TACC Technical Report IMP-01

IMP Distribution Theory

Victor Eijkhout[*]

February 15, 2018

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

Abstract

The Integrative Model for Parallelism (IMP) is a new approach to parallel programming based on a generalized notion of data parallelism. The fundamental notion of IMP is that of a distribution: a description of the correspondence between processors and data. Here we give the basic theory and some examples.

1 Motivation

The concept of distribution of an indexed object comes up naturally in parallel computing through the *owner computes* rule: the distribution is the mapping from the processor to the set of indices where it computes the object. If we now consider the data parallel computation of one object from another, we have an input distribution and an output distribution. These are typically specified by the user program.

User distributions can be simple, such as blocked, cyclic, or block-cyclic, in one or more dimensions. They can also be irregular, such as they appear in Finite Element Method (FEM) calculations. In addition to these traditional distributions, our theory will cover overlapping and even fully redundant distributions. We will also cover partial distributions, for instance as a way to handle adaptive mesh refinement through our data parallel framework.

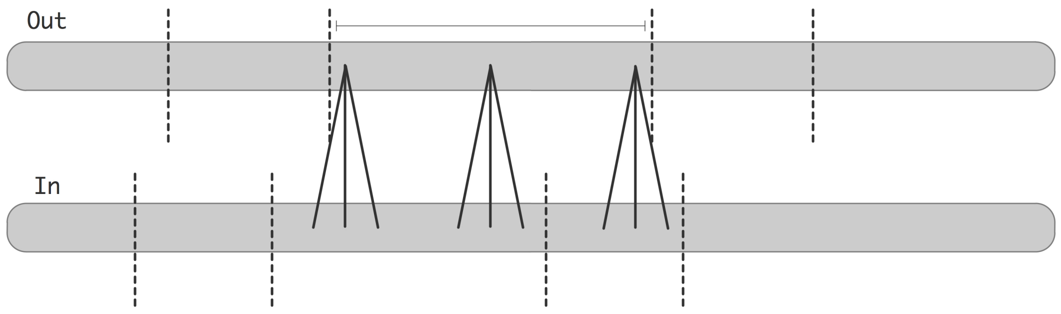
2 Motivating examples

We consider a simple data parallel example, and show how it leads to the basic distribution concepts of Integrative Model for Parallelism (IMP): the three-point operation

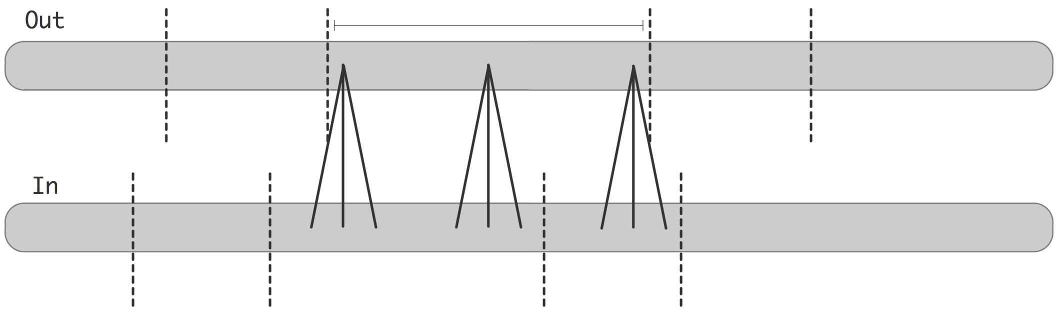
$$\forall_i\colon y_i = f(x_i, x_{i-1}, x_{i+1})$$

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., [6] and the polyhedral model, e.g., [1]. However, we claim far greater generality for our model.) In the case of vectors stored distributed over the processors by contiguous blocks, we see that the operation requires inter-process communication:

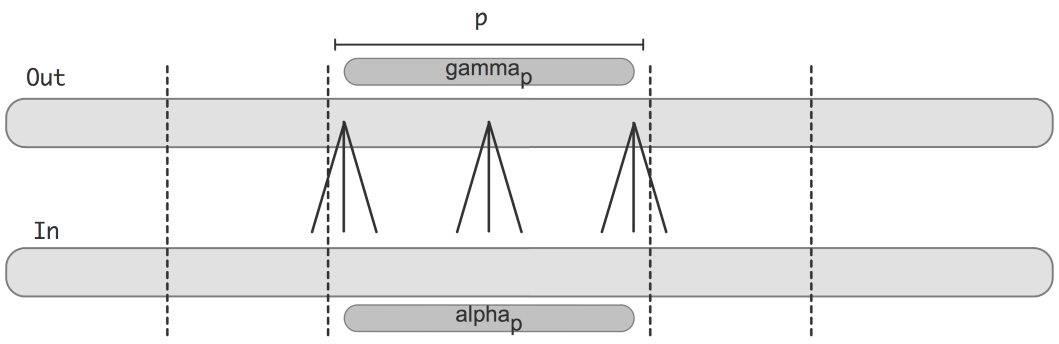


The distribution indicated by vertical dotted lines we call the α-distribution for the input, and the γ-distribution for the output. These distributions are mathematically given as an assignment from processors to sets of indices:

