TACC Technical Report IMP-17

The Conjugate Gradient Method in 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

We discuss the implementation of the Conjugate Gradient method in the Integrative Model for Parallelism (IMP).

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 Power method

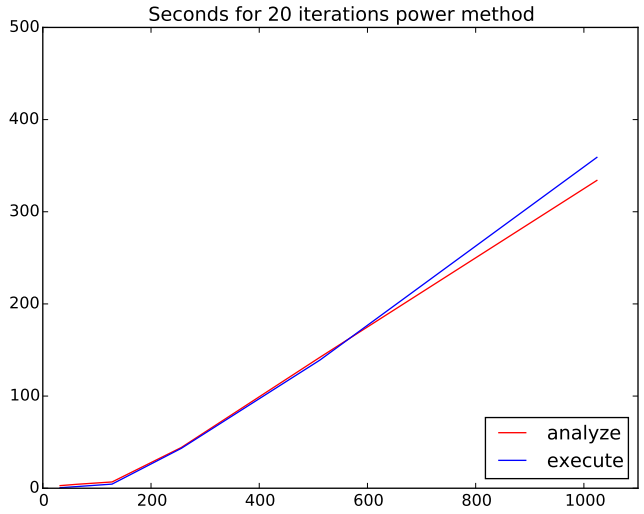
As a preliminary to the Conjugate Gradients (CG) method, we do the power method:

$$\begin{cases} x^{(0)} \equiv 1 \\ y^{(i+1)} = Ax^{(i)} \\ x^{(i+1)} = y^{(i+1)} / \|y^{(i+1)}\| \end{cases}$$

```
%% template_power.cxx
queue->add_kernel( new IMP_origin_kernel(xs[0]) );
for (int step=0; step<nsteps; step++) {
  kernel *matvec, *scaletonext,*getlambda;
  matvec = new IMP_spmvp_kernel( xs[2*step],xs[2*step+1],A );
  queue->add_kernel(matvec);
  getlambda = new IMP_innerproduct_kernel( xs[2*step],xs[2*step+1],lambdas[step] );
  queue->add_kernel(getlambda);
  if (step<nsteps-1) {
    scaletonext = new IMP_scaledown_kernel( lambdavalue+step,xs[2*step+1],xs[2*step+2] );
    queue->add_kernel(scaletonext);
  }
}
```

Runtime is proportional to the number of processors. This may not hold for much larger scale, since we implement the norm as a sequence of messages, not as a collective. See *IMP-15*.

Analyze time is approximately equal to runtime.



2 Conjugate Gradients

The loop body of a CG iteration is easily enough written in IMP:

```
%% 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));
```

2.1 Operation kernels

This uses various kernels that are all strictly written in IMP; the slight exception being the sparse matrix vector product which uses non-trivial skills; see *IMP-13*. Our ideal is exemplified by the copy kernel:

```
%% 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) {
  };
};
```

which declares itself to be a copy kernel and an MPI kernel, where the general copy kernel has all the algorithm knowledge of the copy operation:

```
%% imp_ops.h
class copy_kernel : virtual public kernel {
public:
  copy_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("copy{}",get_out_object()->get_object_number()));
    dependency *d = last_dependency();
    d->set_explicit_beta_distribution(out);
    localexecutefn = &veccopy;
  };
};
```

Other kernels are still awaiting sophisticated application of the *factory pattern* to be so generalized.

2.2 Looping

The looping nature of CG poses a problem: we do not want to allocate as many instances of an object (say, the search direction) as there are iterations. Instead we create that many objects, but let them share their address spaces. The guaranteed correctness of this relies on the execution theory of *IMP-04*, but is for now based purely on programmer insight.

2.3 Collectives

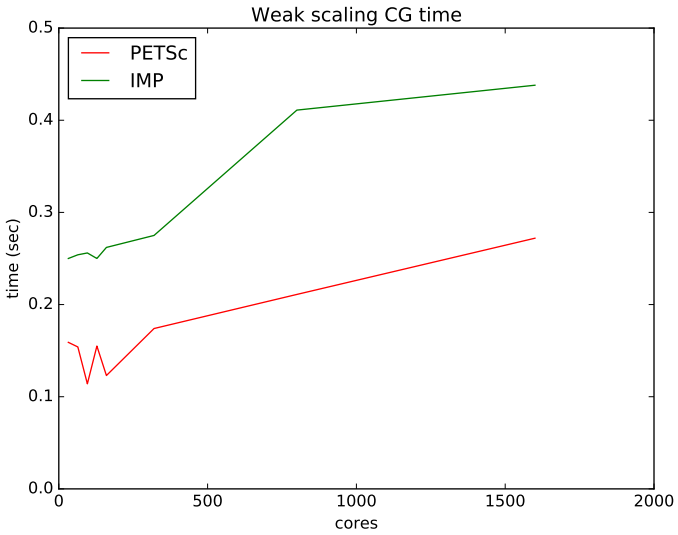
The collectives in CG, their implementation and efficiency, are discussed in *IMP-15*.

