**TACC Technical Report IMP-03**

# The type system of the Integrative Model

Victor Eijkhout*

February 22, 2016



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

**Abstract**

In a previous note we described the mathematical ideas behind the Integrative Model for Parallelism. Here we formalize these as a type system. This gives us rigorous definitions of all the concepts; an implementation of IMP can be based on these.

The following IMP reports are available or under construction:

**IMP-00** The IMP Elevator Pitch
**IMP-01** IMP Distribution Theory
**IMP-02** The deep theory of the Integrative Model
**IMP-03** The type system of the Integrative Model
**IMP-04** Task execution in the Integrative Model
**IMP-05** Processors in the Integrative Model
**IMP-06** Definition of a 'communication avoiding' compiler in the Integrative Model
**IMP-07** Associative messsaging in the Integrative Model (under construction)
**IMP-08** Resilience in the Integrative Model (under construction)
**IMP-09** Tree codes in the Integrative Model
**IMP-10** Thoughts on models for parallelism
**IMP-11** A gentle introduction to the Integrative Model for Parallelism
**IMP-12** K-means clustering in the Integrative Model
**IMP-13** Sparse Operations in the Integrative Model for Parallelism
**IMP-14** 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
**IMP-15** Collectives in the Integrative Model for Parallelism
**IMP-16** Processor-local code generation (under construction)
**IMP-17** The CG method in the Integrative Model for Parallelism (under construction)
**IMP-18** A tutorial introduction to IMP software (under construction)
**IMP-19** Report on NSF EAGER 1451204.
**IMP-20** A mathematical formalization of data parallel operations
**IMP-21** Adaptive mesh refinement (under construction)
**IMP-22** Implementing LULESH in IMP (under construction)

# 1 Motivation

In *IMP-01* we described the mathematical theory of the Integrative Model for Parallelism (IMP). We are probably not exaggerating in saying that IMP is more based on a mathematical theory than other systems for parallel programming. This also means that its programming concepts are direct translations of these mathematical concepts. In this note we reintroduce the mathematical ideas, but immediately translate them into program concepts.

# 2 A type system for the IMP model

In this section we will develop a formal treatment of parallel computations. It will show how the concepts of the IMP model fit together.

## 2.1 The basic concepts: data parallel functions

We consider only one datatype: the real numbers.

**Datatype** Real: Real numbers

Real

Computation is modeled by the set Func of functions $f$ from multiple real inputs to a single real output[1]:

**Datatype** Func: functions with a single output

$\text{Func} \equiv \text{Real}^k \to \text{Real}$

where $k$ is some integer. We could make different sets of functions depending on the value of $k$, but let's not.

As a first tool for parallel computing we define the set Array of arrays as indexed sets of real numbers[2]:

**Datatype** Array: arrays of real numbers

$\text{Array} \equiv \text{Real}^N$

1. Notation: the '$\equiv$' relation stands for 'is defined as'; the '$\to$' arrow connects sets that are input and output of a mapping, while the '$\mapsto$' arrow is used to indicate the actual definition of the mapping.
2. We identify the number $N$ with the set $\{0,\ldots,N-1\}$. That sounds recursive and it is. It is also well defined.

so that $x \in$ Array means that

$$\forall_{i \in N} : x(i) \in \text{Real}.$$

We will leave the finite index set $N$ (note: not the set of natural numbers) typically unspecified. This should not lead to confusion.

Clearly we will be talking about sets of indices quite a bit, so we define[3]

**Datatype** Ind: sets of indices

$$\text{Ind} \equiv 2^N$$

Now we can define how to apply functions in Func to arrays, producing output arrays. We call such a parallel function application a 'kernel' (see section 2.3 for the definition), and it is our, to be proven, contention that parallel programs can be described as the composition of kernels. For now we analyze the structure of individual kernels.

Let $x, y \in$ Array and let the function $f$ compute each element of $y$ from certain elements of $x$. We use a function $\sigma$ (or $\sigma_f$ to indicate its provenance explicitly) called the 'signature function' to determine the mapping from output indices to input sets of indices:

**Datatype** Signature: Signature of data parallel functions

$$\text{Signature} \equiv N \to \text{Ind}$$

We will use the obvious notational shorthand that

$$\text{for } S \in \text{Ind}, \text{ define } \sigma(S) = \cup_{i \in S} \sigma(i)$$

and

$$x\big(\sigma(i)\big) = \{x_j : j \in \sigma(i)\}.$$

**Example 1** *For instance, in our motivational example of three-point averaging (*IMP-11*, section 3), $y_i$ was computed from $x_{i-1}, x_i, x_{i+1}$, making $\sigma(i) = \{i-1, i, i+1\}$.*

**Remark 1** *The function $\sigma_f$ can also be interpretated as an adjacency matrix: a matrix with elements that are either zero or one. In the case that $f$ computes a sparse matrix vector product, that will be the right representation for the implementation. For other operations, for instance stencil-based, different representation of the signature function will be more efficient.*

3. If $A, B$ are sets, $A^B$ denotes all functions $B \to A$. This makes $2^N$ the set of functions from $N$ to $\{0,1\}$, in other words, the set of all subsets of $N$, which no coincidentally has $2^N$ element.