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

The traditional concept of distributions in parallel programming systems is that of an assignment of data indices to a processor, 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. (The utility of this for redundant computing is obvious. However, it will also seen to be crucial for our general framework.)

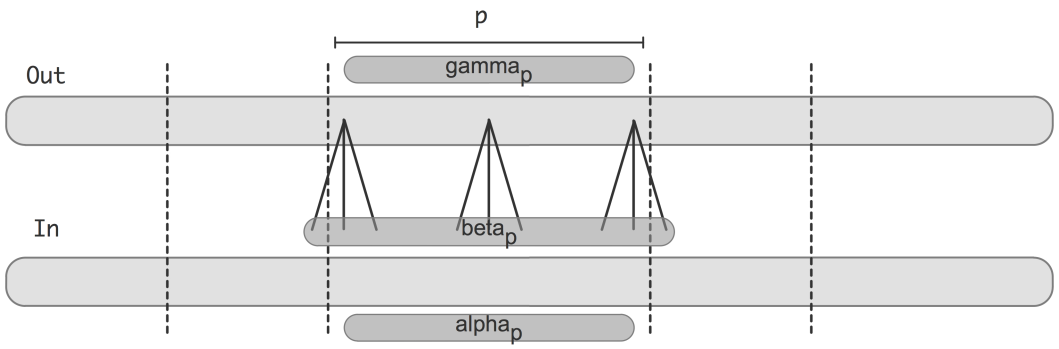
For purposes of exposition we will now equate the input α-distribution and the output γ-distribution, although that will not be necessary in general.



This picture shows how, for the three-point operation, 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 $p-1$. This gives rise to what we call the β-distribution:

$\beta(p)$ is the set of indices that processor p needs to compute the indices in $\gamma(p)$.

The next illustration depicts the different distributions for one particular process:



Observe that the β-distribution, unlike the α and γ ones, is not disjoint: certain elements live on more than one processing element. It is also, unlike the α and γ distributions, not specified by the programmer: it is derived from the γ-distribution by applying the shift operations of the stencil. That is,

The β-distribution brings together properties of the algorithm and of the data distribution.

We will formalize this derivation below.

2.1 Deriving the dataflow formulation

This gives us all the ingredients for reasoning about parallelism. Defining a *kernel* as a mapping from one distributed data set to another, and a *task* as a kernel on one particular process(or), all data dependence of a task 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_i, and this set of predecessors can be derived from the α, β distributions: q_i is a predecessor if

$$\alpha(q_i) \cap \beta(p) \neq \emptyset.$$

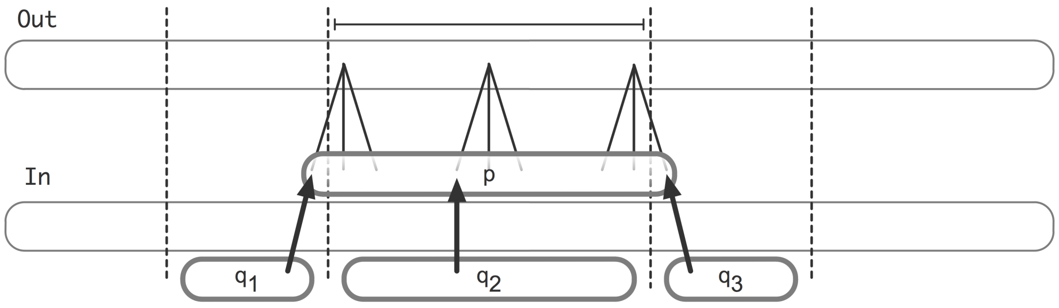


Figure 1 illustrates this: the left DAG is the sequential program of a heat equation evolution; the right DAG is the dataflow representation derived when this sequential program is run on six processors.

2.2 Parallelism-mode independence through dataflow

The term 'dataflow' comes with lots of baggage. However, in the abstract, a dataflow formulation does not imply any specific hardware model. In fact, we argue that by deriving (not programming!) a dataflow formulation of algorithms we can achieve mode-independent parallel programming.

