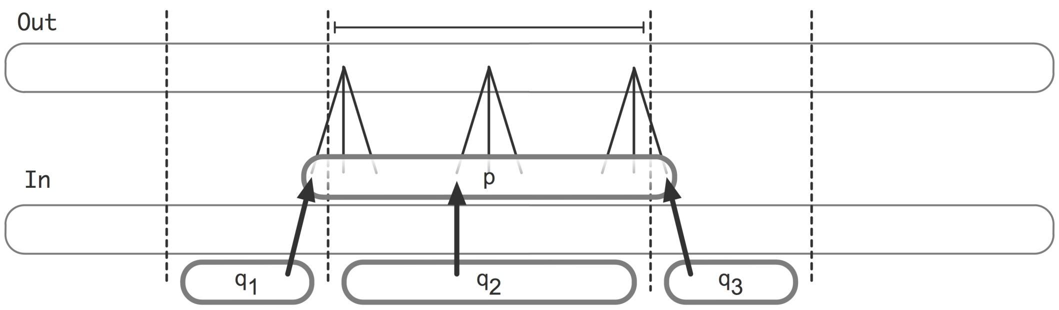


Figure 1 illustrates this: the left DAG is the sequential program of a heat equation evolution; the right DAG is the dataflow representation derived when this sequential program is run on six processors.

2.2 Parallelism-mode independence through dataflow

The term 'dataflow' comes with lots of baggage. However, in the abstract, a dataflow formulation does not imply any specific hardware model. In fact, we argue that by deriving (not programming!) a dataflow formulation of algorithms we can achieve mode-independent parallel programming.

First of all: in message passing, these dataflow dependences obviously corresponds to actual messages: for each process p, the processes q that have elements in $\beta(p)$ send data to p. (If $p = q$, of course at most a copy is called for.)

Secondly, in shared memory we don't interpret distributions with a physical distribution, but rather with a logical one: we identify the α-distribution on the input with tasks that produce this input. The β-distribution then describes what input-producing tasks a task p is dependent on. The dataflow description then becomes a task dependency graph.

3 Practical realization

3.1 Programming the model

In our motivating example we showed how the concept of 'β-distribution' arises, and how from it we can derive messages and task dependencies.

We will now argue the practicality of this concept. From the motivation example, it is clear that the β-distribution generalizes concepts such as the 'halo region' in distributed stencil calculations, but its applicability extends to all of (scientific) parallel computing. We will touch on this in section 5.

It remains to be argued that the β distribution can actually be used as the basis for a software system. To show this, we associate with the function f that we are computing an expression of

