TACC Technical Report IMP-19

Progress report on the Integrative Model under NSF EAGER grant 1451204

Victor Eijkhout*

May 9, 2017

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

Abstract

The Integrative Model for Parallelism (IMP) is a parallel programming system, primarily aimed at scientific applications.

Its aim is to offer an integrated model for programming many different modes of parallelism. In fact, in the IMP system one source file can be compiled into message passing, Directed Acyclic Graph-based, and hybrid execution models. Also, performance enhancing techniques such as redundant computation and latency hiding are automatically achieved without need for programmer intervention.

From September 2014 through August 2015 the IMP project was supported by NSF Eager grant 1451204. We report on progress made during that time.

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 The Integrative Model for Parallelism

The Integrative Model for Parallelism (IMP) is a parallel programming system based on a new theory of parallel computing, primarily aimed at scientific applications.

By offering a mathematical description of distributed objects and calculations, IMP arrives at a description of parallel algorithms that is independent of the actual type of parallelism. Thus, we can compile one source file into MPI, Directed Acyclic Graph (DAG)-based (such as OpenMP tasking), and hybrid execution models. The theory is powerful enough to formalize techniques such as redundant computation and latency hiding. The IMP sytem can apply such optimizations achieved without need for programmer intervention.

Proof-of-concept tests show a performance similar to code written by hand or realized in popular libraries such as PETSc.

1.1 IMP API basics

To the user, the IMP software offers an interface that can be described as 'sequential programming with distributed objects'. The basic sequential step is called a 'kernel', and it corresponds to a generalized notion of a data parallel operation: an operation that takes one or more distributed objects as input, and produces a distributed object as output, where each element of the output can be computed independently.

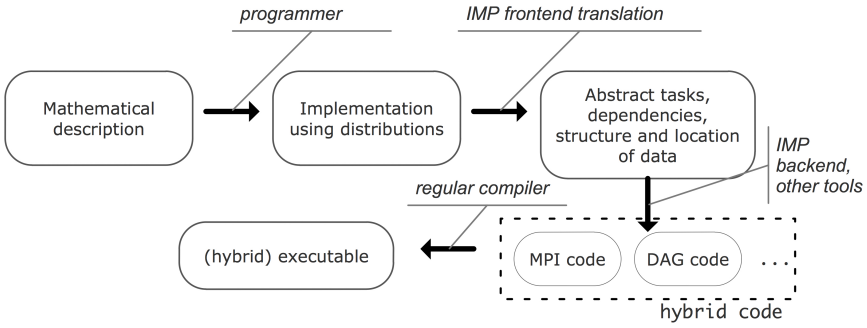


Figure 1: Structure of the IMP system

The IMP system then translates this user description of an algorithm to an Intermediate Representation (IR) that is independent of the underlying mode of parallelism. A second translation stage then realizes this as a parallel program in a particular system, for instance using MPI or OpenMP calls[1] or hybrid combinations.

1. We use the tasking mechanism of OpenMP, not the more common loop parallelization functionality.

1.2 Inspector-executor

To the user an IMP program looks like a sequential program of C++ library calls. However, instead of being executed, these library calls actually insert a kernel (see above) into a algorithm data structure. This algorithm is then separately analyzed and executed, in a clear case of the inspector-executor paradigm [3]. A first result of the algorithm analysis is a kernel dataflow formulation based on a dependency analysis of the input and output objects. A task dataflow graph as it is commonly considered is then derived by considering the execution of a kernel on a processing element as a task.

1.3 Dataflow

The previous item mentioned two forms of dataflow. An example will elucidate this distinction.

The heat equation is rendered by a single sequence of kernels; see figure 2, left. This is the dataflow that the user programs. On the other hand, the task dataflow that governs interaction between processors is derived by the IMP system; see figure 2 right.

Thus our notion of dataflow programming is much simpler than dataflow programming as it is commonly proposed. In essence, the programmer codes the algorithm, not its execution.

1.4 Communication and synchronization

Based on the kernel dataflow graph and distribution of objects specified by the programmer, the IMP system derives a (much larger) task dataflow graph. The task dependencies can be satisfied as MPI messages, or as OpenMP explicit task directives.

We note that we do not concern ourselves with the task-local code, which is provided through a function pointer, much as in other DAG execution models such as Quark [4]. This may be a question for a compiler project in the future; see section 3.2.