First of all: in message passing, these dataflow 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.)

Secondly, in shared memory we don't interpret distributions with a physical distribution, but rather with a logical one: we identify the α-distribution on the input with tasks that produce this input. The β-distribution then describes what input-producing tasks a task p is dependent on. The dataflow description then becomes a task dependency graph.

2.3 Non-trivial example

Consider one level of multigrid coarsening, with 6 points divided over 4 processes. Since the coarse level has fewer points than processes, we need to have some duplication. (The alternative, of having inactive processes, is harder to code, and in fact less efficient.)

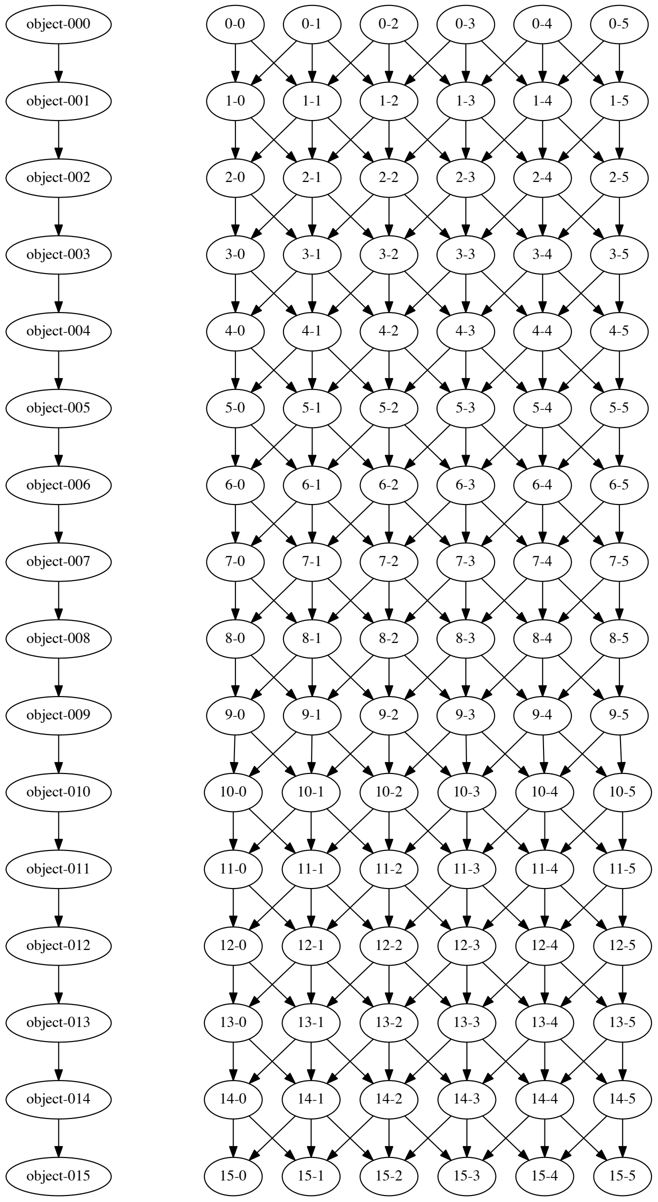
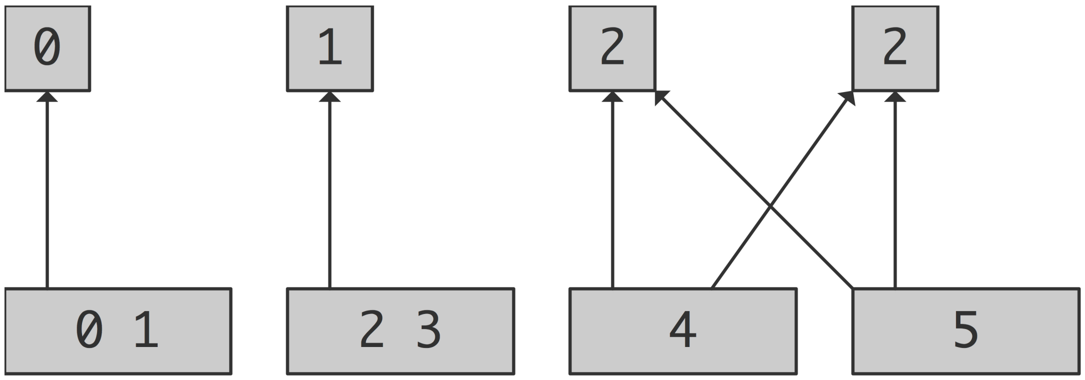


Figure 1: Kernel (left) and task relations (right) for the one-dimensional heat equation, executing 15 steps on 6 processors.



