TACC Technical Report TR-14-01

A DSL for Integrative Parallel Programming

Victor Eijkhout*

September 1, 2014

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

Abstract

In this report we show a preliminary software realization of the ideas of the recently proposed Integrative Model for Parallelism (IMP). This model allows for expression of parallel operations through sequential semantics, giving a mode of expression that is easier for the programmer than other existing systems. The theoretical basis of IMP ensures that its concepts can be realized in multiple parallelism modalities, such as task scheduling, message passing, accelerator offloading, or hybrid combinations of these. We will illustrate this point by showing that a single main program can be realized both as MPI and OpenMP task execution. In either case, efficiency is comparable to a hand-coded solution in these systems.

1 Introduction

Many models of parallelism assume processing elements (processors, tasks, threads) that perform independent tasks. However, in practice many computations are of a general type of data parallelism where these 'subtasks' are part of a parallel 'supertask'. For instance, in Finite Element calculations billions of grid points undergo largely identical fine-grained operations as part of a single grid update; in large scale dense linear algebra subblocks are factored or multiplied as part of the factorization or multiplication of a large distributed matrix. Current parallel programming models are able to express the asynchronous execution of the subtasks; they do not express the conceptual coherence of the supertask. We aim to improve this situation by having a language or library that allows for the macroscopic 'supertask' expression of an algorithm, and where the task activity of the individual processing elements, including their asynchronous execution, follows as a consequence, rather than being explicitly programmed.

Various attempts have been made of the last few decades to design programming systems that allow such single statement descriptions (we call this 'sequential semantics') of parallel activities. The conceptual attraction of such an approach is eloquently formulated in [12]:

> [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[.]

However, with the exception of low-level systems (vector instructions) and specialized applications (linear algebra libraries such as Scalapack [4] and Elemental [13]), no such coding systems with sequential semantics exist.

On the other hand, there is considerable theory and practice of parallelism based on a model that considers the execution on individual processing elements as a task by itself, rather than as part of a macro task, and then describes the interaction. The reality of computer architecture argues strongly for the merits of this second approach, since it explicitly accounts for the locality of data to processing elements. In current computer architectures the cost of data motion is not to be neglected, so such a system that explicitly handles locality is a guarantee for high performance. (Prime evidence here is of course the MPI library, which is the only way to get million-way parallelism with high efficiency.)

In this paper we introduce language concepts and a DSL based on the Integrative Model for Parallelism (IMP) [6], a theoretical system for describing, analyzing, and implementing general data parallel operations on distributed data. We argue that the IMP model is theoretically strong enough to derive relevant concepts such as overlapping computation and communication, and optimal granularity of synchronization. By means of a software prototype we show that the system is interpretable in multiple parallelism types with efficiency comparable to hand-coded software.

1.1 Motivating example

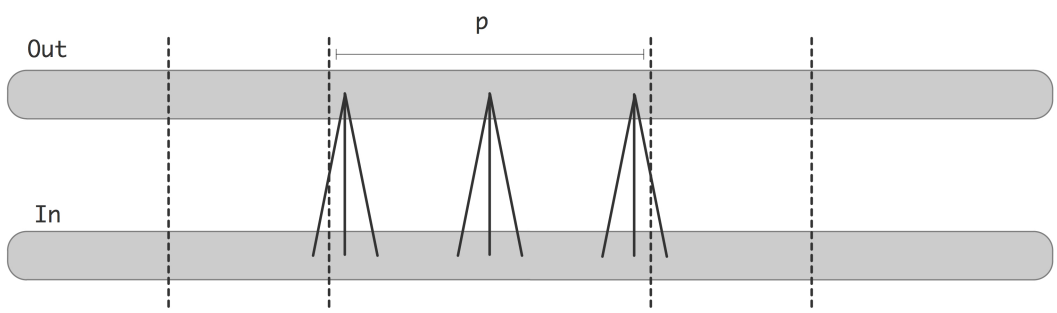
We consider a simple data parallel example, and show how it leads to the basic concepts of IMP: the three-point operation

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

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., [17] and the polydral model, e.g., [5]. 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, and it corresponds to the traditional concept of distributions in parallel programming systems:

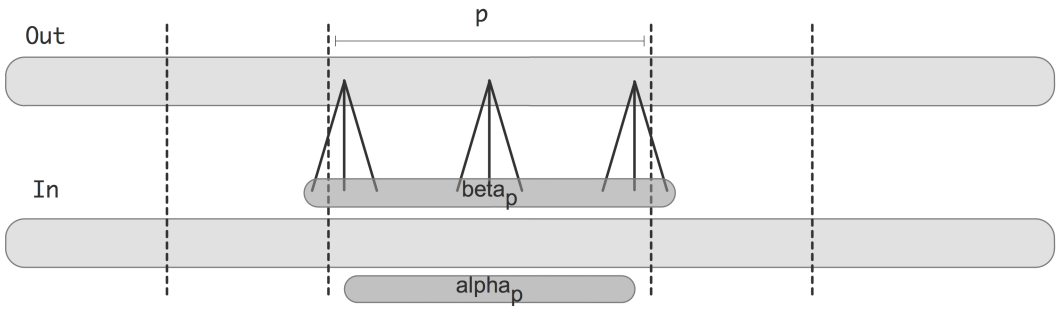
$$\alpha\colon p \mapsto [i_{p,\min}, \ldots, i_{p,\max}].$$

The picture shows how 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 $p-1$.

Now, we can in fact identify a second distribution that differs from the traditional notion of distribution. Traditionally, a distribution in the context of a vector is defined as a mapping from the vector indices to processors, reflecting that each index 'lives on' one processor, or that that processor is responsible for computing that index of the output.

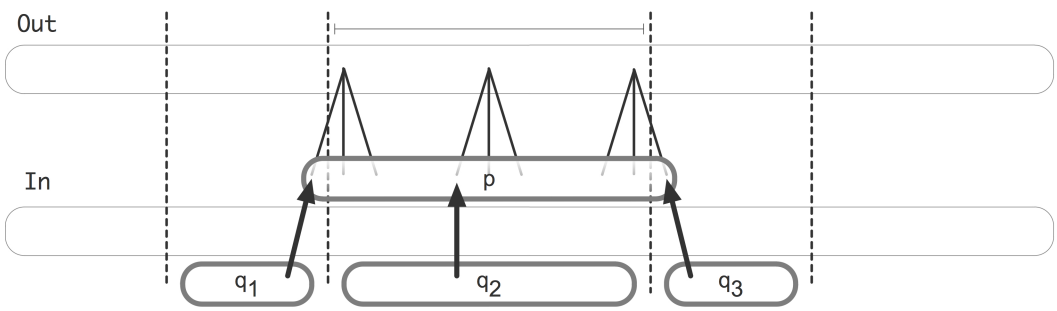
We turn this upside down: we define a distribution as a mapping from processors to indices. This means that an index can 'belong' to more than one processor. This concept naturally arises in the context of the three-point operation: for each processor we have not only the set of indices that it computes, but also the set of all indices it needs as inputs. This we call the β-distribution: $\beta(p)$ is the set of indices that processor p needs to compute the indices in $\alpha(p)$.

The second illustration depicts these two distributions for one particular process:



Observe that the β-distribution, unlike the α one, is not disjoint: certain elements are assigned to more than one processing element.

This gives us all the ingredients for reasoning about parallelism. Defining a kernel as a mapping from one distributed data set to another, all data dependence 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, and this set of predecessors can be derived from the α, β distributions.



In message passing, these 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.)

Interestingly, this story has an interpretation in tasks on shared memory too. If we identify the α-distribution on the input with tasks that produce this input, then the β-distribution describes what input-producing tasks a task p is dependent on. In this case, the transformation from α to β-distribution gives rise to a *dataflow* formulation of the algorithm.

This example and discussion show that the IMP model can make some claims to expressiveness in dealing with algorithms in multiple types of parallelism. We also need to argue that this model can be programmed. For this we need to define some notations, which we will do below in detail. In

this notational framework, if x is a distributed object, and d its α-distribution, by $x(d)$ we will mean 'x distributed with d'. Furthermore, we will introduce transformations on distributions, so that for instance $d \gg 1$ means 'd right-shifted by 1'. With this, we can write the three-point kernel as

$$y(d) \leftarrow f\big(x(d), x(d \ll 1), x(d \gg 1)\big). \tag{1}$$

This states that vector y with distribution d can be computed from x, if x is given on the distributions $d, d \gg 1, d \ll 1$. The resulting dataflow can then be derived by a compiler-like component.

