TACC Technical Report TR-14-02

A Demonstration of Integrative Parallelism

Victor Eijkhout[*]

December 30, 2013

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

Abstract

The Integrative Model for Parallelism is a new model for expressing and analyzing parallel algorithms. It is intuitively clear that the theory expresses concepts that are analogous to for instance MPI messages and OpenMP (version 3) task dependencies.

However, in an eminently practical field like High Performance Computing the proof is in the pudding, so this paper presents a prototype implementation and compares it to handcoded equivalent MPI and OpenMP codes.

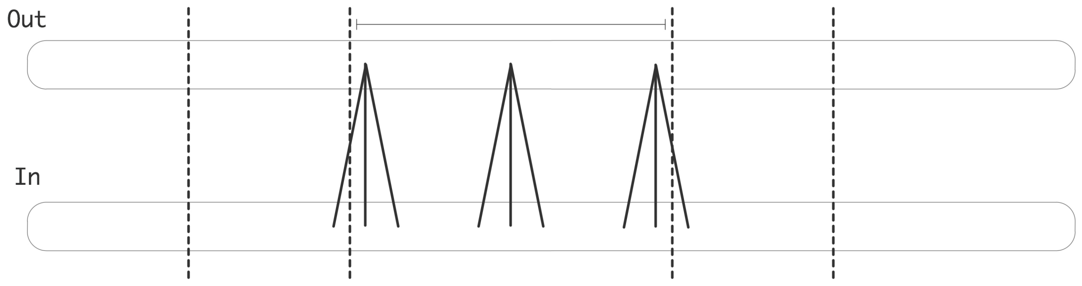
1 Introduction

In this paper we introduce the Integrative Model for Parallelism (IMP), a theoretical system for describing, analyzing, and implementing operations on distributed data; by means of a software prototype we show that the system is implementable with efficiency comparable to hand-coded software.

To show how we can theoretically derive data dependence patterns we consider a simple example: the three-point operation

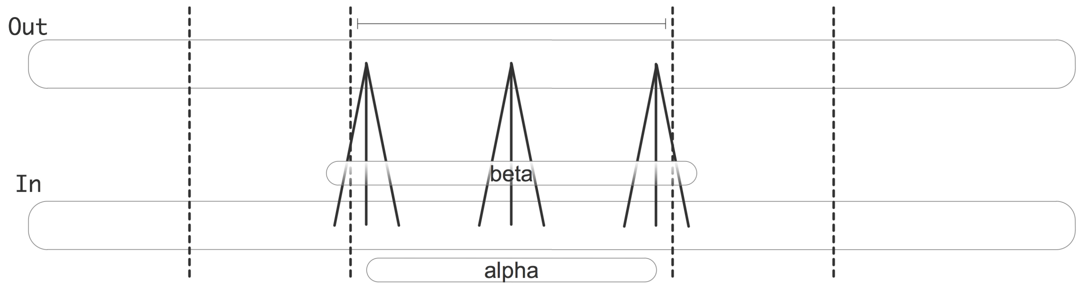
$$\forall_i \colon y_i = f(x_i, x_{i-1}, x_{i+1}).$$

We illustrate this graphically by depicting the input and output vectors, stored distributedly 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.

Now, we can in fact identify a second distribution that differs from the traditional notion. We let the β-distribution be defined as the mapping from each processor to all of its needed input elements. The second illustration depicts these two distributions for one particular process:



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. In message passing, these dependences obviously corresponds to actual messages: for each process p, the processes $q \in \beta(p)$ 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 kernel-task is dependent on. In this case, the transformation from α to β-distribution gives rise to a *dataflow* formulation of the algorithm[1].

This example shows that the IMP model can make some claims to expressiveness in dealing with parallel algorithms. 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

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

Below we will develop a Domain-Specific Language (DSL) to express this in languages such as C/C++ or Fortran.

We will now give a brief discussion of the theory of IMP, and how it relates to other parallel systems.