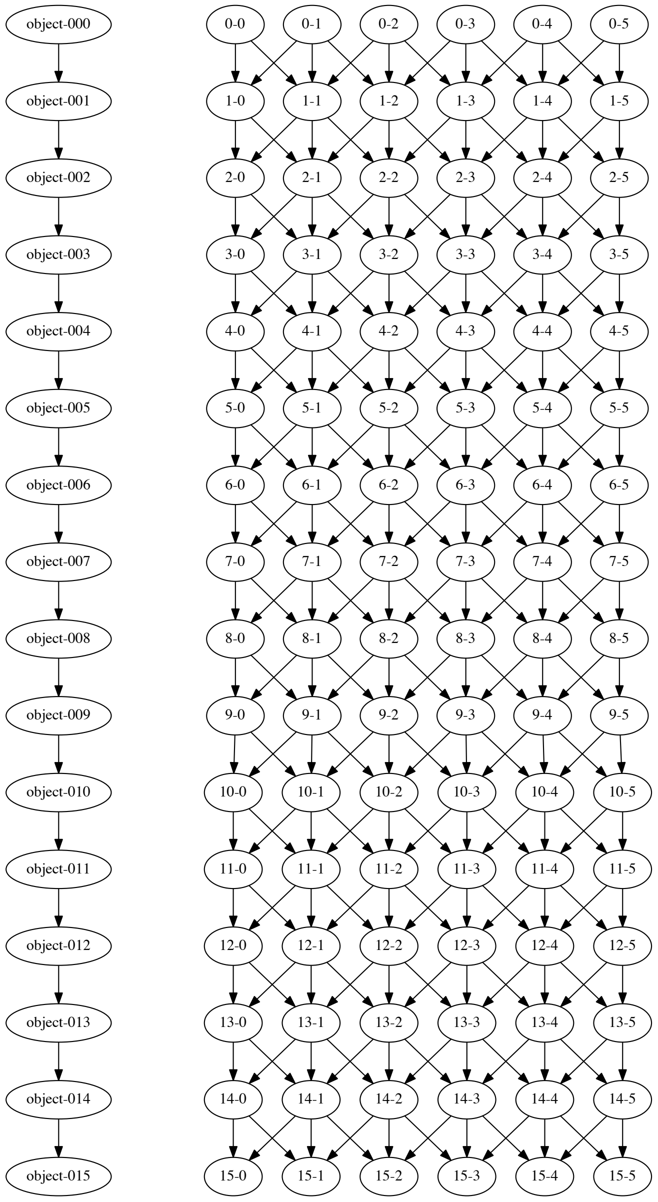


Figure 1: Kernel (left) and task relations (right) for the one-dimensional heat equation, executing 15 steps on 6 processors.

the algorithm (not the parallel!) data dependencies, called the ‘signature function’, denoted σ_f. For instance for the computation of $y_i = f(x_i, x_{i-1}, x_{i+1})$, the signature function is

$$\sigma_f(i) = \{i, i-1, i+1\}.$$

With this, we state (without proof; for which see section **??** and [**?**]) that

$$\beta = \sigma_f(\gamma).$$

It follows that, if the programmer can specify the data dependencies of the algorithm, a compiler/runtime system can derive the β distribution, and from it, task dependencies and messages for parallel execution.

Specifying the signature function is quite feasible, but the precise implementation depends on the context. For instance, for regular applications we can adopt a syntax similar to stencil compilers such as the Pochoir compiler [6]. For sparse matrix applications the signature function is isomorphic to the adjacency graph; for collective operations, $\beta = \gamma$ often holds; et cetera.

3.2 Software realization and proof of concept

In this section we present the outline of a software implementation of the above theoretical notions. At first we consider the motivating example, then we discuss in more detail the implementation of the I_f function.

3.2.1 Basic constructs

There are many ways of creating a distribution. The simplest is a block distribution with even distribution of points:

```
IMP_distribution *blocked =
      new IMP_block_distribution(decomp,globalsize);
```

(We do not go into the `decomp` object, which keeps track of processors, threads, and any topology on those.)

Given a distribution, creating an object is simple:

```
IMP_object *input_vector = new IMP_object( blocked );
IMP_object *output_vector = new IMP_object( blocked );
```

Defining a kernel takes a couple of steps. The basic definition takes two vectors; after that we specify the function that is executed locally after the $\alpha \rightarrow \beta$ communication:

```
IMP_kernel *update_step =
  new IMP_kernel(input_vector,output_vector);
update_step->localexecutefn = &threepoint_execute;
```

The trickiest part is specifying the mechanism by which the β-distribution is constructed. For a threepoint kernel that is by three shift operators:

```
update_step->add_beta_oper( new ioperator(">>1") );
update_step->add_beta_oper( new ioperator("<<1") );
update_step->add_beta_oper( new ioperator("none") );
```

For a general sparse matrix we let the beta distribution be derived from the adjacency matrix:

```
%% 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());
  };
};
```

3.2.2 Execution

The general execution mechanism collects the kernels into an algorithm:

```
IMP_algorithm* algorithm =
      new IMP_algorithm(decomp);
algorithm->add_kernel( step,update_step );
```

on which we perform analysis, and which, by the inspector-executor model, can be executed multiple times:

```
algorithm->analyze_dependencies();
algorithm->execute();
```

3.2.3 Local code: global programming

So far we have not addressed the local function that is evaluated in a kernel. In keeping with the mode-independent nature of our model, this function is also specified without reference to the nature of parallelism.