The first two processes achieve their coarsening by a local operation, but the second pair needs to exchange messages. As we will see, this non-uniform messaging is easily achieved in IMP.

2.4 Programming the model

In our motivating example we showed how the concept of 'β-distribution' arises, and how from it we can derive messages and task dependencies.

We will now argue the practicality of this concept. From the motivation example, it is clear that the β-distribution generalizes concepts such as the 'halo region' in distributed stencil calculations, but its applicability extends to all of (scientific) parallel computing. We will touch on this in section **??**.

It remains to be argued that the β distribution can actually be used as the basis for a software system. To show this, we associate with the function f that we are computing an expression of the algorithm (not the parallel!) data dependencies, called the 'signature function', denoted σ_f. For instance for the computation of $y_i = f(x_i, x_{i-1}, x_{i+1})$, the signature function is

$$\sigma_f(i) = \{i, i-1, i+1\}.$$

With this, we state (without proof; for which see section **??** and [4]) that

$$\beta = \sigma_f(\gamma).$$

It follows that, if the programmer can specify the data dependencies of the algorithm, a compiler/runtime system can derive the β distribution, and from it, task dependencies and messages for parallel execution.

Specifying the signature function is quite feasible, but the precise implementation depends on the context. For instance, for regular applications we can adopt a syntax similar to stencil compilers such as the Pochoir compiler [6]. For sparse matrix applications the signature function is isomorphic to the adjacency graph; for collective operations, $\beta = \gamma$ often holds; et cetera.

3 Distributions

In this section we formalize the concepts we motivated by way of example in section 2.

3.1 Introductory considerations

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 algorithms.

The communication involved in constructing this second distribution can now be considered as a formally defined transformation object. We will do this in section 4. 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 **??**.

3.2 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[2]:

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

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

3.3 Examples of distributions

There are some common distributions, in use in languages such as UPC or HPF.

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 $\{0,\ldots,N-1\}$.

With $\beta = N/P$ (assuming for simplicity's sake that N is evenly divisible by P) we define the block distribution

$$b \equiv p \mapsto [p\beta, \ldots (p+1)\beta - 1], \tag{2}$$

and the cyclic distribution

$$c \equiv p \mapsto \{i\colon \quad \mod(i, \beta) = p\}.$$

Our definition does not require distributions to be a disjoint assignment. A common non-disjoint assignment is the redundant replication, notated

$$* \equiv p \mapsto N,$$

where each processor has the full index set.

Another useful distribution is the following. Let n be an integer, then we define

$$n \equiv p \mapsto \{n\}.$$

In other words, n denotes the distribution that puts the index n on every processor.

Remark. In the IMP model we decouple a distribution from how it is constructed. For instance, the previous two non-disjoint distributions can appear as the end result of an allgather and a broadcast respectively, but they do not need to be. A replicated vector can have been constructed that way, and a replicated single element can come about as a restriction of a replicated full vector. Actual data movement will be seen to be result of matching up *two* distributions.

3.4 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][4]:

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

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:

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

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.

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

$$\begin{aligned}A(u,*) &\equiv p \mapsto A[u(p),*]\\ A(*,u) &\equiv p \mapsto A[*,u(p)]\end{aligned}$$

3.5 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 $\mathbb{N} \to \mathbb{N}$ 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'. (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

$$y(u) \leftarrow 2x(u) - x(u \ll 1) - x(u \gg 1). \tag{3}$$

3.6 Partial distributions

One problem with the basic definition is that it requires a statement about the content of every processor. To be able to talk about, for instance, a reduction onto a single processor, multigrid type interpolation onto a subset of the processors, or adaptive refinement, we need a partially defined distribution.

Let $S \subset P$, then we notate a distribution u defined on S by $S : u$. Its definition is given as

$$S : u \equiv p \mapsto \begin{cases} u(p) & \text{if } p \in S \\ \text{undefined} & \text{otherwise} \end{cases}. \tag{4}$$

The original definition is now a shorthand for the case $S = P$.

Using partial distributions, we can describe a vector that is only defined on certain processes:

$$x(S : u) \equiv S \ni p \mapsto x[u(p)]$$

In the limit case, $x(\{n\} : u)$ describes a distributed vector that is defined on just one processor. This situation arises for instance as the result of a reduction with processor n as root.

See *IMP-14* [3] for an application of partial distributions.