2 Work performed

In the past year, we have solidified the IMP infrastructure, written the MPI and OpenMP backends to where they can implement some meaningful codes, and have started on the MPI+OpenMP hybrid backend.

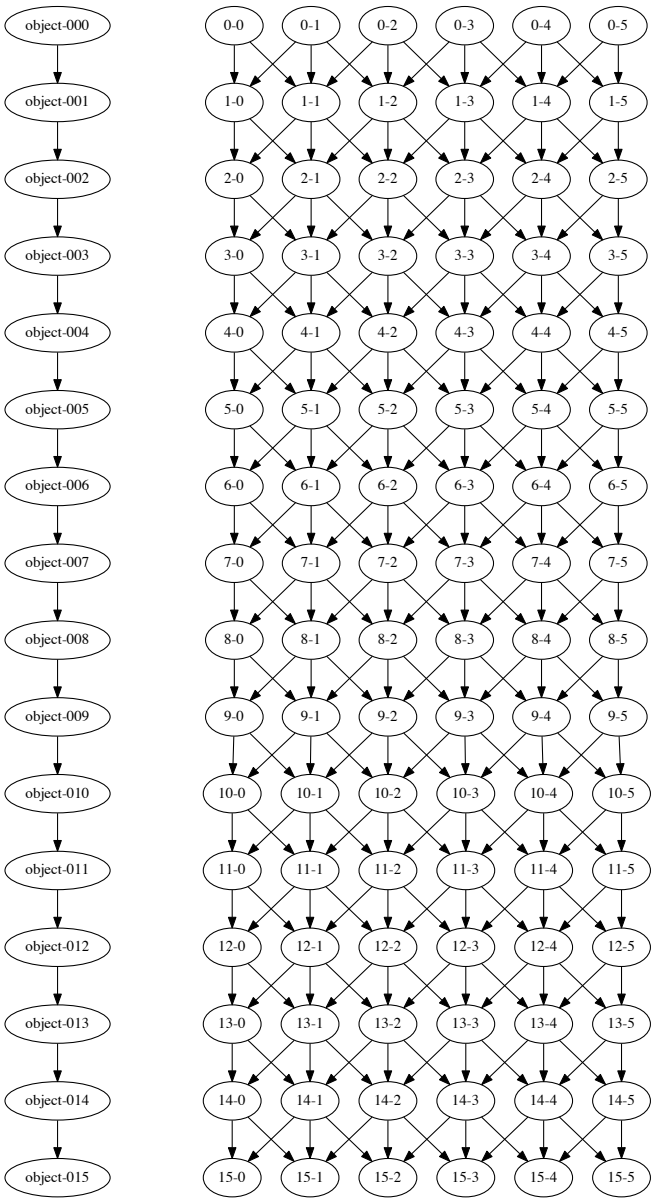


Figure 2: Kernel dataflow (left) and task dataflow (right) for the one-dimensional heat equation, executing 15 steps on 6 processors.

2.1 Infrastructure; MPI and OpenMP backends

We have proved the integrative nature of our model by realizing both a message passing (MPI) and DAG-based (OpenMP) backend to our software. This required building a substantial amount of infrastructure. Interesting is the generalization of the communication and synchronization mechanisms and the algorithm execution.

Also we implemented a very crude sparse matrix class. Under the assumption that for many sparse matrices bandwidth is the performance determining factor, we did not optimize this at all.

As an indication of our conceptual integration in code, we mention that the base code is around 4k lines, with 1.5k lines for the derived classes of MPI and OpenMP each, and 0.7k lines for MPI/OpenMP hybrid. (We also note that our testing code is about 16k lines, double that of all the actual library code.) About half of this codebase is written in the past year.

2.2 Hybrid computing

In probably our most intriguing development, we have combined the MPI and OpenMP backends, and combined them into a hybrid MPI+DAG implementation. (As noted above, thanks to our fargoing code reuse design, this part of the library is actually very small.) In our current design, we take MPI communication and synchronization mechanisms, but replace the task-local code by an OpenMP DAG of tasks.

This serves several purposes. For instance, on very large processor counts it cuts down on the $O(P)$ space that is typically allocated on each process. In case of irregular workloads it can also achieve a statistical load balancing by making load-to-processor assignment flexible.