To explain why the type of parallelism figures in this question, consider the threepoint kernel

$$y_i = x_{i-1} + x_i + x_{i+1}$$

which would be coded sequentially as

```
for (i=0; i<N; i++)
  y[i] = x[i-1] + x[i] + x[i+1]
```

with some exception for the edge cases.

Using OpenMP tasks this would become:

```
ilo = lower_bound(p); ihi = upper_bound(p);
for (i=ilo; i<ihi; i++)
  y[i] = x[i-1] + x[i] + x[i+1]
```

where p is the abstract 'process number' as used in section 2.

In MPI the local indices would be start numbering at zero, giving code

```
myN = local_size(p);
for (i=0; i<myN; i++)
  y[i] = x[i-1] + x[i] + x[i+1]
```

except that on all but the first processor we have a halo region which is itself zero-based[1]:

```
myN = local_size(p);
for (i=0; i<myN; i++)
  y[i] = x[i] + x[i+1] + x[i+2]
```

The whole of this mess goes away in IMP: the code for a damped central difference is for instance:

```
%% imp_functions.cxx
void central_difference_damp
    (int step,processor_coordinate *p,
     std::vector<object*> *invectors,object *outvector,
     double damp,double *flopcount)
{
  double
    *outdata = outvector->get_data(p), *indata = invector->get_data(p);
  auto pstruct = outvector->get_processor_structure(p);
  domain_coordinate
    pfirst = pstruct->first_index_r(), plast = pstruct->last_index_r();
  index_int lo=pfirst[0],hi=plast[0];
  for (index_int i=lo; i<hi; i++) {
    index_int
      Iin  = INDEX1D(i,in_offsets,in_nsize),
      Iout = INDEX1D(i,out_offsets,out_nsize);
    outdata[Iout] = ( 2*indata[Iin] - indata[Iin-1] - indata[Iin+1] )
```

1. One of the very few cases where Fortran offers an easy way out!

```
                    *damp;
  }
  *flopcount += 4*(hi-lo+1);
```

where the offsets and bounds are boilerplate code that is independent of the parallelism mode.

4 Test application: Conjugate gradient

We have implemented a conjugate gradient algorithm. The following snippet shows how algorithm steps are added to the algorithm. Kernels such as innerproducts and vector updates are given here as user-level primitives; however, they are easily implemented in the IMP basic system as shown in examples in section 5.