1.2 Philosophy of IMP

With the above motivating discussion, we arrive at the structure of the IMP system: the mathematical definition of an algorithm is expressed in terms of kernels and distribution, independent of any target parallelism. We can do this through a programming language, a library, or any solution in between. The IMP compiler translates this global expression to a dataflow formulation where the work is broken up over processing elements, still independent of target hardware and software. The IMP backend software then translates this to specific calls in the target parallel system. This is

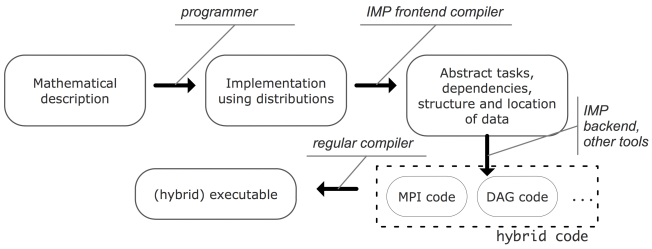


Figure 1: Structure of the IMP system

summarized in figure 1.

Above we briefly motivated IMP, both from an example and from a general discussion on the desirability of sequential semantics. We will now give a brief discussion how IMP relates to other systems.

1.2.1 Connections to existing notions

IMP is rooted in some well-known concepts, but gives a new perspective on them. To forestall obvious criticism we briefly discuss our relation to them, noting that none of this is essential.

Our basic **programming** concept is that of kernels: the expression of a mathematical operation on parallel data, expressed with sequential semantics. Above we already argued the attraction of this approach. There is also a distinct similarity in expression, though not in execution, between IMP kernels and Bulk Synchronous Parallelism (BSP) supersteps [16, 18].

Data in IMP is defined with respect to a *distribution*. The concept of distribution appeared for instance in Universal Parallel C (UPC) [2]; in High Performance Fortran (HPF) [9, 8] it allowed the user to program manipulation of parallel objects in code with otherwise sequential semantics.

Our execution concept is that of **dataflow**, which has long been studied [1] and has recently seen a revival in Directed Acyclic Graph (DAG) based schedulers [3, 14]. While acknowledging the universality of the concept, we argue that it should not be the programming model.

1.2.2 What is new in IMP

The strength of IMP, and its essential innovation, lies in the fact that not only is data distribution explicitly specified, but the same holds for the distribution of execution. This means that an IMP translator can (when runtime parameters are known) analyze an algorithm in terms of the data distributions involved, and derive all necessary and sufficient data motion / data dependencies.

The precise form in which execution distribution is specified is still under discussion and development, but a first indication can be seen in equation (1) above. Here, instead of specifying the operation in terms of indices in some flat space, the operation is specified in terms of the data distributions of the input and output. All aspects of data motion and data dependencies can then be derived by formal derivation.

Another way of phrasing the essence of IMP is to note that parallel execution is a convolving of actions and data layout. In IMP the actions are explicitly phrased in terms of the layout, giving the IMP translator (which operates both at compile and runtime) unprecedented power to arrive at efficient execution.

2 Software realization

To argue that our ideas are efficiently implementable, we present a prototype code along the following lines:

- There are IMP base classes, implementing the basic structure of the model;
- There are two sets of derived classes, one for MPI and one for OpenMP, and both offering the same API;
- There is a single main program expressed in the API of the previous point; if this program is linked to the MPI classes it becomes an MPI program, if it is linked to the OpenMP classes it becomes an OpenMP program. In both cases, the resulting program performs comparably to hand-coded implementations of the same algorithm.

2.1 Threepoint averaging example

For simplicity of exposition, we use the threepoint kernel of the introduction, repeated a number of times, in essence the one-dimensional heat equation. We stress that this code is a prototype. Any

functionality implemented is there to serve the demonstration; functionality not present does not reflect on capabilities of the IMP model as such, only on the time available to the implementer.

The program starts by creating an environment object that keeps track of commandline arguments, as well as the parallel environment:

```
threepoint_environment *problem_environment =
    new threepoint_environment(argc,argv);
```

Next we create receptors for data and work objects

```
IMP_object **all_objects = new IMP_object*[nsteps+1];
IMP_task_queue* queue =
      new IMP_task_queue(problem_environment);
```

Next we declare distributions. First we a declare a distribution with which objects are created:

```
IMP_disjoint_blockdistribution *blocked =
      new IMP_disjoint_blockdistribution
          (problem_environment,globalsize);
for (int step=0; step<=nsteps; ++step) {
  IMP_object
    *output_vector = new IMP_object( blocked );
  all_objects[step] = output_vector;
}
```