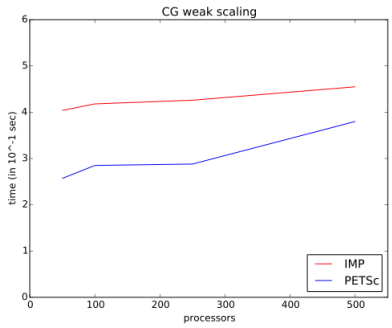
However, the main promise of hybrid computing lies in the future; see section 3.1.

2.3 Conjugate gradients

Using a crude sparse matrix class we implemented a conjugate gradients method, and compared its performance to a PETSc code. We see that our code is slower than PETSc, but by less than a factor of 2. More importantly, we note that our scaling behaviour looks similar to PETSc[2].

2. At the moment, scaling to large numbers of processors runs into a design mistake: we wanted each messsage to be identifiable by a unique tag. Unfortunately MPI tag space is not large enough for this. This will be fixed fairly quickly.

Our CG code (see figure 3) looks in fact much like the PETSc implementation, with one library call per basic linear algebra operation (inner product, matrix-vector product, et cetera). Two differences: first of all, these linear algebra operations are themselves implemented in IMP, as opposed to the PETSc operations which can be low level code. Secondly, IMP code looks slightly more intricate, since scalar operations are operations on redundantly distributed objects, and therefore have to be realized with library calls.



2.4 N-body problems

We have implemented the algorithm structure of a tree code such as for an N-body problem; see [2]. Figure 4 shows the resulting kernel dataflow. (See section 1.3 for a discussion of dataflow in the IMP model.)

The point of interest in this application is the treatment of the higher tree levels, where there are fewer points than processors. Since IMP takes a very symmetric look at processes, it uses redundant replication of points on these levels. The decision to use replication can be defended from a point of energy efficiency. Under this strategy the prolongation down the tree is fully local, whereas a non-replicated strategy would involve a good deal of message traffic.

The resulting algorithm is illustrated, for 8 leave nodes and 4 processors, in figure 5.

The main point worth noting here is that this takes no effort at all to program: the user never specifies the redundancy, only the halving of the points per level. In the following code snippet the distribution `level_dist` on the finest level is given; all other distributions, including the redundantly replicated ones, are derived here:

```
%% template_nbody.cxx
auto coarsen = new multi_sigma_operator
  ( new sigma_operator( std::shared_ptr<ioperator>( new ioperator("/2") ) ) );
auto distributions = new std::vector<distribution*>;
distributions->push_back(level_dist);
index_int g = level_dist->global_volume();
for (int level=0; ; level++) {
  distributions->push_back(new_dist);
  g /= 2;
  new_dist->set_name(fmt::format("level-{}-distribution-of-{}-pts",level,g));
```

```
%% 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));
```