1.1 Preliminary discussion

Our basic theoretical concept is that of a *distribution*, which is a natural concept in High Performance Computing (HPC). Parallel scientific computing is often about spreading large datasets, such as matrices or Finite Element meshes, over a number of processors to increase the problem size that can be handled and reduce the running time. Thus, the need is clear for a description of how a large dataset is partitioned.

The concept of distribution is not new. For instance, High Performance Fortran (HPF) [9, 7] and Universal Parallel C (UPC) [3] allow the user to specify the distribution of parallel objects in code with otherwise sequential semantics. The conceptual attraction to such an approach is eloquently formulated in [10]:

> [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

1. Like distributions, dataflow is not a new concept. However, we give a new twist on it: we do not propose programming dataflow directly. Rather, our system *derives* the dataflow interpretation of the kernel.

> 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[.]

After this nod to history, we argue that our ideas differ in a number of respects.

- First of all, the abstract concept of distribution is clear, but the precise definition is not. While existing systems consider, explicitly or implicitly, a distribution as a mapping from data elements to processors (or processes), we turn this around. Considering a distribution as a mapping from processors to data it becomes possible to reason about redundantly duplicated data, and even redundantly duplicated work.
- Next, we argue that this approach of distributions as hints to the compiler is too limited. Instead we argue that distributions should be explicit objects in a parallel programming language. This idea has been implemented in systems such as PETSc [1, 2] and Trilinos [6]. Dynamic distributions are an absolute necessity in irregular applications, or contexts where data is dynamically created or refined.
- Building on the previous two points, the motivating example above showed that merely having a distribution for describing the storage of an object is not enough. Data traffic patterns are defined by the interaction of this distribution with the distribution that describes how data is used. Since our model allows for expression of the latter, we can formally derive data motion patterns as rigorously defined objects that can exist on the language level, thus facilitating cost analysis, scheduling, and resilience of execution.
- Finally, we point out that a programming system with sequential semantics, as described above, need not imply synchronized execution, for instance as in the Bulk Synchronous Parallelism (BSP) model [11, 13]. It has recently been recognized that execution needs to progress asynchronously in order to achieve high performance [5, 8]. Our model marries sequential semantics to asynchronous execution.

Finally, we argue that our system of distributions has interpretations beyond the obvious one of message passing. By reinterpretation of messages as data dependencies we show that shared memory task programming becomes a second target layer for a compiler of our IMP system.

1.2 Outline

We will give a shortened presentation of the IMP model, leaving out examples and footnotes, in order to work up to the presentation of the prototype software. We conclude with an discussion to evaluate the state of affairs and possible future directions.

2 Distributions

Informally, a distribution is a description of how data is mapped to processors. For instance, the index space of an array could be distributed by assigning contiguous blocks of indices to processors.

While this is a valid view, it does not allow for redundant replication of data on multiple processors, which happens for instance in an all-gather call, or when a 'halo region' is established. Thus, we turn the definition of distribution around and view it as a, potentially non-disjoint, mapping of processors to sets of indices.

With this approach we do not consider the result of an all-gather, or a halo construction, as local data. Instead we consider this as merely another distribution of some data object. This strategy will be seen to be a powerful tool for expressing and analyzing algorithm.

The communication involved in constructing this second distribution can now be considered as a formally defined transformation object. We will do this in section **??**. Such a transformation contains a full abstract description of all data dependencies, and as such can be compiled to existing parallel programming systems. We will consider this point further in section 4.

2.1 Basic definition

Rather than immediately taking about distributions of data, we start by considering the distribution of the index set of a data object.

Let us then consider a vector[2] of size N and P processors. A distribution is a function that maps each processor to a subset of N[3]:

$$u\colon P \to 2^N. \tag{1}$$

Thus, each processor stores some elements of the vector; the partitioning does not need to be disjoint.

2.2 Distributed objects

Let x be a vector and u a distribution, then we can introduce an elegant, though perhaps initially confusing, notation for distributed vectors[4][5]:

$$x(u) \equiv p \mapsto x[u(p)] = \{x_i\colon i \in u(p)\}.$$

That is, $x(u)$ is a function that gives for each processor p the elements of x that are stored on p according to the distribution u.

