A DSL for Integrative Parallel Programming

Victor Eijkhout
Texas Advanced Computing Center
The University of Texas at Austin
Austin, TX, USA
eijkhout@tacc.utexas.edu

Abstract—Parallel programming is commonly done through a library approach, as in the Message Passing Interface (MPI); directives, as in OpenMP; language extensions, as in High Performance Fortran (HPF); or whole new languages, as in Chapel. However, we argue that the concepts underlying these different programming systems show great commonality. Hence, we propose a Domain-Specific Language (DSL) that expresses an abstraction of these common concepts. As we show by means of a prototype that uses both MPI and OpenMP tasks as backend, this common vocabulary can then be expressed in multiple parallelism types.

Keywords—Parallel programming, Domain-specific language, hybrid programming

I. INTRODUCTION

In this paper we introduce language concepts and a DSL based on the Integrative Model for Parallelism (IMP) [1], a theoretical system for describing, analyzing, and implementing operations on distributed data. By means of a software prototype we show that the system is interpretable in multiple parallelism types with efficiency comparable to hand-coded software.

A. Motivating example

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

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

(Stencil operations are much studied; see e.g., [2] and the polydral model, e.g., [3]. However, we claim far greater generality for our model.) 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 $\alpha$-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 $\beta$-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 $\beta$-distribution, unlike the $\alpha$ 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 $\alpha$ to $\beta$-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 $\alpha$-distribution on the input with tasks that produce this input, then the $\beta$-distribution describes what input-producing tasks a kernel-task is dependent on. In this case, the transformation from $\alpha$ to $\beta$-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 $\alpha$-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

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

Below we will develop a 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. Detailed discussion will follow.

B. Preliminary discussion

IMP is rooted in some well-known concepts, but gives a new perspective on them. We briefly discuss our relation to
Our basic **programming** concept is that of kernels: the expression of a mathematical operation on parallel data, expressed with sequential semantics. The conceptual attraction to such an approach is eloquently formulated in [4]:

"An 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."

This quote indicates the importance of the basic **data** concept of a **distribution**. The concept of distribution appeared for instance in Universal Parallel C (UPC) [5]; in HPF [6], [7] 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 [8] and has recently seen a revival in Directed Acyclic Graph (DAG) based schedulers [9]. While acknowledging the universality of the concept, we argue that it should not be the programming model.

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

- Existing systems typically specify distributions as annotations on data allocation. This only works for regular data; we argue that distributions should be first-class objects; an idea implemented in PETSc [10] and Trilinos [11]. Dynamic distributions are an absolute necessity in irregular applications, or contexts where data is dynamically created or refined.

- Next, distributions are not a sufficient mechanism for efficient execution, especially in distributed memory. For instance, compilers that analyze data dependencies will typically give many small messages. Since data dependencies (read: messages) result from the interaction between algorithm and data distribution, we argue that distributions should also be used in the specification of the algorithm. We will show how this can markedly improve this efficiency; for as far as we are aware, no systems other than IMP currently use this strategy.

- The cost of data motion, both vertically through the memory hierarchy and horizontally through the network, is so high that modern programming systems really need to account for it. In IMP we base execution on dataflow, implying that tasks know where data is coming from and where it is going to. This means that locality and latency can be exploited formally, and exploited during execution. However, explicit dataflow programming, as in [9], is breaking the abstraction of expressing the mathematical operations through kernels, so we use it only as intermediate representation.

- 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 [12], [13]. It has recently been recognized that execution needs to progress asynchronously in order to achieve high performance [14], [15]. Our model marries sequential semantics to asynchronous execution resulting from our dataflow representation.

In this discussion we have been deliberately vague about what type of parallelism (shared memory, distributed, accelerators) we target. We will argue that our ideas give an overarching abstraction to all of these.

### C. Outline

The outline of this paper is as follows. We define the basic IMP concepts of distributions and distribution transformations in sections II and III respectively. (The IMP model was described earlier in [1] in graph/set-theoretic terms, but without formalizing the concept of distribution which is needed for implementation.) Section IV has a first example showing these concepts in action. We show how these concepts can be used as algorithm analysis tools in section V. Finally, in section VI we discuss how our concepts can be realized in software. We conclude with a short discussion.