3.7 Trivial product distributions

If we have multi-dimensional arrays with a high aspect ratio, we can decide to keep the 'shallow' dimension entirely undistributed. This is for instance done in an *extruded mesh*, a typically unstructured mesh on a large domain, with one dimension added to each finite element, and an identical number of elements stacked on each element of the unstructured mesh.

In this case we construct a distribution one dimension lower, and add a single parameter for the orthogonal dimension. This simplifies the logistics of the code, bringing down the amount of indexing information that is needed to locate an element.

There is also an important performance consideration. If we keep the extruded dimension as innermost and contiguous, this means that all halo elements, both on source and target, are contiguous, and therefore efficient in memory bandwidth.

3.8 Non-trivial product distributions

We can make a distribution of a Cartesian index set over a Cartesian processor grid. This requires us to use processor numbers that are no longer linearly ordered indexes. This is no essential generalization, since we never used the linear ordering on the processor numbers.

3.9 Two-dimensional distributions

Let us now extend the distribution notation to two-dimensional distributions, considering the scalable matrix-vector product. Assume a $P \times Q$ processor grid, and let the matrix be of size $N = PQ$. In practice the matrix will be larger, so we will consider it to be divided in appropriately sized blocks and N is the block dimension.

x_0				x_3				x_6				x_9			
a_{00}	a_{01}	a_{02}	y_0	a_{03}	a_{04}	a_{05}		a_{06}	a_{07}	a_{08}		a_{09}	$a_{0,10}$	$a_{0,11}$	
a_{10}	a_{11}	a_{12}		a_{13}	a_{14}	a_{15}	y_1	a_{16}	a_{17}	a_{18}		a_{19}	$a_{1,10}$	$a_{1,11}$	
a_{20}	a_{21}	a_{22}		a_{23}	a_{24}	a_{25}		a_{26}	a_{27}	a_{28}	y_2	a_{29}	$a_{2,10}$	$a_{2,11}$	
a_{30}	a_{31}	a_{32}		a_{33}	a_{34}	a_{35}		a_{37}	a_{37}	a_{38}		a_{39}	$a_{3,10}$	$a_{3,11}$	y_3
	x_1				x_4				x_7				x_{10}		
a_{40}	a_{41}	a_{42}	y_4	a_{43}	a_{44}	a_{45}		a_{46}	a_{47}	a_{48}		a_{49}	$a_{4,10}$	$a_{4,11}$	
a_{50}	a_{51}	a_{52}		a_{53}	a_{54}	a_{55}	y_5	a_{56}	a_{57}	a_{58}		a_{59}	$a_{5,10}$	$a_{5,11}$	
a_{60}	a_{61}	a_{62}		a_{63}	a_{64}	a_{65}		a_{66}	a_{67}	a_{68}	y_6	a_{69}	$a_{6,10}$	$a_{6,11}$	
a_{70}	a_{71}	a_{72}		a_{73}	a_{74}	a_{75}		a_{77}	a_{77}	a_{78}		a_{79}	$a_{7,10}$	$a_{7,11}$	y_7
		x_2				x_5				x_8				x_{11}	
a_{80}	a_{81}	a_{82}	y_8	a_{83}	a_{84}	a_{85}		a_{86}	a_{87}	a_{88}		a_{89}	$a_{8,10}$	$a_{8,11}$	
a_{90}	a_{91}	a_{92}		a_{93}	a_{94}	a_{95}	y_9	a_{96}	a_{97}	a_{98}		a_{99}	$a_{9,10}$	$a_{9,11}$	
$a_{10,0}$	$a_{10,1}$	$a_{10,2}$		$a_{10,3}$	$a_{10,4}$	$a_{10,5}$		$a_{10,6}$	$a_{10,7}$	$a_{10,8}$	y_{10}	$a_{10,9}$	$a_{10,10}$	$a_{10,11}$	
$a_{11,0}$	$a_{11,1}$	$a_{11,2}$		$a_{11,3}$	$a_{11,4}$	$a_{11,5}$		$a_{11,7}$	$a_{11,7}$	$a_{11,8}$		$a_{11,9}$	$a_{11,10}$	$a_{11,11}$	y_{11}

Figure 2: Distributions for the matrix-vector product on a $P \times Q$ two-dimensional processor grid with $P = 3$, $Q = 4$.