We note two important special cases:

2. We can argue that this is no limitation, as any object will have a linearization of some sort.
3. We make the common identification of $N = \{0,\ldots,N-1\}$ and $P = \{0,\ldots,P-1\}$. Likewise, N^M is the set of mappings from M to N, and thereby 2^N is the set of mappings from N to $\{0,1\}$; in effect the set of all subsets of $\{0,\ldots,N-1\}$.
4. We use parentheses for indicating distributions; actual vector subsections are denoted with square brackets.
5. We use set notation in the rhs of this definition; sometimes we will consider the rhs as an ordered set. This should not lead to confusion.

- $x(*)$ describes the case where each processor stores the whole vector.
- $x(n)$ describes the case where each processor stores $x[n]$.

We now see the wisdom of defining distributions as mapping from processors to index sets, rather than the reverse: the result of operations such as an all-gather now corresponds to a vector with elements that are replicated across processors, and the result of a broadcast is a single replicated element.

The concept of vector distributions is easily extended to one-dimensional matrix distributions. If u is a vector distribution, we can define

$$A(u,*) \equiv p \mapsto A[u(p),*]$$
$$A(*,u) \equiv p \mapsto A[*,u(p)]$$

True two-dimensional distributions are possible too, as are sparse matrices.

2.3 Transformed distributions

So far we have only shown distributions that are statically specified. It is also possible to derive new distributions from old ones.

One way to derive a new distribution is to have it be induced by a function on indices. For instance, adding a constant increment to each index corresponds to shifting a distribution.

Formally, let f be a function $\mathbb{N} \rightarrow \mathbb{N}$ and u a distribution. We can define $f(u)$ as

$$f(u) \equiv p \mapsto \{f(i) \colon i \in p(u)\}.$$

As a simple example, let x have a distribution u, and let $\cdot \gg \cdot$ be the infix function $\lambda_{i,j} : i+j$ then $u \gg 1$ is 'u right-shifted by 1':

$$p \mapsto \{i+1 \colon i \in p(u)\}.$$

The distributed vector $x(u \gg 1)$ is then the scheme that puts $x(i+1)$ on a processor if $x(i)$ was there under the original u. In other words, $x(u \gg 1)$ is 'x left-shifted by 1'.

It is easy to image that a programming language based on distributions could feature statements such as

$$y(u) \leftarrow 2x(u) - x(u \ll 1) - x(u \gg 1). \qquad (2)$$

which has the formal interpretation that processor p executes

$$\forall_{i \in u(p)} \colon y(i) \leftarrow 2x(i) - x(i-1) - x(i+1).$$

3 Kernels and distributions

In the above we have informally used the word 'kernel'. More formally[6], we define a kernel as an operation between two distributed datasets, where the first data set functions purely as input, the second one purely as output. The motivating example in the introduction, which models one step in a Jacobi iteration, is a kernel in this sense. A Gauss-Seidel operation, on the other hand, is not a kernel in this sense, since the output data also functions as partial input.

As already sketched in the motivating example, we now associate with a kernel two distributions:

- The α-distribution is the way the input is stored. This is the notion of distribution that many previous parallel programming systems have used: it is given by the context in which a kernel is used, and is independent of the kernel.
- The β-distribution describes the way the input is used. This distribution is determined by the kernel, and is typically not the same as the α-distribution describing the storage of the input. In many cases, this distribution is a transformation of an existing distribution, such as '$u \gg 1$' in equation 2 above.

3.1 Formal definition

If v and w are distributions of the same index set, we can consider the transformation that takes one arrangement of that index set and turns it into the other. We let ourselves be inspired by the formula $v = u \circ u^{-1} \circ v$, which would imply

$$x(v) \equiv p \mapsto x(u)[u^{-1}v(p)],$$

stating how x distributed with u is transformed to a distribution with v through the mapping $u^{-1}v$.