### II. 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.

#### A. Basic definition

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

Let us then consider a vector\(^1\) of size \(N\) and \(P\) processors. A distribution is a function that maps each processor to a subset of \(N\):

\[
\mu : P \rightarrow 2^N.
\]

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

\(^1\)We can argue that this is no limitation, as any object will have a linearization of some sort.

\(^2\)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\}.
B. 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\(^3\):

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

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:

\( x(\ast) \) 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

\[
A(u, \ast) \equiv p \mapsto A[u(p), \ast]
\]

\[
A(\ast, u) \equiv p \mapsto A[\ast, u(p)]
\]

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

C. Operations on distributions

Most systems that use distributions only use them for declaring objects. We now introduce the concept of operations on distributions, which allows one to formulate algorithms in terms of distributions, leading to powerful analysis of data dependencies.

Let us first consider operations on distributions induced by a simple function in indices. Let \( f \) be a function \( \mathbb{N} \rightarrow \mathbb{N} \) and \( u \) a distribution. We can define \( f(u) \) as

\[
f(u) \equiv p \mapsto \{ f(i) : i \in u(p) \}.
\]

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’:

\[
p \mapsto \{ i + 1 : i \in u(p) \}.
\]

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’. (Note the left/right distinction: the algorithm requests an element from the right, which involves a message to the left.)

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

\[
y(u) \leftarrow 2x(u) - x(u \ll 1) - x(u \gg 1).
\]

\(^3\)We use parentheses for indicating distributions; actual vector subsections are denoted with square brackets.

\(^4\)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.

which has the formal interpretation that processor \( p \) executes

\[
\forall i \in u(p) : g(i) \leftarrow 2x(i) - x(i - 1) - x(i + 1).
\]

D. Distributions from algorithms

From equation (2) we see that there is a distribution \( u \cup u \ll 1 \cup u \gg 1 \) associated with the algorithm. Thus we see the wisdom of, firstly, having dynamically constructed distributions, which, secondly, can be non-disjoint.

III. Kernels and distributions

In the above we have informally used the word ‘kernel’. More formally, 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 \( \alpha \)-distribution is the way the input is stored, and the \( \gamma \)-distribution describes the output; these two are often identical. Many programming systems have used this concept of distribution, although usually not as first-class object: it is a description of data independent of the kernel operation.

- It is an innovation of IMP to introduce the \( \beta \)-distribution, a dynamically generated object describing the way the input is used. This distribution is determined by the kernel; in many cases, this distribution is a transformation of an existing distribution, such as \( \cdot \gg \cdot \) in equation 2 above.

A. 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

\[
u^{-1}v : p \mapsto 2^p
\]

as

\[
u^{-1}v(p) \equiv \{ q : u(q) \cap v(p) \neq \emptyset \}.
\]

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.

In the communication phase we refer to the initial distribution as the \( \alpha \)-distribution and the result as the \( \beta \)-distribution. The former is the stored data distribution, while the latter is defined by the algorithm. Thus, \( \alpha^{-1}\beta \) expresses the interaction between data and execution, summarizing the essence of the parallel algorithm.

B. Dataflow

Now we see that the transformation from \( \alpha \) to \( \beta \)-distribution gives us a dataflow interpretation of the algorithm by investigation of the relation (3). We note that this task dataflow can be derived formally from the algorithm, rather than being explicitly programmed [9], and does not depend on a compiler (the prototype IMP software is a few hundred lines of C++ code) or runtime system. It achieves perfect aggregation of small messages, and data can be made available (e.g., through message passing) as early as possible, thus hiding latency.

Graphically, in the threepoint example, we see that in order to build the \( \beta \)-distribution on one process we depend on a few other processes:

![Diagram](Image 59x128 to 321x421)

Formally, for each kernel and each process \( p \) we find a partial 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 \( p \), while 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 algorithms 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.

IV. Example: Matrix-Vector Product

The dense matrix-vector product provides a simple illustration of the principle that a parallel algorithm can be written in terms of distributions, and that data movement is formally derived. We make our point by giving two versions of the algorithm, and showing how the resulting very different data traffic patterns follow automatically.