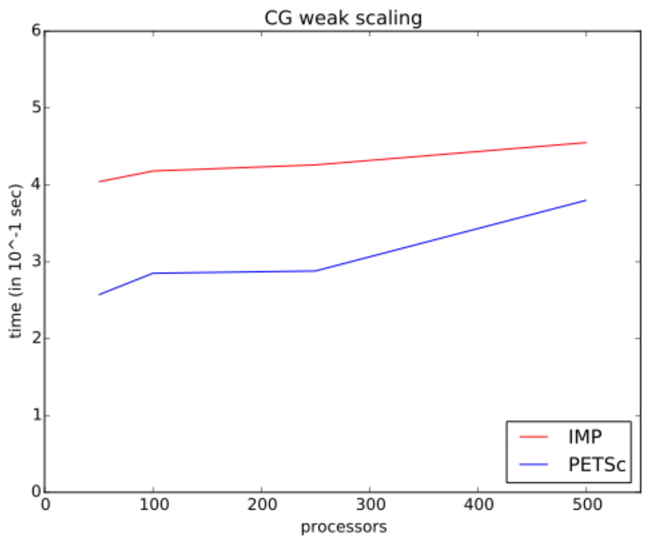
```
%% template_cgr.cxx
kernel *rnorm = new IMP_norm_kernel( r,rnorms[it] );
cg->add_kernel(rnorm); rnorm->set_name(fmt::format("r norm{}",it));
if (trace) {
  kernel *trace = new IMP_trace_kernel(rnorms[it],fmt::format("Norm in iteration {}",it));
  cg->add_kernel(trace); trace->set_name(fmt::format("rnorm trace {}",it));
}

kernel *precon = new IMP_preconditioning_kernel( r,z );
cg->add_kernel(precon); precon->set_name(fmt::format("preconditioning{}",it));

kernel *rho_inprod = new IMP_innerproduct_kernel( r,z,rr );
cg->add_kernel(rho_inprod); rho_inprod->set_name(fmt::format("compute rho{}",it));
if (trace) {
  kernel *trace = new IMP_trace_kernel(rr,fmt::format("rtz in iteration {}",it));
  cg->add_kernel(trace); trace->set_name(fmt::format("rtz trace {}",it));
}

if (it==0) {
  kernel *pisz = new IMP_copy_kernel( z,pcarry );
  cg->add_kernel(pisz); pisz->set_name("copy z to p");
} else {
  kernel *beta_calc = new IMP_scalar_kernel( rr,"/",rrp,beta );
  cg->add_kernel(beta_calc); beta_calc ->set_name(fmt::format("compute beta{}",it));

  kernel *pupdate = new IMP_axbyz_kernel( '+',one,z, '+',beta,p, pcarry );
  cg->add_kernel(pupdate); pupdate->set_name(fmt::format("update p{}",it));
}

rrp = new IMP_object(scalar); rrp->set_name(fmt::format("rho{}p",it));
kernel *rrcopy = new IMP_copy_kernel( rr,rrp );
cg->add_kernel(rrcopy); rrcopy->set_name(fmt::format("save rr value{}",it));
```

```
kernel *matvec = new IMP_centraldifference_kernel( pcarry,q );
cg->add_kernel(matvec); matvec->set_name(fmt::format("spmvp{}",it));

kernel *pap_inprod = new IMP_innerproduct_kernel( pcarry,q,pap );
cg->add_kernel(pap_inprod); pap_inprod->set_name(fmt::format("pap inner product{}",it));

kernel *alpha_calc = new IMP_scalar_kernel( rr,"/",pap,alpha );
cg->add_kernel(alpha_calc); alpha_calc->set_name(fmt::format("compute alpha{}",it));

kernel *xupdate = new IMP_axbyz_kernel( '+',one,x, '-',alpha,pcarry, xcarry );
cg->add_kernel(xupdate); xupdate->set_name(fmt::format("update x{}",it));

kernel *rupdate = new IMP_axbyz_kernel( '+',one,r, '-',alpha,q, rcarry );
cg->add_kernel(rupdate); rupdate->set_name(fmt::format("update r{}",it));
```

The algorithm is analyzed and executed:

```
algorithm->analyze_dependencies();
algorithm->optimize();
algorithm->execute();
```

A preliminary test shows scaling comparable to PETSc:



The performance is lower than PETSc, but this is due to the fact that the IMP code has very little hardwired and almost everything derived from the algorithm. This leaves plenty of space for optimizations.

We briefly remark on collectives. Semantically, IMP will interpret a collective such as an all-reduce as the sum of its dependencies, and therefore implement it as as series of sends and receives. This is very inefficient, so we added a grouping mechanism to IMP:

- an all-reduce starts as all-to-all sends and receives in process groups,
- followed by all-to-all reducing the groups.

Effectively, this reduces the $O(P^2)$ messages of the strict implementation to $O(P\sqrt{P})$, where optimally it would be $O(P\log_2 P)$.

- Part of our performance loss is due to this log-versus-root order of complexity;
- with some more sophistication we can achieve the logarithm;
- our approach has the advantage that all operations in the reduction are actually tasks in the IMP model.

5 Applications of the IMP model

In this section we will show how the basic elements of the IMP system can be used to define the user-level kernels for various algorithms.

5.1 Flexible treatment of distributions

Simple operations such as vector addition become complicated if the vectors concerned do not have the same distribution. For instance, the PETSc library requires matrices and vectors that appear in one operation, such as `MatMult` or `VecAXPY`, to be identically distributed.

In IMP, a vector addition would mathematically be

$$y(d) \leftarrow x_1(d) + x_2(d)$$

where d is a distribution. If the α-distributions of x_1, x_2 are not d, data motion is automatically generated.