Figure 3: Loop body of the IMP CG code

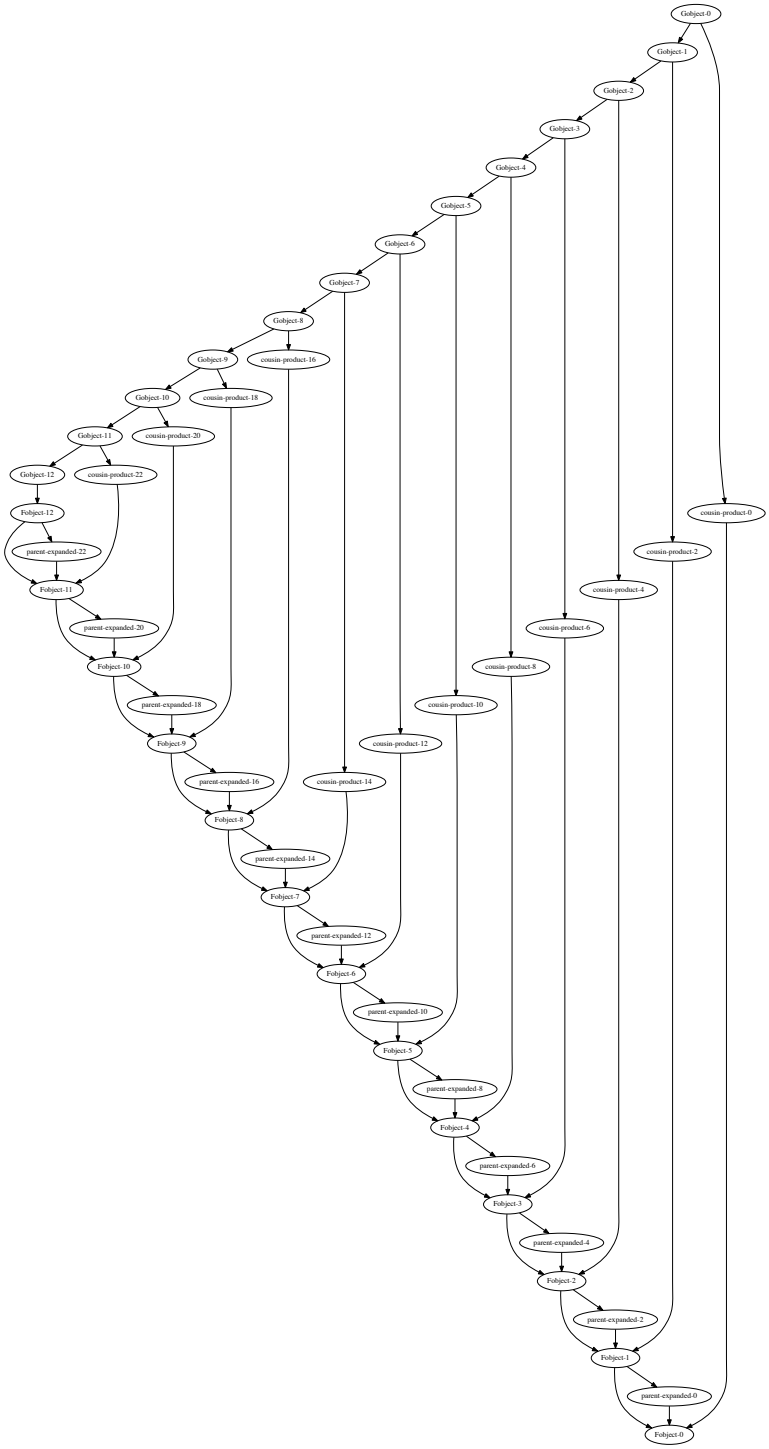


Figure 4: The kernel structure of an N-body code

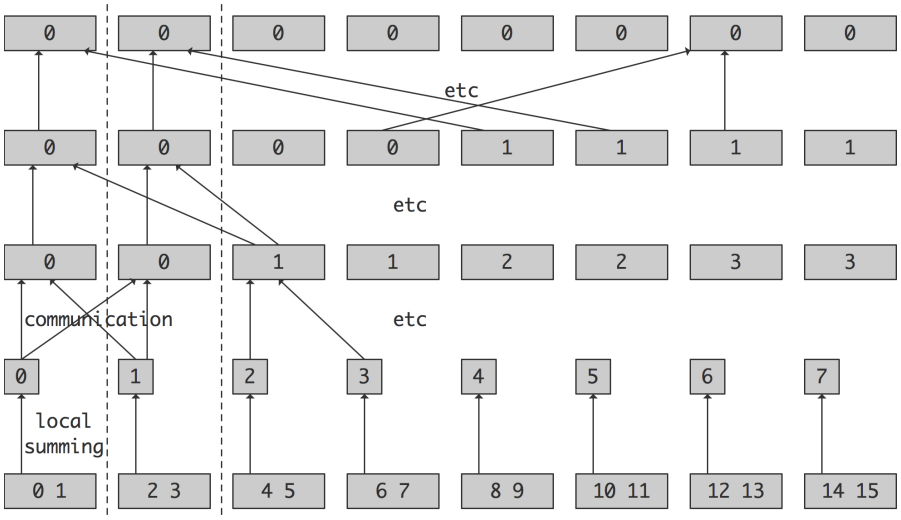


Figure 5: Computation and communication up the tree

```
  if (g==1) break;
  level_dist = new_dist;
}
```

2.5 Latency optimization

MPI non-blocking calls hold the promise of overlapping computation and communication. To this end, the IMP system moves posting the Isend and Irecv calls as far down the task graph as possible. This also implies that non-blocking collectives can be employed in the IMP system without the user ever having to specify them.

On our current test systems, this mechanism has not paid off by more than a modest 10–15 percent. We will seek out clusters that offer hardware support for communication offload.

3 Future work

3.1 Latency tolerance and communication avoidance

Probably the most exciting future prospect of our hybrid scheme of section 2.2 is manipulations on the global task graph. If each MPI-task contains a non-trivial OpenMP task DAG, we can migrate and duplicate tasks between MPI processes in such a way as to effect a 'communication avoiding' scheme [1].