**Remark 2** *In most of our exposition we will assume a single input object. For operations that require multiple inputs, such as a vector inner product, we need multiple inputs and accordingly multiple signature functions. This makes us realize that the signature is actually a property of the dependence on the input vector, rather than of the algorithm as such.*

Armed with the definition of $\sigma_f$, we can define the *data parallel* application of $f$:

$$\texttt{map}\, f\colon \text{Array} \to \text{Array} \qquad \text{is the mapping} \qquad y = (\texttt{map}\, f)(x) \equiv \forall_i\colon y_i = f\big(x(\sigma_f(i))\big)$$

We will usually omit the `map` operator and the index gathering, and write the array operation as $y = fx$.

### 2.2 Data distribution

We need to start with a processor type.

**Datatype** Proc: processing elements

Proc

These processors are fairly abstract, describing everything from cluster nodes to short-lived entities such as threads: it is really a labeled subdivision of data/work. Initially we do not put a structure on them, for instance considering the set Proc isomorphic with $P = \{0,\ldots,P-1\}$, but in general we don't actually use ordering properties.

Since we are mostly interested in parallel computing, we need to introduce concepts related to the parallel distribution of data. A distribution is a mapping from processors to subsets of indices:

**Datatype** Distr: distributions of data over the processing elements

Distr $\equiv$ Proc $\to$ Ind

Traditionally, distributions are considered as mappings from data elements to processors, which assumes a model where a data element lives uniquely on one processor. We have already shown the concept of a $\beta$-distribution, where some elements are present on more than one processor. For this reason, and for scenarios such as redundant computation, we reverse the standard definition and define a distribution as a mapping of processors to elements.

Occasionally we need to compare distributions:

$$u, v \in \text{Distr} \quad \text{then} \quad u \subset v \equiv \forall_p\colon u(p) \subset v(p).$$

Using the shorthand established above we can now distribute arrays:

**Datatype** DistrArray: distributed arrays

$$\text{DistrArray} \equiv \text{Array} \circ \text{Distr} = \text{Proc} \to 2^{\text{Real}}$$

This definition states that, given $x \in \text{Array}$ and $u \in \text{Distr}$,

$$x(u) \qquad \text{is the function} \qquad x(u)\colon\ p \mapsto x\big[u(p)\big] = \{x_i\colon\ i \in u(p)\}$$

where we use square bracket notation to indicate vector indexing.

At the expense of a lot more notation we could have defined DistrArray in such a way to give us the operator

**Function** distr: the distribution of an array

$$x(d) \in \text{DistrArray} \quad \text{then} \quad \text{distr}\big(x(d)\big) = d$$

For ease of exposition we will just posit the existence of this operator and spare ourselves the complication.

Next we define operations on distributions. This will, for instance, allow us to represent the signature function of a stencil operation more compactly than as an adjacency matrix; see above. Distribution operations are based on functions $g \in G$ from the index set to itself (or to a different index set, however we can always augment one or the other with dummy indices to give them equal cardinality):

**Datatype** Oper: operations on integers

$$\text{Oper} \equiv N \to N$$

Alternatively:

**Datatype** Signature: operations on integers

$$\text{Signature} \equiv N \to 2^N$$

We extend this operation to sets of indices:

**Datatype** Oper: operations on sets of indices

$$\text{Oper} \equiv \text{Ind} \to \text{Ind}$$

**Datatype** Signature: operations on integers

Signature $\equiv$ Ind $\to$ Ind

with the obvious definion that, for $g \in$ Oper, $S \in$ Ind:

$$g(S) = \{g(i) \colon i \in S\}.$$

or $\sigma \in$ Signature, $S \in$ Ind:

$$\sigma(S) = \{\sigma(i) \colon i \in S\}.$$

Now we can apply these operations to distributions:

**Datatype** Oper: operations on distributions

Oper $\equiv$ Distr $\to$ Distr

**Datatype** Signature: operations on distributions

Signature $\equiv$ Distr $\to$ Distr

with the definition that, for $g \in$ Oper and $u \in$ Distr

$$g(u) = p \mapsto g(u(p)) \quad \text{where} \quad g(u(p)) = \{g(i) \mid i \in u(p)\}$$

or, for $\sigma \in$ Signature and $u \in$ Distr

$$\sigma(u) = p \mapsto \sigma(u(p)) \quad \text{where} \quad \sigma(u(p)) = \{\sigma(i) \mid i \in u(p)\}$$

We will use a similar process of extension to take the index function $\sigma_f$ (section 2.1) and extend it to distributions:

$$\sigma_f u \colon p \mapsto \sigma_f\big(u(p)\big).$$

### 2.3 Parallel computing with kernels

Given a scalar function $f \colon \text{Real}^k \to \text{Real}$, we described above its extension to

$$f \colon \text{Array} \to \text{Array}$$

by mapping $f$. In this section we consider the extension to

$$f\colon \text{DistrArray} \to \text{DistrArray}$$

which will give rise to some interesting concepts.

We can formally define a 'kernel' as a function applied between distributed objects:

**Datatype** Kernel: Data parallel functions between distributed arrays

$$\text{Kernel} \equiv \text{Func} \times \text{DistrArray} \times \text{DistrArray}.$$

If $K \in \text{Kernel}$ and $K = \langle f, x, y \rangle$, we adopt the notation $y = f(x)$ with the understanding that this is between distributed objects.

We define shorthand functions:

**Function** In, Out: components of a kernel

$$k = \langle f, x, y \rangle \quad \text{then} \quad \text{In}(k) = x,\ \text{Out}(k) = y$$

**Remark 3** *When we compose kernels to arrive at a full algorithm we impose the restriction that for each object $y$ there is* exactly *one kernel $k$ such that $y = \text{Out}(k)$. This means we can identify a kernel by its output object. On the other hand, an object can function as input for any number of kernels.*

**Remark 4** *This also means that we have a functional language of sorts. Strategies for obviating the need for a garbage collector are discussed in* IMP-04.

We will often talk about the distributions of a kernel:

**Function** $\alpha, \gamma$: distributions of a kernel

$$K = \langle f, x, y \rangle \quad \text{then} \quad \begin{array}{l} \alpha(K) = \text{distr}(x) \\ \gamma(K) = \text{distr}(y) \end{array}$$

We can now start talking about the distribution of the work of a kernel. We define

**Function** β: the β distribution of a kernel

$$K = \langle f, x, y \rangle \quad \text{then} \quad \beta(K) = \sigma_f\big(\gamma(K)\big)$$

In general, $x$ is unlikely to be distributed as $x(\beta)$: The α-distribution describes the actual storage, while β describes how the algorithm uses $x$.

In *IMP-01* we argued for the importance of $\beta = \sigma_f(\gamma)$ as describing a formalization of the halo concept: $\beta(p)$ consists of all the indices that processor $p$ needs to perform a local computation of kernel in question.

### 2.4 Tasks

A task is a kernel, executed on a specific processor:

**Datatype** Task: A kernel executed on a specific processor

$$\text{Task} \equiv \text{Kernel} \times \text{Proc}$$

giving us the obvious shorthand

$$t \in k \equiv t = \langle k, p \rangle \quad \text{for some } p.$$

We define a bunch of shorthands, relating tasks and kernels and their objects and distributions

$$\begin{cases} \text{if } t = \langle k, p \rangle \text{ then } \alpha(t) = \alpha(k) \text{ and } \beta, \gamma \text{ similar.} \\ \text{Tasks}(k) & \text{all } t \text{ such that } t = \langle k, p \rangle \text{ for some } p \\ \text{Proc}(t) & p, \text{ where } t = \langle k, p \rangle \\ \text{Kernel}(x) & \text{the unique } k \text{ such that } x = \text{Out}(k) \end{cases}$$

We can define a predecessor relationship on kernels, bearing in mind the uniqueness of kernel output:

**Function** Pred: predecessors of a kernel

$$\text{Pred}(k) = \{k' \colon \text{Out}(k') \subset \text{In}(k)\}$$

This allows us to define a predecessor relationship on tasks:

**Function** Pred: predecessors of a task

$$\text{Pred}\colon \text{Task} \to 2^{\text{Task}}$$
$$\begin{cases} \text{Let } t = \langle k,p\rangle,\ t' = \langle k',p'\rangle \\ \text{let } \alpha = \alpha(k) \text{ and } \beta = \beta(k) \\ \text{then } t' \in \text{Pred}(t) \equiv k' \in \text{Pred}(k) \wedge \alpha(p') \cap \beta(p) \neq \emptyset \end{cases}$$

### 2.5 Messages

The most specific element of the IMP model is a message.