Since distributions are not strictly invertible, we have to define the expression $u^{-1}v$ properly. We define

$$u^{-1}v\colon P \mapsto 2^P$$

as

$$u^{-1}v(p) \equiv \{q\colon u(q) \cap v(p) \neq \emptyset\}. \tag{3}$$

Such transformations can be motivated from the common *owner computes* model of parallel computing. Think of a code being composed of episodes consisting of

- Communication where distributed data is rearranged, typically starting with a disjointly partitioned object and arriving at a non-disjoint distribution such as for a ghost region; and
- Local computation, where a processor acts on data that is directly accessible to it, whether in node memory in a distributed memory context, or cache in a shared memory context.

6. But still to totally formally, for that we refer to our more elaborate writings.

In the communication phase we refer to the initial distribution as the α-distribution and the result as the β-distribution[7].

3.2 Dataflow

Now we see that the transformation from α to β-distribution gives us a *dataflow* interpretation of the algorithm by investigating the relation (3).

For each kernel and each process p we find a partial Directed Acyclic Graph (DAG) with an arc from each $q \in \alpha^{-1}\beta(p)$ to p. In the abstract interpretation, each q corresponds to a task that is a dependency for the task on p. Practically:

- In a message passing context, q will pass a message to p, and
- In a shared memory threading model, the task on q needs to finish before the task on p can start.

We have now reached the important result that our distribution formulation of parallel alorithms leads to an abstract dataflow version. This abstract version, expressed in tasks and task dependencies, can then be interpreted in a manner specific to the parallel platform.

4 Software realization

To argue that our ideas are efficiently implementable, we present a prototype code[8] 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.

For simplicity of exposition, we use the threepoint kernel of the introduction, repeated a number of times, in essence the one-dimensional heat equation. However, we hope the user sees that the ideas are general and allow for implementation of a wide variety of codes.

7. There is a deeper IMP theory that can handle somewhat more general notions of execution, for which see [4]. In terms of that theory we associate with the execution of a kernel $K = \langle \mathrm{In}, \mathrm{Out}, E \rangle$ four distributions.
   - The α-distribution of a kernel is $p \mapsto \mathrm{In}_p$;
   - The β-distribution is $p \mapsto \mathrm{In}(E_p)$;
   - The δ-distribution is $p \mapsto \mathrm{Out}(E_p)$; and
   - The ε-distribution is $p \mapsto \mathrm{Out}_p$.

Thus, every kernel consists of the $T(\alpha, \beta)$ traffic, followed by local computation, followed by the $T(\delta, \varepsilon)$ traffic.

8. The interested reader can inspect the full code at `https://bitbucket.org/VictorEijkhout/imp-demo`.

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;
}
```

However, we also create transformed distributions:

```
IMP_disjoint_blockdistribution
  *rightshift = new IMP_disjoint_blockdistribution(problem_environment,globalsize);
rightshift->operate(">>1");
IMP_disjoint_blockdistribution
  *leftshift = new IMP_disjoint_blockdistribution(problem_environment,globalsize);
leftshift->operate("<<1");
```

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_kernel *update_step =
    new IMP_kernel(step,output_vector,blocked);
  IMP_object
    *input_vector = all_objects[step-1];
  update_step->localexecutefn = &threepoint_execute;
  update_step->add_beta_vector( input_vector );
  update_step->add_beta_vector( input_vector,leftshift );
  update_step->add_beta_vector( input_vector,rightshift );
  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();
```

Discussion The above code shows first of all that we truly have an integrative notion of parallelism: the exact same main program is interpretable as efficient execution through MPI and OpenMP. Next, the expression of the algorithm is in global terms, giving us the sequential semantics that other parallel programming systems often strive for; section 1.1.

However, execution is not synchronized. For instance, in the MPI realization of the IMP code a kernel execution consists of sends/receives and local execution. Thus, the only synchronization is through local dependencies as expressed in MPI point-to-point operations. There are no BSP-style global synchronizations, other than those intrinsic in the algorithm, such as for norm calculations.

By separating the dependency analysis and execution we realize the *inspector-executor* model. If the distributions are invariant, for instance with the kernel appearing in a loop, this runtime transformation can be computed ahead of time. Its construction is then amortized in what is known as the *inspector-executor* model [12].