We write the basic operation \( \forall i : y_i = \sum_j a_{ij} x_j \) by splitting the computation into temporaries and reduction:

\[
\forall i : y_i = \sum_j a_{ij} x_j = \sum_j t_{ij}, \quad t_{ij} = a_{ij} x_j.
\]

For simplicity we assume a square matrix, and the same distribution \( u \) is used for both input and output, that is, we assume that \( x \) is distributed as \( x(u) \) on input, and \( y \) as \( y(u) \) on output. The work distribution is determined by the decision to let \( t_{ij} \) be computed where \( a_{ij} \) is stored. We will now derive communication for the variants of the product operation induced by the matrix distribution by rows and columns.

Product by rows : We write the product by rows as

\[
\begin{cases}
t(u, \ast) & \leftarrow A(u, \ast) \cdot x(u) \\
y(u) & \leftarrow \sum_j t(u, \ast).
\end{cases}
\]

where dot-times indicates a column scaling by right multiplication, and we reason as follows:

• The input \( x \) has an \( \alpha \)-distribution of \( x(u) \) and a \( \beta \)-distribution of \( x(\ast) \). Thus, the transformation to \( x(\ast) \) is an allgather.

• The temporaries are distributed upon construction as \( t(u, \ast) \), which is the correct distribution to enable the reduction, so no communication is needed there.

• On output, \( y \) is distributed as \( y(u) \), which is the desired distribution, so no further communication is needed.

Product by columns : Similarly, we write the matrix-vector product by columns in distribution terms as

\[
\begin{cases}
t(\ast, u) & \leftarrow A(\ast, u) \cdot x(u) \\
y(u) & \leftarrow \sum_j t(\ast, u).
\end{cases}
\]

The reasoning is now:

• \( A(\ast, u) \) describes the distribution of \( A \) by columns.

• \( x \) is distributed as \( x(u) \), which is the required distribution for the \( t_{ij} \) calculation.

• the \( \alpha \)-distribution of the temporaries, coming out of the first kernel, is \( t(\ast, u) \), while the \( \beta \)-distribution is \( t(u, \ast) \), so a data transposition is needed here.

• \( y(u) \) is the resulting reduction as before.

We note that the transpose and subsequent reduction can be merged into reduce-scatter operation. Our framework does not derive such a merge, rather we assume that software tools can effect this [16].

Brief discussion : While this example is fairly simple, it exhibits the main features of the IMP model.

• Distribution notation allows for a natural expression of parallel algorithms, with something resembling sequential semantics.

• A kernel features an \( \alpha \)-distribution, which describes the object at the start of the parallel kernel, and a \( \beta \)-distribution, describing the rearrangement of the data that is necessary for local computation. The transformation from \( \alpha \) to \( \beta \)-distribution is again constructable as a formal object. Thus, we can reason over it, or software middle layers can use it to optimize scheduling or introduce resilience.
V. Algorithm Analysis

In section III we introduced the formal transformation from one distribution $u$ to another $v$ as the function $u^{-1}v$ defined by equation (3) (here reproduced)

$$u^{-1} \circ v(p) \equiv \{ q : u(q) \cap v(p) \neq \emptyset \}.$$  

This transformation is most naturally applied to the $\alpha, \beta$ distributions of a kernel, and allows us to analyze various aspects of an algorithm. In this section we will see some examples, taken both from distributed memory programming and from task models.

A. Aggregation

For each $q \in \alpha^{-1} \beta(p)$, the intersection

$$\alpha(q) \cap \beta(p)$$

consists of the data communicated between $q$ and $p$. This set can be derived from an analysis of where the kernel is applied in the context of a computation, that is, it can be derived from the program structure. The practical implication is that communications can be scheduled as soon as inputs are available, and that this communication can be fully aggregated.

B. Cost measures

There are various ways of attaching a cost to a parallel algorithm. We indicate how we can model several of them.