For instance, this is the definition of the vector sum operation:

```
%% imp_ops.h
class sum_kernel : virtual public kernel {
public:
  sum_kernel( object *in1,object *in2,object *out )
    : kernel(in1,out),entity(entity_cookie::KERNEL) {
    if (in1==nullptr) throw(std::string("Null in1 object in sum kernel"));
    if (in2==nullptr) throw(std::string("Null in2 object in sum kernel"));
    if (out==nullptr) throw(std::string("Null out object in sum kernel"));

    dependency *d;
    set_name(fmt::format("vector sum{}",get_out_object()->get_object_number()));
    d = last_dependency(); d->set_explicit_beta_distribution(out);
    add_in_object(in2);
    d = last_dependency(); d->set_explicit_beta_distribution(out);
    localexecutefn = &vectorsum;
  }
};
```

5.2 Regular stencils

The motivating example above used mechanism of shifts on the output distribution. These shifts are very similar to the mechanism used to specify stencils in the Pochoir compiler [6].

5.3 Irregular sparse matrices

We implement distributed sparse matrices with a small derived class:

```
%% 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());
  };
};
```

5.4 Redundant computation

Tree-structured computations are problem case for distributed computing: at the highest levels in the tree there will be fewer nodes per level than processors. This means that we have to decide either to let processors go inactive, or to have partial or total redundance calculation.

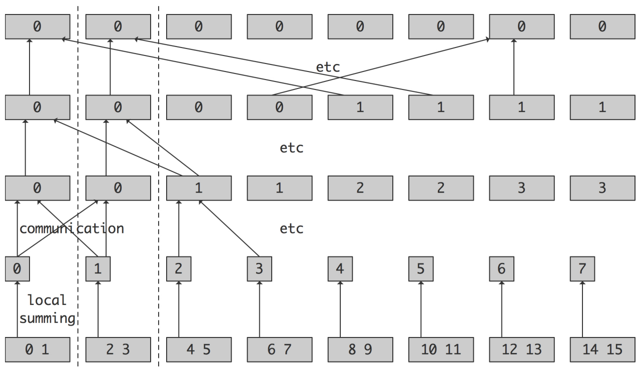


Figure 2: Distributed structure of a tree with 16 leaves on 8 processors.

There are various arguments why redundancy is the better strategy. The obvious counter argument that redundant computation entails more work is countered by the fact that (in the downward tree) there will be less communication. Furthermore, redundant computing is more 'symmetric', hence probably easier to reason about and to program; it will certainly be easier to realize in the case of non-uniform refinement on the partially populated levels.

Here we look at a gather operation towards the root of a tree, where the top levels of the tree are redundantly distributed in a manner we will make precise. For purposes of illustration it is easiest to consider figure 2.

1. The bottom two levels are disjointly distributed, and the first step of the reduction is a strictly local operation.
2. The top three levels are partially or totally redundant;
3. The reductions to the redundant tree levels are no longer local and involve communication. (If there source level is redundant, this communication is no longer uniquely determined.)

Mathematically, we derive these distributions by successive division. Let γ be the distribution of one level, and say

$$\gamma(p) = [i_0, \ldots, i_1],$$

then $\gamma' = \gamma/2$ is the distribution

$$\gamma'(p) = [i_0/2, \ldots, i_1/2].$$

This gives the desired behaviour:

$$\left.\begin{array}{r} \gamma(p) = [0] \\ \gamma(p+1) = [1] \\ \gamma(p+2) = [2] \\ \gamma(p+3) = [2] \end{array}\right\} \Rightarrow \left\{\begin{array}{r} \gamma'(p) = [0] \\ \gamma'(p+1) = [0] \\ \gamma'(p+2) = [1] \\ \gamma'(p+3) = [1] \end{array}\right.$$