**Datatype** Message: Message between two tasks

$$\text{Message} \equiv \text{Task} \times \text{Task} \times \text{Ind} \times \text{DistrArray}$$

Note that this definition of a message is more semantically endowed than MPI messages: both processors here exchange an index set, rather than arbitrary data.

Let's spend a minute to take this definition apart.

We start by writing a specific message as

$$\big\langle \langle k',p'\rangle, \langle k,p\rangle, S, x\big\rangle$$

For this to make sense we need a number of conditions. First of all

$$x \in \text{Out}(k') \quad \text{and} \quad x \in \text{In}(k).$$

Since we associate an object uniquely with the kernel that produces it, we write

$$k' = \text{Kernel}(x), \quad x \in \text{In}(k).$$

Next, the index set $S$ needs to be produced by processor $p'$:

$$S \subset \gamma_{k'}(p') = \alpha_k(p')$$

and used by $p$:

$$S \subset \beta_k(p)$$

so that in fact, for kernel $k$,

$$S = \alpha(p') \cap \beta(p)$$

as seen many times above.

Thus, we could in fact define

**Datatype** Message: Message between two tasks

$$\text{Message} \equiv \text{Task} \times \text{Task} \times \text{Obj}$$

and define

**Function** Content: content of message

$$\text{Content}(m) = \alpha(p') \cap \beta(p) \text{ where } p, p', \alpha, \beta \text{ as above}$$

Using the equivalence between kernels and object we can also define the messages of a kernel:

**Function** message$(K,q,p)$: message between two processors wrt a kernel

$$\text{message}(K,q,p) \equiv \text{message}_{\alpha(K),\beta(K)}(q,p)$$

This means that we have proved the following:

**Lemma 1** *In a given kernel, there will be only one message between a pair of tasks.*

### 2.5.1 Dependency structure

We might ask what the minimal structure is that can produce the messages of a kernel.

**Datatype** Dependency: a dependency of a kernel

$$\text{Dependency} = \text{Kernel} \times \text{DistrArray} \times \text{Signature}$$

To see how this works, we show that a dependency $\delta = \langle k, x, \sigma \rangle$ induces a function

$$\text{Dependency}\colon \text{Proc} \to 2^{\text{Proc}}$$

as follows

let $\gamma = \gamma(k)$ and let $\beta = \sigma\gamma$
$\delta\colon p \mapsto \{p'\colon \alpha(p') \cap \beta(p) \neq \emptyset\}$

A dependency also induces a function that produces messages:

$$\text{Dependency}\colon \text{Proc} \times \text{Proc} \to \text{Message}$$

as follows. Let $\delta = \langle k, x, \sigma \rangle$, then, if $p' \in \delta(p)$

$$\delta(p', p) = \big\langle \langle k', p' \rangle, \langle k, p \rangle, S, x \big\rangle$$

where

$$k' = \text{Kern}(x), \quad \alpha = \gamma(k'), \quad \beta = \sigma\gamma(k), S = \alpha(p') \cap \beta(p)$$

### 2.5.2 I don't think this theorem is important

The totality of all messages constitute the $\beta$ distribution:

**Theorem 1** *Suppose distributions $\alpha, \beta$ satisfy*

$$\forall_p\colon \beta(p) \supset \cup_q \alpha(q).$$

*Then for any $p$:*

$$\beta(p) = \cup_{q \in \text{pred}_{\alpha,\beta}(p)} \text{message}_{\alpha,\beta}(q, p)$$

*Proof.* *By the condition on this theorem we have*

$$\beta(p) = \cup_q \alpha q \cap \beta(p). \tag{1}$$

*Rewrite $\alpha(q) \cap \beta(p)$ as $\text{message}_{\alpha,\beta}(q, p)$, and use lemma* **??** *to observe that $q$-terms in (1) are empty if $q \notin \text{pred}(p)$.*

We now introduce the mechanism that turns $x(\alpha)$ into $x(\beta)$, where in practical terms $\alpha$ describes the storage scheme for $x$, while $\beta$ describes the rearrangement that is needed for a parallel computation, as described above.