2.4 Timing results

A first timing run shows essentially the same scaling as a PETSc[2] implementation, with slightly higher runtime. This does not surprise me or even concern me, since everything about the IMP implementation is derived, without shortcuts or optimizations.

2016 November Stampede run, 16 cores per node, up to 200 nodes, 300k points per process

Size	32	64	96	128	160	320	800	1600
PETSc	0.159	0.154	0.114	0.155	0.123	0.174	0.211	0.272
IMP	0.250	0.254	0.256	0.250	0.262	0.275	0.411	0.438



3 Pipelined CG

First (incomplete) outlined in a presentation by Gropp [1], pipelined CG is based on introducing explicit recurrences for $M^{-1}r_i$ and Ap_i:

let $q_1 = Ap_1, z_1 = M^{-1}r_1$ be given

$$\begin{cases} \pi = p_1^t A p_1 \\ \alpha = \rho_1/\pi \\ r_2 = r_1 + A p_1 \alpha \\ z_2 = M^{-1} r_2 \\ \rho_2 = r_2^t z_2 \\ \beta = \rho_2/\rho_1 \\ p_2 = z_2 + p_1 \beta \end{cases} \Rightarrow \begin{cases} \pi = p_1^t q_1 \\ \alpha = \rho_1/\pi \\ \begin{cases} r_2 = r_1 + q_1 \alpha \\ z_2 = z_1 + M^{-1} q_1 \end{cases} \\ \rho_2 = r_2^t z_2 \\ \beta = \rho_2/\rho_1 \\ \begin{cases} p_2 = z_2 + p_1 \beta \\ q_2 = A z_2 + q_1 \beta \end{cases} \end{cases}$$

We now observe that the added relations contain terms $M^{-1}q_1$ and Az_2 that can be overlapped with inner product calculations:

$$\begin{cases} \text{overlapped:} \\ \text{compute } M^{-1} q_1 \\ \pi = p_1^t q_1 \end{cases}$$
$$\alpha = \rho_1/\pi$$
$$\begin{cases} \text{in arbitrary order:} \\ r_2 = r_1 + A p_1 \alpha \\ z_2 = z_1 + M^{-1} q_1 \alpha \end{cases}$$
$$\begin{cases} \text{overlapped:} \\ \text{compute } A z_2 \\ \rho_2 = r_2^t z_2 \end{cases}$$
$$\beta = \rho_2/\rho_1$$
$$\begin{cases} \text{in arbitrary order:} \\ p_2 = z_2 + p_1 \beta \\ q_2 = A z_2 + q_1 \beta \end{cases}$$

Figure 1 illustrates the independence of the Az_2 and $r_2^t z_2$ products: they have common inputs, common outputs, but no causal relationship. In the IMP model, all communications are initiated as part of the kernel that produces z_2, then overlapped with local computations.

References

[1] Bill Gropp. Update on libraries. http://jointlab-pc.ncsa.illinois.edu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf.

[2] W. D. Gropp and B. F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference, IEEE 1994*, pages 87–93.

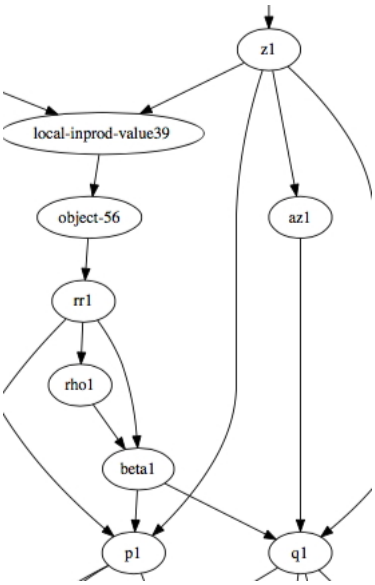


Figure 1: Matrix-vector and vector-vector inner product in the kernel structure of pipelined CG