Next, let there be two functions v, w that injectively and surjectively map processor coordinates P, Q to a linear enumeration, for instance

$$\begin{cases} v(p,q) = p + qP & \text{column-major enumeration} \\ w(p,q) = q + pQ & \text{row-major enumeration.} \end{cases}$$

These functions describe the distribution of the vectors; in our example figure 2 the vectors x, y are distributed as

$$x(v) \equiv p,q \mapsto x(v(p,q)), \quad y(w) \equiv p,q \mapsto y(w(p,q)).$$

This is the same notation for processor to index mapping as we used before for one-dimensionally indexed objects, now extended to two-dimensional indices.

The approach of deriving matrix distribution by starting with vector distributions is also taken in [2]. On the other hand, [5] uses matrix distributions exclusively. The latter approach is less elegant, as it fails to express the relationship between various distributions analytically.

We introduce a notation:

$$\begin{cases} v(:\,,q) \equiv [v(0,q), \ldots, v(P-1,q)] & \text{all indices in a processor column} \\ w(p,:\,) \equiv [w(p,0), \ldots, w(p,Q-1)] & \text{all indices in a processor row} \end{cases}$$

First of all, we observe that transforming a distributed vector

$$(p,q) \mapsto x(v(p,q)) \quad \text{to} \quad (p,q) \mapsto x(v(:,q))$$

involves an allgather over p for each value of q; that is, an independent allgather in each processor column.

Next, we use this notation to define the matrix distribution as

$$p,q \mapsto A\big(w(p,:),v(:,q)\big). \tag{5}$$

where for sets $\tilde{w},\tilde{v}$ we interpret

$$A(\tilde{w},\tilde{v}) = \{A(i,j)\colon i\in\tilde{w},\ j\in\tilde{v}\}.$$

Note that this is a proper assignment of the whole matrix, since

$$\cup_p w(p,:) = N, \quad \cup_q v(:,q) = N.$$

3.9.1 Cyclic distributions

The above story used a block distribution, but that fact was not actually used after the initial definition. The whole story goes through regardless the definitions of v,w.

Thus, we easily cover the two-dimensionally block cyclical case by redefining the v,w functions. Let π be a permutation of P and γ of Q, then we define

$$v(p,q) = \pi(p)+\gamma(q)P, \quad w(p,q) = \gamma(q)+\pi(p)Q.$$

This induces the two-D block cyclic distribution known from popular dense linear algebra software.

4 Transformations between distributions

Distributions by themselves are a useful tool, but their most important application is describing the communication involved in converting between one distribution and another. As already remarked above, we consider the result of an all-gather or the construction of a halo region as a different distribution of a data set, rather than as 'local data' as is mostly done in distributed memory programming. In this section we formalize this transformation.

4.1 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

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

as

$$u^{-1}v(p) \equiv \{q\colon u(q) \cap v(p) \neq \emptyset\}. \tag{6}$$

We can even extend this definition to partial distributions:

$$\begin{array}{rcl} q \in (S:u)^{-1}v(p) & \equiv & (S:u)(q) \cap v(p) \neq \emptyset \\ & \Rightarrow & q \in S \wedge u(q) \cap v(p) \neq \emptyset \end{array}$$

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 α-distribution and the result as the β-distribution.

4.2 Example transformations

It is hard to say much about transformations in general, but some special cases are easily described.

The theory of collectives is discussed in report IMP-15.

4.2.1 Gather

Attempt1 If u is an arbitrary α-distribution and the β-distribution is $*$ we have described an all-gather. With a β distribution of $S\colon *$ we have a gather that is replicated on the set S. If S has only a single element, we have a traditional gather onto a root process.

Attempt2 The gather operation is a bit subtle because of how simple it is.

- With just a single element per processor, we have $N = P$.
- And $\alpha\colon\ p \mapsto p$.
- Gathering everything together means $\gamma\colon\ p \mapsto P$.

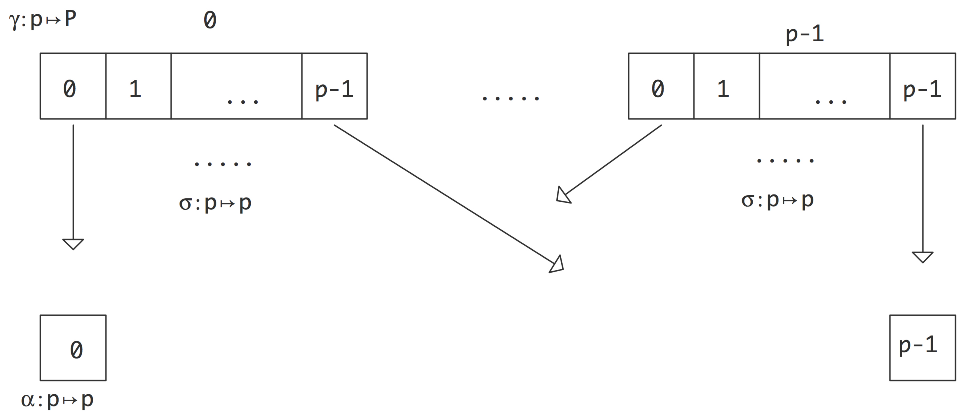


Figure 3: Illustration of the distributions involved in a gather

- The index function is $\sigma_{\text{collective}}: p \mapsto p$, saying that the gathered array just contains everything collected together.