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 code uses the OpenMP v3 task mechanism, while the reference code uses the `omp parallel for` pragma. Thus, at low thread counts the OMP version has lower overhead. However, figure 1 shows that at 12 and 16 threads the two codes perform similarly. The OpenMP code does not show approximately linear scaling, probably because of the excess of available bandwidth at low core/thread counts. It is not clear why this bandwidth argument does not help the IMP code.

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 2 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.)

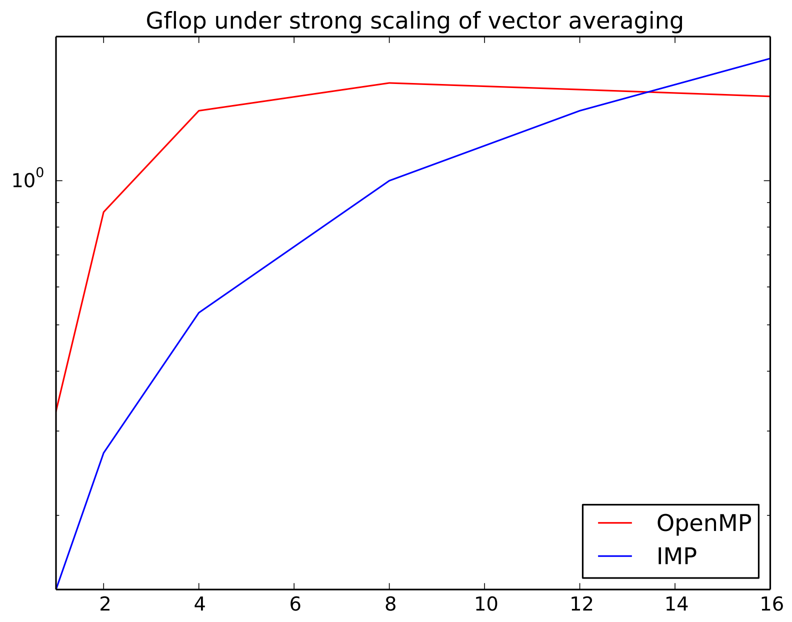


Figure 1: Strong scaling of IMP/OMP code versus OpenMP reference code

Heterogeneity Heterogeneous architectures are generally considered to be difficult to program. First of all, the programmer needs to decide whether to bother with two programming systems, typically MPI and OpenMP, or to do everything with one. (In the latter case, choosing OpenMP clearly limit the scale of the computation.) Then the choice needs to be made in what relative proportions the parallelism is assigned to the two systems.

With IMP this matter is greatly simplified. Since MPI and OpenMP, and presumably other systems, are all covered under the IMP syntax, only one system needs to be learned and used. Apportioning parallelism is also easier. This needs a word of explanation.

While we talk about a heterogeneous mix of programming systems, there is a structure to this mix that can be described as a product graph. There is a primary graph of the MPI tasks, superimposed on which each MPI task can be seen as a graph of threading tasks. There is no true mix in the sense that a thread on one MPI node can talk to a thread on another node, other than going through MPI.

Thus we can implement the most common type of heterogeneity by having an MPI-based IMP execution, where each task creates its own thread-based IMP execution. Using IMP here increases the flexibility of the application, since now the balance between MPI and threading can dynamically be shifted.