Next we create a kernel for each step. The shifted distributions are used to create the inputs for the kernel.

```
for (int step=0; step<=nsteps; ++step) {
  IMP_object
    *input_vector = all_objects[step-1],
    *input_vector = all_objects[step];
  IMP_kernel *update_step =
    new IMP_kernel(input_vector,output_vector);
  update_step->localexecutefn = &threepoint_execute;
  update_step->add_beta_oper( new ioperator(">>1") );
  update_step->add_beta_oper( new ioperator("<<1") );
  update_step->add_beta_oper( new ioperator("none") );
  queue->add_kernel( step,update_step );
```

The next two instructions contain the MPI/OpenMP specific parts. Analyzing the dependencies boils down to:

- In MPI constructing the pattern of sends and receives;
- In OpenMP condensing task dependencies into a task graph.

```
queue->analyze_dependencies();
```

After that, the execute step

- in MPI, performs the actual sends and receives;
- in OpenMP, schedules the tasks with the proper dependencies;

In both cases, a task execution involves the execution of the local compute function.

```
queue->execute();
```

This execute step is unavoidable in threading interpretations, where it activates the scheduler. However, in an MPI interpretation we can replace it by explicit kernel activations:

```
all_kernels[0]->execute();
// execute local function 0 explicitly
all_kernels[1]->execute();
// execute local function 1 explicitly
et cetera
```

or in fact by cycling over a single kernel, with different in and outputs.

2.1.1 Comparison to reference code

OpenMP: We compared the threepoint averaging kernel on 3×10^7 elements, running on one node of the Stampede supercomputer at the Texas Advanced Computing Center (http://www.tacc.utexas.edu/stampede). The IMP implementation and the OpenMP reference both use the OpenMP v3 task mechanism. Figure 2 then shows that the overhead of the IMP mechanisms is negligible. Both tasking codes do not show approximately linear scaling, probably because of the

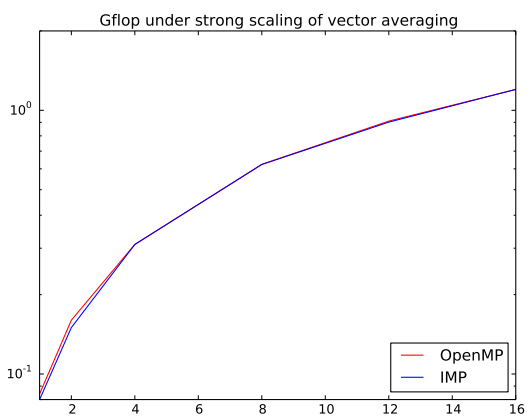


Figure 2: Strong scaling of IMP/OMP code versus OpenMP reference code as a function of core count

excess of available bandwidth at low core/thread counts.

MPI: We wrote an MPI code that performs the left/right sends hardwired. Since the IMP code is general in its treatment of communication patterns, it includes a preprocessing stage that is absent from the 'reference' code. We did not include it in the timings since

1. Preprocessing will in practice be amortized, and
2. For irregular problems an MPI code will have to perform a very similar analysis, so the choice of our test problem made the reference code unrealistically simple.

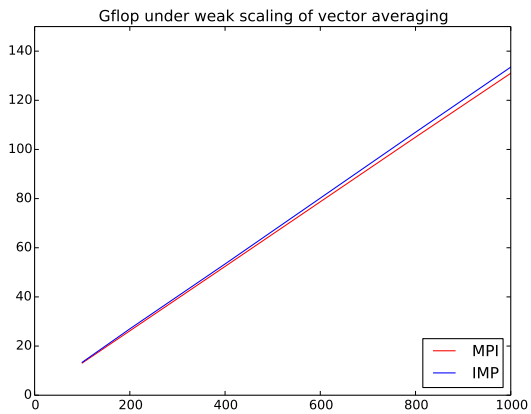


Figure 3: Weak scaling of IMP/MPI code versus MPI reference code as a function of core count

Figure 3 shows the behaviour. Since the two codes do essentially the same thing it is not surprising to see the same perfect linear scaling from both. (It is not clear why the IMP code is in fact 2–3% faster.)

2.1.2 Discussion of the prototype

We address a few possible criticisms of the prototype.

Limitations This example uses a 1D distribution, but 2D and sparse distributions are possible.