- First of all, for any $p$, $\alpha^{-1} \beta(p)$ is the set of processors that send to $p$.
- In a shared memory dataflow interpretation, $\alpha^{-1} \beta(p)$ are the processes that the task\footnote{We did not formally define tasks. Loosely, they correspond to the kernel execution on a single process.} on $p$ depends on.
- Counting the quantity $\alpha^{-1} \beta(p)$ corresponds to measuring latency: the number of messages arriving at a processor $q$ is

$$|\alpha^{-1}(\beta - \alpha)(p)| = |\{ q : \alpha(q) \subset \beta(p) \} - \{ p \}|.$$

- Assuming no duplication in the $\alpha$ distribution, we can model total message volume as:

$$|\{ \beta - \alpha \} (p)|.$$

- In a shared memory context the previous formula also gives the size of the ‘ghost region’, which corresponds to the amount of temp local store that is needed to perform local calculations.
- This means that we have a tool for programming for explicitly managed local storage.
- Going on to task dataflow we can model execution start time:

$$t(p) = \max_{q \in \alpha^{-1} \beta(p)} t(q) + e(p). \quad (4)$$

where $e(p)$ is the local execution time of the task.
- That last formula is based on the same asynchronous execution model that underlies the $logP$ model. On the other hand, BSP execution time is easier since it uses global barriers

$$t(p) = \max_q t(q).$$

- Formula (4) closely resembles concepts in traditional distributed systems theory. For instance, we recognize Lamport’s notion of network time \cite{17}, where local clocks are updated to ensure proper ordering of message sends and receives. With a unit execution time $e(p)$, the ‘time’ becomes a counting of how many tasks causally predate the current.
- We can do more with tasks. For instance, one might wonder when the input $x$ of a kernel application $y \leftarrow Kx$ can be reused. The answer is that $x(p)$ can be reused when the tasks $y(q)$ with

$$q \in \beta^{-1} \alpha(p)$$

are finished.

VI. Software Realization

To argue that our ideas are efficiently implementable, we present a prototype code\footnote{The interested reader can inspect the full code at https://bitbucket.org/VictorEijkhout/imp-demo.} 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. 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:

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

Next we create receptors for data and work objects

```c
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:

```c
*problem_environment =
```
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();

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

A. Comparison to reference code

OpenMP: We compared the threepoint averaging kernel on $3 \times 10^7$ 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 1 then shows that the overhead of the IMP mechanisms is negligible. Both tasking codes do not show approximately linear scaling, probably because of the 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 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.)

B. Discussion

This prototype is limited. However, that is mostly in the extent of its implemented functionality, rather than its nature.
Higher dimensional distributions as well as sparse data structures can easily be accommodated.

On the other hand, the 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 I-B.

Some of the concepts in IMP are not new; for instance, distributions have been part of several parallel systems before. However, none of these have proven to be efficient. The difference lies in our identification of the $\beta$-distribution and the fact that all concepts relevant to data dependencies are explicitly first-class object in our system, hence amenable to analysis and optimization.

Resulting from the global sequential semantics noted above, the IMP system gives the impression of being Bulk Synchronous in nature. This is only true in a very weak sense. The IMP algorithm specification has sequential semantics, but the actual execution is unsynchronized, obeying only the data dependencies that are intrinsic to the algorithm. 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.

**VII. RESEARCH DIRECTIONS**

A. 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 limits 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 probably 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.

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.

B. Towards a language for parallel programming

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 ‘$\beta$-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 parallelism in terms of distributions may provide the required high level view. For instance, statements such as

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

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

C. Local code

In this paper we have only proposed a DSL, aimed at the data dependency aspects of parallelism. Close inspection of the demonstration code shows that the local function is a different code snippet for the two target systems. For a truly integrative system the difference between the task-local snippets should also be handled by some system component. However, this
problem can not be solved without including in the IMP system a substantial dose of traditional compiler technology, thus taking it out of the realm of being a DSL. This section offers some thoughts on the matter.

The treatment of task-local code can take various guises. For instance, the distinction between MPI and OMP code in this paper lies mostly in a shifting of indices. For accelerator devices it would involve allocating device-local memory and generating offload instructions. A very different example would be time-blocking a repeated kernel so that a single transfer suffices for multiple kernel iterations [18], [19], [20], [21]. Such transformations could be handled by compiler technology such LLVM or the Rose compiler. However, this is a research topic in itself.

One question is what we use as a basis for these transformations. It might in fact be possible to start from a formulation in a regular loop-based language. Another possibility is to start with a Chapel-like notion of indicating an operation on a formally described index set.

VIII. 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 on a demonstration prototype 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 in global terms 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 heterogenous programming through a unified model is a possible future development of IMP.

REFERENCES