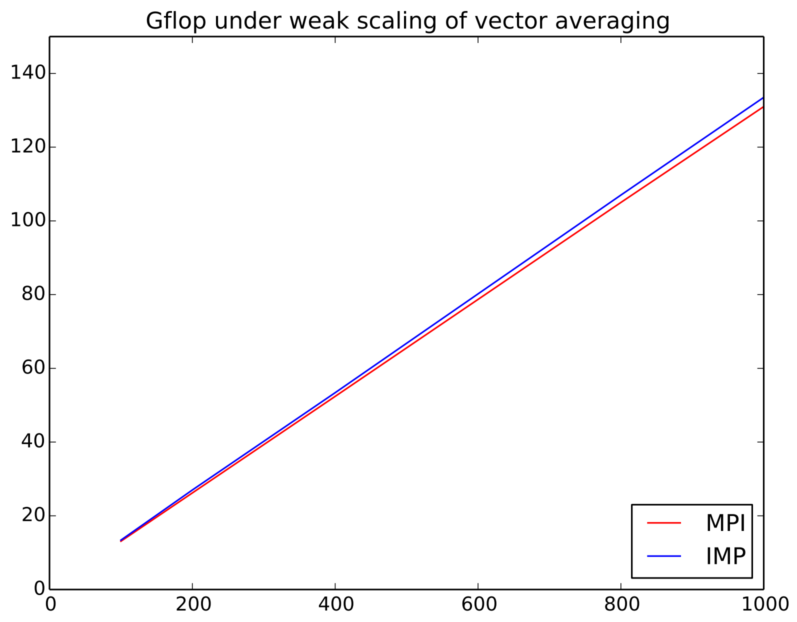


Figure 2: Weak scaling of IMP/MPI code versus MPI reference code

In this story we recognize the task 'spreading' and 'grouping' of the Degas project (`https://www.xstackwiki.com/index.php/DEGAS`). In effect, with IMP we offer an infrastructure for this notion.

Towards a language for parallel programming In this paper we have only proposed a DSL, aimed at the data dependency aspects of parallelism. We will now briefly dwell on whether we have the beginnings of a parallel language here. We need to cover two aspects: the data dependency part, and the 'local code' part.

Languages for parallel programming have not been a success story. Part of the reason for this is the inertia of the scientific user community, part of it is that languages have not managed to deliver on performance across the board. The major failing here is dealing with parallelism, especially distributed memory parallelism.

We argue that the DSL outlined above is systematic enough that it could conceivably be the backend of a true parallel language.

In the current IMP design, the main user task is that of identifying the correct 'β-vectors'. This task can conceivably be done by a compiler of a hypothetical parallel language, if this language is suitably designed. Starting from a traditional loop-based language will not work, for all the reasons that parallelizing compilers have not been successful. However, in the motivating example we indicated that programming in terms of distributions may provide the required high level view. For instance, statements such as

$$y(d) \leftarrow F\bigl(x(d'), x(d'')\bigr)$$

where d', d'' are transformations of d, are both well-defined, and easily translatable to the above DSL.

The other half of the story is generation of efficient local code. Traditional compiler technology could come to the rescue here, since the required transformations can be as simple as the transformation of the iteration space that an MPI programmer regularly engages in. Of course, we posited in the previous paragraph that we do not want to start from a traditional arrays and loops-based language. Maybe the way out is to start with a Chapel-like notion of indicating an operation on a formally described index set.

5 Discussion and conclusion

In this paper we have presented the Integrative Model for Parallelism (IMP), motivated by a practical example. After presenting the theory we showed a concrete syntax for realizing this example by means of a DSL. Tests bear out that IMP code is comparable in efficiency to hand-coded MPI and OpenMP code.

However, apart from efficiency, the IMP model is an interesting development for several reasons. One is that it enables expressing parallel algorithms under quasi-sequential semantics. This will lower the threshold for parallel programming and increase programmer productivity, even for programmers already familiar with parallelism.

Another reason for programming in IMP is that it is platform-agnostic, and in fact allows the same source code to be interpreted as either MPI or OpenMP code. It does not require a great leap of the imagination to see that heteregenous programming through a unified model is a possible future development of IMP.

References

[1] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc webpage. http://www.mcs.anl.gov/petsc, 2011.

[3] 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.

[4] 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.

[5] Bill Gropp. Changing how programming think about parallel execution. Online lecture. `http://bit.ly/16jI1X9`.

[6] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[7] 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.

[8] Kyungjoo Kim and Victor Eijkhout. A parallel sparse direct solver via hierarchical DAG scheduling. Technical Report TR-12-05, Texas Advanced Computing Center, The University of Texas at Austin, 2012. to appear in ACM Transactions on Mathematical Software.

[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] 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.

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

[12] Alan Sussman, Joel Saltz, Raja Das, S. Gupta, Dimitri Mavriplis, and Ravi Ponnusamy. Parti primitives for unstructured and block structured problems, 1992.

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