The code implementing this is relatively straightforward:

```
%% unittest_struct.cxx
auto div2 = std::shared_ptr<ioperator>( new ioperator("/2") );
auto mul2 = std::shared_ptr<ioperator>( new ioperator("x2") );
distribution *distributions[nlevels];
product_object *objects[nlevels];
for (int nlevel=0; nlevel<nlevels; nlevel++) {
  if (nlevel==0) {
    distributions[0]
        = new product_block_distribution(decomp,points_per_proc,-1);
  } else {
    distributions[nlevel] = distributions[nlevel-1]->operate(div2);
  }
  INFO( "level: " << nlevel << "; g=" << distributions[nlevel]->global_size(0) );
  objects[nlevel] = new product_object(distributions[nlevel]);
```

The indirect functions of the kernel are then given by an explicit function pointer:

```
%% unittest_struct.cxx
product_kernel *kernels[nlevels-1];
for (int nlevel=0; nlevel<nlevels-1; nlevel++) {
  INFO( "level: " << nlevel );
  char name[20];
  sprintf(name,"gather-%d",nlevel);
  kernels[nlevel] = new product_kernel(objects[nlevel],objects[nlevel+1]);
  kernels[nlevel]->set_name( name );
  kernels[nlevel]->set_signature_function( &doubleinterval );
  kernels[nlevel]->set_localexecutefn( &scansum );
}
```

5.5 Load balancing

In IMP, a load redistribution can be realized through a copy operation between two different distributions on the same data. We implement this through the basic mechanism of operating on distributions, driven by an object that describes the current workload.

```
%% template_balance.cxx
auto average =
  new distribution_sigma_operator
  ( [load] (distribution *d,processor_coordinate *p) -> std::shared_ptr<multi_indexstruct> {
    return transform_by_average(d,p,load); } );
  newblock = block->operate(average,true);
```

The `transform_by_average` function is not entirely trivial, but can be written on the user level: all available functionality follows from our definition of distributions.

6 Innovation and impact

In addition to the **theoretical novelty** argument, we can make several practical arguments for our proposed system, showing that in several aspects our system creates the opportunity for a significant leap in the state of parallel scientific programming, rather than a simple incremental improvement.

For the Integrative Model we claim a **productivity** advantage over existing software/hardware systems. Secondly, by programming in a mode-independent manner we make a **code portability** and **future-proofing** argument: codes will become transferable to different and future hardware environments. Thirdly, we argue that an IMP system is capable of dealing with circumstances such as replicated data that current systems typically do not include, giving us a **capability** argument.

We further claim a **capability argument** that our system can deal with hybrid computing as easily as with single-mode (distributed or shared memory) computing. On projected future (**exascale**) architectures that can become a very important consideration [3], and we are pretty much unique in making this claim.

A hybrid system also allows us to make a **performance** argument. Since the ISP system can analyze the Intermediate Representation of the code, it can engage in task re-ordering and migration. We have shown theoretically how this can transform codes to a ‘communication avoiding’ state.

7 Conclusion

We have made a theoretical and practical case that it is possible to have an efficient, and efficiently programmable parallel system based on the following principles:

- There should be a separation between how algorithms are expressed and how they are executed. In particular, we argue for a programming model with sequential semantics and an execution model based on dataflow.
- Information about the algorithm and the execution should be explicitly expressed, rather be implicit and derived by a compiler or runtime system. We have shown that IMP offers a way to express these algorithm and data properties in a way that is not a great imposition to the programmer.
- For an efficient execution, it is necessary to express both information about the data and about the algorithm. IMP has a construct, the β-distribution, that contains this information and that is derived from the user specification of the data and algorithm. In particular, data dependencies (including messages) are not explicitly programmed, but rather derived, in IMP.
- It is possible to have a model for parallelism that is completely mathematically defined. This makes it possible to prove correctness, efficiency, and to define transformations on the algorithm. In work not reported here we have shown that IMP can accomodate, for instance, load balancing and redundant computing.

References

[1] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, pages 375–386, Piscataway, NJ, USA, 2013. IEEE Press.

[2] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[3] Peter Kogge and John Shalf. Exascale computing trends: Adjusting to the New Normal' for computer architecture. *Computing in Science and Engg.*, 15(6):16–26, November 2013.

[4] Rishiyur S. Nikhil. An overview of the parallel language id (a foundation for ph, a parallel dialect of haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1993.

[5] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[6] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.