This makes $\beta = \sigma\gamma = \gamma$; see figure 3.

This is for instance seen in how the scaling kernel is implemented:

```
%% imp_ops.h
scale_kernel( std::shared_ptr<object> a,std::shared_ptr<object> in,std::shared_ptr<object> out ) : scal
  set_name(fmt::format("scaleby object{}",get_out_object()->get_object_number()));
  a->require_type_replicated();
  add_in_object(a);
  dependency *d = last_dependency(); d->set_name("wait for scalar");
  d->set_explicit_beta_distribution( a.get() );
};
```

4.2.2 Reduction

It is a failing of the normal form that communication and computation are strictly decoupled. That means we cannot describe a reduction directly: instead we need to do a gather to a replicated array, followed by a local reduction.

4.2.3 Broadcast

Conversely, we can describe broadcasts. With an α distributed object that is defined on a single processor

$$x(\{p\}:n) \equiv \text{if } p = p' \text{ then } p' \mapsto x[n(p')] = x[n].$$

and a β distributed object $x(n)$, the $\alpha^{-1}\beta$ transformation corresponds to a broadcast of $x[n]$ with processor p as root.

To describe a scatter we write

$$\alpha = x(S\colon u) \text{ and } \beta = x(v)$$

The condition that all v elements are in S is

$$\cup_{p\in S} u(p) = \cup_{p\in P} v(p).$$

Of course, this simplifies if $S = \{p\}$.

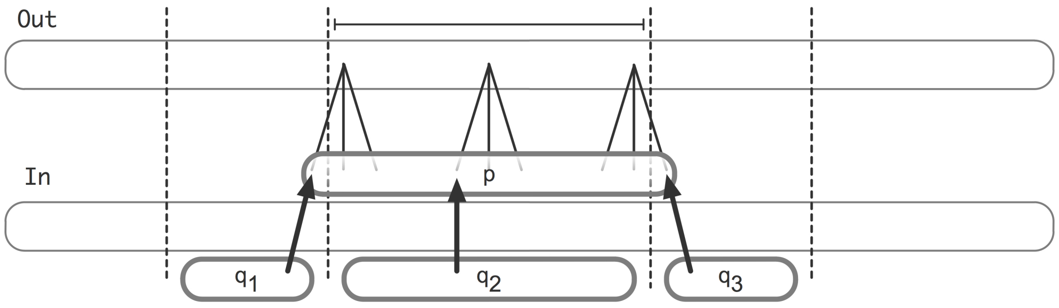
4.2.4 Local reductions

Under the normal form, a kernel consists of a communication stage and a local computation stage. However, the local computation may itself be considered a kernel, just one with no communication. This means that $\beta = \alpha$, while the γ distribution can be anything, as long as it makes the computation local.

4.3 Dataflow

Now we see that the transformation from α to β-distribution gives us a *dataflow* interpretation of the algorithm by investigating the relation (6).

Graphically, in the threepoint example, we see that in order to build the β-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.

The issue of dataflow between tasks is explored in more detail in *IMP-05*.

4.4 Simple results

If we denote $T \equiv \alpha^{-1}\beta\colon P \to 2^Q$, we can also define a $F\colon Q \to 2^P$ by $F \equiv \beta^{-1}\alpha$. While $T(p)$ gives for a process p the processes q it depends on, F describes the converse: $F(q)$ contains the processes p that depend on q.

There are some simple relations between T,F. For instance, if we define a predicate

$$\mathrm{collective}(T) \quad := \quad \forall_p\colon T(p) = Q,$$

we find the following lemma that states that, if every process p receives from every q, then every q sends to every p.