In *IMP-06* [**?**] we formally derived the task subsets that will completely hide latency, while minimizing communication and redundant computation.

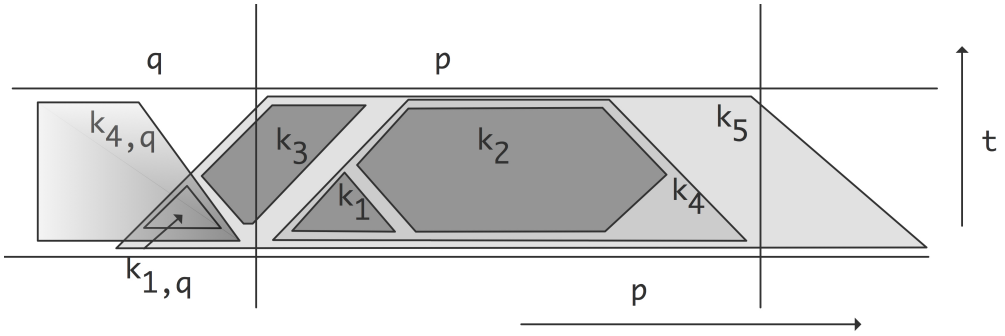


Figure 6: Formal derivation of a communication avoiding partitioning of a task graph

3.2 Local code generation

In the current IMP design, the task-local code is provided by a function pointer. The local code is different between MPI and OpenMP, and perhaps even between standalone OpenMP and OpenMP in hybrid context. Generation of this code from a more abstract description of the algorithm is an open question.

While the mere generation of this code is only a moderately interesting project, various other questions have research potential:

- Can code be generated with annotations on where it synchronizes with other tasks, and where it is independent? Can the synchronization be quantified? As argued elsewhere, this may lead to programmable prefetch streams, which can obviate caches and coherence.
- If we go to very fine-grained tasks, can a compiler generate code that optimally uses registers and last-level caches?
- Can code generation interact with an optimization mechanism, so that granularity is adaptively chosen?

3.3 The Eijkhout-McCalpin garbage preventor

The above description of kernel dataflow being based on input and output objects implicitly carries the assumption that an object is only ever created once, reminiscent of a functional programming style. However, the dynamic memory management and attendant garbage collection are unacceptable in a scientific computing context. Therefore we can indicate that one object reuses the memory of another.

The typically application for this mechanism is an iterative loop, where each iteration produces a new object, but based on the memory of the object of the previous iteration. In many cases, the user can make the decision to reuse; however, we have a theoretical framework under which the system can make such a decision; see *IMP-04* [**?**].

3.4 Distributions

Our distribution mechanism is theoretically clear. Practical implementation is a matter of time. The following two directions are considered for future work.

Implementing dense linear algebra would be an interesting test case for the IMP model. For this we first of all need to develop two-dimensional distributions.

All of our test applications at the moment have a very symmetrical treatment of the processes. Locally adaptive grid refinement can be formulated in the IMP theory by using partially defined distributions. The theory for this, and some mathematically described examples, are found in *IMP-01* [**?**]. The implementation is another story.

3.5 Redundant work and other resiliency topics

IMP distributions easily accomodate redundant computation as we have shown in simple examples. However, we have not yet implemented redundant duplication of whole tasks. While the model supports this, the implementation needs much research.

Communication avoidance (see 3.1 above) is one application of this mechanism; another would be adding resilience to the model. Rather than pure duplication, we could also consider the on-demand recomputation of failed tasks.

4 Conclusion

We have shown in a proof-of-concept code that it is possible to code scientific applications in an essentially sequential manner, with the translation to multiple parallel backends happening automatically. The translating stage is smart enough to reproduce the performance of traditional codes; non-trivial code transformations are a distinct future possibility.

Acknowledgement

Acknowledgement

This work was supported by NSF EAGER grant 1451204. The code for this project is available at https://bitbucket.org/VictorEijkhout/imp-code.

References

[1] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *IEEE International Parallel and Distributed Processing Symposium*, 2008.

[2] John K. Salmon, Michael S. Warren, and Gregoire S. Winckelmans. Fast parallel tree codes for gravitational and fluid dynamical n-body problems. *Int. J. Supercomputer Appl*, 8:129–142, 1986.

[3] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(1-4):73–86, 1992.

[4] A. Yarkhan, J. Kurzak, and J. Dongarra. QUARK users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee Innovative Computing Laboratory, 2011.