Regarding the originality of distributions The concept of distribution is not new. However, the innovation here lies in the fact that distributions are also used as a tool for describing the algorithm. Thus, the IMP translator can relate data storage and data requirements, and thus come up with an efficient implementation.

Synchronous vs asynchronous The IMP kernels seem to imply a notion of BSP-like supersteps. This is not the case: the kernels translate to a dataflow Intermediate Representation (IR) which can be executed fully asynchronously.

Function pointer callbacks The above implementation of the 1D example uses callbacks. These would not be necessary in an MPI realization, but they *are* necessary for a task-based interpretation. For MPI, the system allows you to execute a kernel directly.

2.2 Software reference

Noting that the whole system is stil under development, here are some highights of the currently implemented part.

2.2.1 `IMP_environment`

Objects in this class keep track of the execution environment. Methods:

`nprocs` The number of MPI processes or OpenMP threads.

2.2.2 `IMP_distribution`

Distribution objects describe the distribution of data over processes or threads. In the OpenMP case this does not imply any sort of affinity since the model deals in tasks, not in actual threads.

The definition of a distribution uses two auxiliary classes

`indexstruct` which describes a set of indices intended for a single processing element, and
`parallel_indexstruct` which describes the index structures of a set of processing elements.

The distribution class definition has a 'type' parameter, with values such as

`disjoint_blockdistribution` A distribution of a linear index by disjoint contiguous blocks.
`replicated_scalar` A distribution that puts the same data on each processing elements; this is typically the result of an allgather type operation.

However, these are mostly shorthand declarations; as a result of operations on distributions, any possible distribution can arise. Operations include:

`operate( ioperator* )` Apply an operator, such as shift, to a distribution.
`local_size( int p )` Query the number of elements assigned to a processing element.
`distr_union( distribution *d )` Merge two distributions, yielding on each processor the union of the local index sets.

2.2.3 `IMP_object`

Objects have a storage component, with an interpretation based on the distribution used in the create call `IMP_object( distribution* )`.

Various query routines, such as for number of processing elements or local size, are inherited from the underlying distribution.

2.2.4 `IMP_kernel`

A kernel object corresponds to the quasi data parallel execution of an operation, taking a (distributed) object as input and yielding another (distributed) object. Internally, a kernel is split into tasks which correspond to executing the kernel operation on a single processing element, yielding the part of the output object belonging to that processing element.

`IMP_task` A task object contains a list of messages that need to be performed before the task can execute. The interpretation of performing a message differs between the MPI and OpenMP realizations of an IMP program: in the former case an IMP message corresponds to actual send and receive calls, whereas in the latter it is realized as a task dependence.

`IMP_message` A message object describes the sender/receiver pair, and the `indexstruct` on both sender and receiver. This class is only used internally.

Kernel methods include:

`split_to_tasks()` This yields a set (or rather an `stl::vector`) of tasks, to be inserted in a task queue.

`add_beta_operation( ioperator* )` This routine is used to build up the description of the beta distribution for a kernel. See the threepoint kernel example above.

2.2.5 `task_queue`

Methods:

`add_kernel( kernel* )` Add a kernel to the task queue; the kernel will be split into tasks as described above.

`analyze_dependencies()` After all kernels have been declared, this call establishes all messages between the tasks.

`execute()` This call triggers the execution of all tasks in the queue. In the MPI realization it will perform the necessary send and receive calls to satisfy data dependencies; in the OpenMP realization task dependencies are satisfied more directly.

3 Discussion and conclusion

In this paper we have presented a Domain-Specific Language (DSL) based on Integrative Model for Parallelism (IMP). After motivation with a small, but non-trivial, parallel example, we showed a concrete syntax for realizing this example in our DSL. Tests bear out that IMP code is comparable in efficiency to hand-coded MPI and OpenMP code.

3.1 Relation to other research

Here we briefly discuss how the IMP concepts fit in with earlier work and how they improve on it.

Data parallelism and sequential semantics IMP expresses data parallel calculations using 'sequential semantics': the idea that a program conceptually behaves as a single instruction stream operating on potentially distributed objects. This is related to Single Program Multiple Data (SPMD) execution, but it is a stronger notion: the code only refers to objects, and their partitioning, but never to any specically assigned segment of the object.

The idea of sequential semantics has been used as a basis for several programming languages, most notably HPF [9, 8]. However, there the compiler could not adequately derive locality and movement of data, resulting in inefficient code.