Above we introduced the function $\text{pred}_{\alpha,\beta}$ that maps a processor to its predecessors wrt two distributions. With this we define

**Function** $\text{transform}_{\alpha,\beta}(p)$: transformation between distributions

$$\text{transform}\colon \text{Distr} \times \text{Distr} \to \text{Distr}$$
$$\text{transform}_{\alpha,\beta}(p) = \cup_{q \in \text{pred}_{\alpha,\beta}(p)} \alpha(q)$$

Composing this with the Array type, we get

**Lemma 2** *If $x$ is a distributed object and $\alpha,\beta$ are distributions, then*

$$x(\beta) = Tx(\alpha)$$

*where* $T = \text{transform}_{\alpha,\beta} = \text{depend}_{\alpha,\beta}$.

*Proof. Intermediate result:*

$$x(\beta) = x\big(\alpha(\alpha^{-1}\beta)\big)$$

*and we write the rhs as $x\big(\alpha(T)\big)$, which by notational convenience we write as $Tx(\alpha)$.*

We just used an auxiliary lemma:

**Lemma 3** *Let $\alpha$ be a distribution and let $I$ be a set of integers. Generalizing the definition for $\alpha^{-1}\beta$ (**??**) to*

$$\alpha^{-1}(I) \quad := \quad \{q\colon \alpha q \cap I \neq \emptyset\}$$

*we find:*

- *let $Q = \alpha^{-1}(I)$, then $\alpha(Q) \supset I$*
- *so, for a second distribution $\beta$, $\alpha\big(\alpha^{-1}\beta(p)\big) \supset \beta(p)$. (Proof: choose $I = \beta(p)$ and $Q = \alpha^{-1}(I) = \alpha^{-1}\beta(p)$.)*

### 2.5.3 Buffers

We can consider an index set $S \in \text{Ind}$ as a mapping:

$$\text{if } S = \{s_0,\ldots,s_{n-1}\}, \text{ then } S = N \to \text{Ind defined as } S(i) = s_i$$

For index sets $S, S' \in \text{Ind}$, if $S' \subset S$, we define

**Function** Embed: embedding an index set in a superset

$$\text{Embed}\colon \text{Ind} \times \text{Ind} \to (N \to N)$$
$$\text{Embed}(S',S)\colon i \mapsto i' \text{ such that } S(i') = S'(i)$$

Now we can define receive and send buffers of a message:

$$m \in \text{Message}$$
$$S = \text{Ind}(m)$$
$$\rho = \text{Embed}\big(S,\beta(p)\big)$$
$$\sigma = \text{Embed}\big(S,\alpha(p')\big)$$

### 2.6 Dataflow

The transformation $T$ has the following practical interpretation:

- if $q \in T(p)$, part of $x$ that is stored on $q$ is needed on $p$ for the local computation;
- therefore, in message passing terms, a message will be sent from $q$ to $p$;
- or in terms of a task graph, as is often used in shared memory programming, the task on $p$ that computes $y(\gamma(p))$ has to wait for whatever task produced $x(\alpha(q))$.

If we introduce yet another notation and write $Tx$ for $x(\alpha)T$, we get the description for a parallel computation

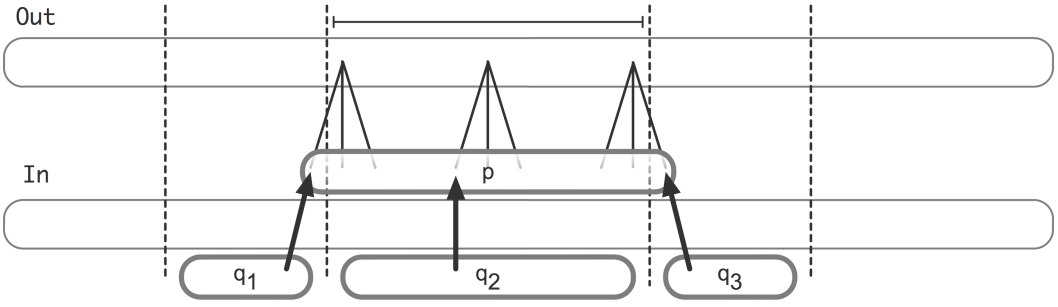
**Theorem 2**

$$y = f(Tx) \equiv \begin{cases} y \text{ is distributed as } y(\gamma) \\ x \text{ is distributed as } x(\alpha) \\ \beta = I_f \gamma \\ T = \alpha^{-1}\beta \end{cases} \tag{2}$$

which says that we can proceed with a local computation of $y = fx$ after the data dependencies described by $T$ have been fulfilled.

We see that the transformation from $\alpha$ to $\beta$-distribution gives us a *dataflow* interpretation of the algorithm by investigating the relation (**??**).

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:




Formally, 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.