Lemma 1 *T is collective iff F is collective.*

Proof.

$$\begin{aligned}\mathrm{collective}(T) &\Leftrightarrow \forall_p\colon \alpha^{-1}\beta(p) = Q\\ &\Leftrightarrow \forall_{p,q}\colon q \in \alpha^{-1}\beta(p) \Leftrightarrow \forall_{p,q}\colon \alpha(q) \cap \beta p \neq \emptyset \Leftrightarrow \forall_{p,q}\colon p \in \beta^{-1}\alpha(q)\\ &\Leftrightarrow \forall_q\colon \beta^{-1}\alpha(q) = P \Leftrightarrow \mathrm{collective}(F)\end{aligned}$$

For a slightly less trivial statement, we derive that, if there is a lower bound on the number of receives of a process, there is also a lower bound on the number of sends.

Lemma 2 *Let k be such that, for all p, $|T(p)| \geq k$. Then for all q, $|F(q)| \geq |P|k/|Q|$.*

Proof. Elementary application of the pigeon-hole principle.

4.5 Parallel assignment

With transformations between distributions we can attach meaning to parallel assignments. As long we talk about one index set we can accomodate different sets of processors:

$$\begin{aligned}&y(v) \leftarrow x(u)\\ &u\colon P_1 \to N\\ &v\colon P_2 \to N\end{aligned}$$

We keep the definition

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

to interpret the assignment as

$$\forall_{p\in P_2}\colon y[v(p)] \leftarrow \oplus_{q\in u^{-1}v(p)} x[u(q)\cap v(p)].$$

where the $\oplus$ operator stands for concatenation of the subsequences.

If the rhs distribution is a transformation of the left

$$y(u) \leftarrow x(f(u))$$

we interpret this as

$$\forall_p\colon y[u(p)] \leftarrow \oplus_{q\in(fu)^{-1}u(p)} x[(fu)(q)\cap u(p)].$$

with the obvious definition

$$q\in (fu)^{-1}u(p) \equiv (fu)(q)\cap u(p)\neq \emptyset$$

For instance if f is a right shift, and u is the distribution

$$u(p) = [n_p\colon n_{p+1}$$

that makes

$$(fu)(q) = [n_1+1\colon n_{q+1}+1]$$

so the intersection is nonzero for

$$\begin{cases} q=p & [n_p+1\colon n_{p+1}] \\ q=p+1 & [n_{p+1}\colon n_{p+1}+1] \end{cases}$$

5 Signature function

We can now start talking about the distribution of the work of a kernel. Informally, we let the distribution of work be induced by the distribution of the output: we say that processor p computes $y(i)$ for all $i\in\gamma(p)$. (We abbreviate this by saying that we 'compute $y(\gamma)$'.)

To compute $f(i)$ on a certain processor, that processor needs $x\bigl(\sigma_f(i)\bigr)$, so

$$\text{the computation of } y(\gamma(p)) \text{ needs } x\bigl(\sigma_f(\gamma(p))\bigr). \tag{7}$$

Here is the crucial part of the story: we call a parallel computation between distributed objects x,y a 'local computation' if, given an element y_i that is computed on processor p, all its required inputs x_j are also present on processor p.

Formally:

Definition 1 *The computation of a kernel $K = \langle f, x, y\rangle$ is local if*

$$\sigma_f \gamma = \alpha$$

where $\alpha = \mathrm{distr}(x)$ and $\gamma = \mathrm{distr}(y)$.

We now have all the machinery for describing a local computation. We translate (7) as the following theorem:

Theorem 1 *The parallel computation*

$$y(\gamma) = f(x(\beta))$$

is local if

$$\beta \supset \sigma_f(\gamma).$$

Proof. First of all note that $\sigma_f(\gamma)$ is a distribution:

$$p \mapsto \sigma_f(\gamma(p)).$$

Now observe that processor p computes y in $\gamma(p)$, which requires $\sigma_f(\gamma(p))$ as total input. Thus, if $\beta \supset \sigma_f(\gamma)$, the input vector is correctly distributed for a local computation.

This states that a computation is local if the input has a suitable distribution.

We have some elementary facts, such as:

Lemma 3 *If γ is a replicated distribution, that is, $\gamma \equiv \gamma_0$ for some index set γ_0, then β is replicated too.*

Proof. $\beta \equiv \sigma_f(\gamma_0)$.

(There appears to be a footnote to this statement. For instance, if we scale a block distributed vector by a constant, the constant will have a replicated distribution while the vector is blocked. Clearly we need to do this separately for each dependency. Maybe the crux is that explicitly set betas do not have to correspond in type?)

References

[1] 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.

[2] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.

[3] Victor Eijkhout. 1.5d all-pairs methods in the integrative model for parallelism (under construction). Technical Report IMP-14, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014.

[4] Victor Eijkhout. IMP distribution theory. Technical Report IMP-01, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014. (Included in source code repository.).

[5] 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.

[6] 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.