We have solved this problem by designing a language where distributions are used to express both the definition of the data and the kernels. In effect, this allows the compiler to 'understand' the algorithm.

Distributions The IMP model uses distributions as an essential tool. The notion of distribution has been used many times before, for instance in HPF (discussed above) and UPC [2]. However, distributions with which the programmer can distribute data are not enough for performance.

The IMP model brings two innovations. Already mentioned is the use of distributions to describe the kernels, rather than just the data. Secondly, IMP distributions are more general than earlier attempts: we consider them as mapping from processors to data, rather than reverse. This makes possible such applications as redundant computing.

Dataflow and asynchronous execution Algorithm descriptions in IMP, despite their synchronous seeming design, lead to an intermediate representation in terms of dataflow. This solves both the problem of asynchronous execution, as well as small message aggregation. While dataflow as such is an old concept [7, 11, 10, 15], and recently in resurgence in the form of DAG models for linear algebra [19], an essential difference is that our dataflow is formally derived from the kernel and its distributions, not explicitly programmed. Thus we get its theoretical advantages without its practical difficulties.

3.2 Research directions

We briefly touch on research directions in IMP.

Heterogeneity Our proof of concept implementation was realized in pure OpenMP tasks or pure MPI. However, hybrid setups are conceivable. With these two particular systems an orthogonal setup is possible with an OpenMP distribution on each process of an MPI distribution. In other situations, such as with accelerators, a mixed setup would be needed. We note that IMP is attractive for offload models of accelerator computing since asynchronous transfer can be initiated as early as algorithmically possible.

A language for parallel programming The HPC community has largely been resistant to new languages for parallel computing, relying mostly on library approaches. Apart from the modest success of UPC, X10, Chapel, probably the largest market penetration was achieved by the OpenMP and Co-Array Fortran language extensions. Initially, therefore, we solidly aim for a library approach to IMP implementation. However, we note that the library calls are the realization of a strict mathematical formalism, so a more direct expression of this in language form is conceivable and will be further investigated.

Local code The focus of IMP is mostly on communication and synchronization between tasks and processes. While unifying several mechanisms in one framework, this leaves the aspect of local computation inside a task unspecified. And in fact, the MPI and OpenMP demonstrations above rely on different user-supplied routines for these computations. It would be a topic in compiler research to generate these codes automatically from an abstract specification.

3.3 Conclusion

IMP is a system for describing parallel algorithms, independent of the parallelism mode. It enables a task parallel execution model even when the high level expression looks data parallel, which enables application programmers to express data parallelism in their code without suffering from the performance consequences of bulk synchronous parallel (BSP) execution models.

Practically, we have made a strong case that IMP algorithms can be executed efficiently on a variety of parallel architectures, and in fact that it allows the same source code to be interpreted in multiple parallelism modes, such as message passing or task graphs.

References

[1] Arvind and David E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[2] Christopher Barton, Cálin Caşcaval, George Almási, Yili Zheng, Montse Farreras, Siddhartha Chatterje, and José Nelson Amaral. Shared memory programming for large scale machines. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 108–117, New York, NY, USA, 2006. ACM.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38:27–51, 2012.

[4] Yaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the fourth symposium on the frontiers of massively parallel computation (Frontiers '92), McLean, Virginia, Oct 19–21, 1992*, pages 120–127, 1992.

[5] 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.

[6] Victor Eijkhout. A theory of data movement in parallel computations. *Procedia Computer Science*, 9(0):236 – 245, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012, Also published as technical report TR-12-03 of the Texas Advanced Computing Center, The University of Texas at Austin.

[7] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. Appl Math.*, 14:1390–1411, 1966.

[8] 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.

[9] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, Guy K. Steel Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[10] Walid Najjar and Jean-Luc Gaudiot. Macro data-flow architecture. In John A. Sharp, editor, *Data Flow Computing: Theory and Practice*, pages 272–291. Ablex Publishing Corporation, Norwood, New Jersey, 1992.

[11] Peter Newton and James C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 167–177, New York, NY, USA, 1992. ACM.

[12] 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.

[13] Jack Poulson, Bryan Marker, Jeff R. Hammond, and Robert van de Geijn. Elemental: a new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. submitted.

[14] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[15] John A. Sharp. *Dataflow Computing: Theory and Practice*. Intellect Books, 1992.

[16] D. B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP, 1996.

[17] 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.

[18] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[19] 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.