A mathematical formalization of data parallel operations

Victor Eijkhout*

February 7, 2016

Abstract

We give a mathematical formalization of 'generalized data parallel' operations, a concept that covers such common scientific kernels as matrix-vector multiplication, multi-grid coarsening, load distribution, and many more. We show that from a compact specification such computational aspects as MPI messages or task dependencies can be automatically derived.

1 Introduction

In this paper we give a rigorous formalization of several scientific computing concept that are commonly used, but rarely defined; specifically distributions, data parallel operations, and 'halo regions'. Taken together, these concepts allow a minimal specification of an algorithm by the programmer to be translated into the communication and synchronization constructs that are usually explicitly programmed.

Looking at it another way, we note that communication and synchronization in a parallel code stem from both algorithm and data distribution properties. The contribution of this work is then that we have found a separation of concerns that allows the programmer to specify them separately, while the resulting communication and synchronization is derived formally and therefore automatically.

We start with a motivating example in section 2, followed by a formal derivation in section 3. We conclude by discussing the practical ramifications of our work.

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

2 Motivating example

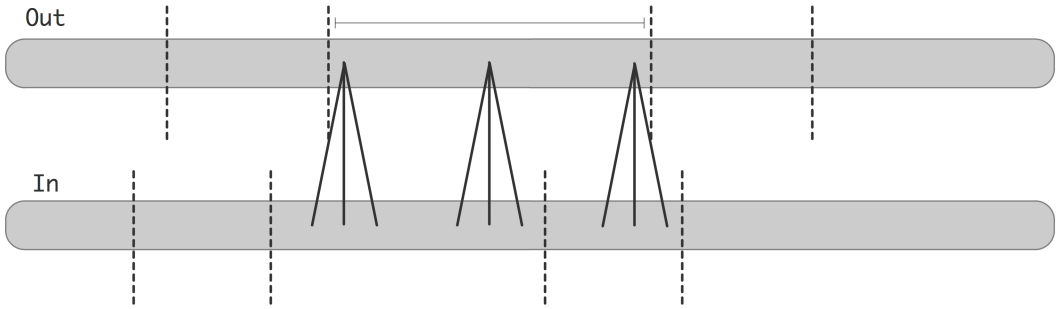
We consider a simple data parallel example, and show how it leads to the basic distribution concepts of Integrative Model for Parallelism (IMP): the three-point operation

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

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., [5] and the polyhedral model, e.g., [1]. However, we claim far greater generality for our model.) We illustrate this graphically by depicting the input and output vectors, stored distributed over the processors by contiguous blocks, and the three-point combining operation:

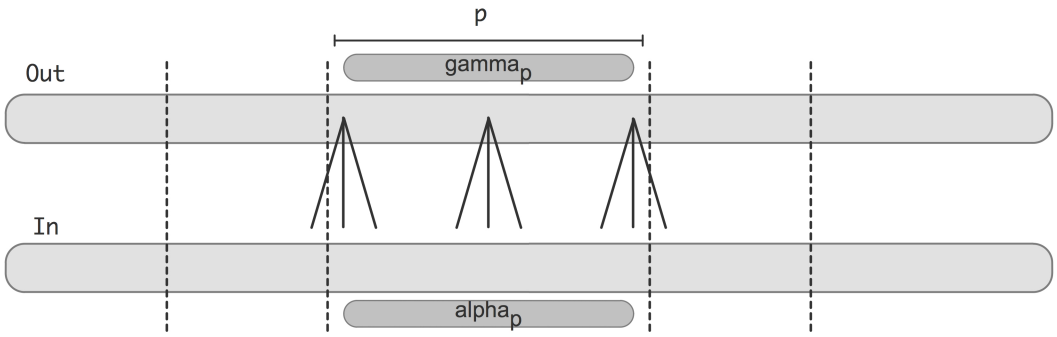


The distribution indicated by vertical dotted lines we call the α-distribution for the input, and the γ-distribution for the output. These distributions are mathematically given as an assignment from processors to sets of indices:

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

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.)

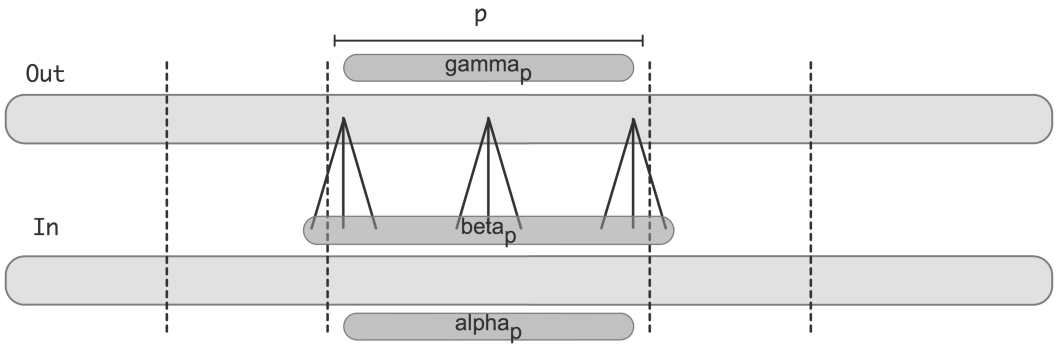
For purposes of exposition we will now equate the input α-distribution and the output γ-distribution, although that will not be necessary in general.



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:

> β(p) is the set of indices that processor p needs to compute the indices in γ(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,

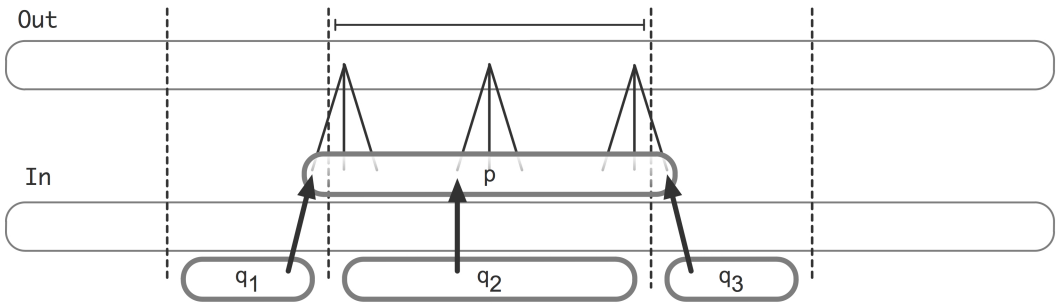
> The β-distribution brings together properties of the algorithm and of the data distribution.

We will formalize this derivation below.

2.1 Definition of parallel computing

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

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



In message passing, these dependences obviously corresponds to actual messages: for each process p, the processes q that have elements in $\beta(p)$ send data to p. (If $p = q$, of course at most a copy is called for.) Interestingly, this story has an interpretation in tasks on shared memory too. If we identify the α-distribution on the input with tasks that produce this input, then the β-distribution describes what input-producing tasks a task p is dependent on. In this case, the transformation from α to β-distribution gives rise to a *dataflow* formulation of the algorithm.

2.2 Programming the model

In our motivating example we showed how the concept of 'β-distribution' arises, and the role it plays combining properties of the data distributions and of the algorithm's data dependencies. This distribution generalizes concepts such as the 'halo region' in distributed stencil calculations, but its applicability extends to all of (scientific) parallel computing. For instance, for collectives we can define a β-distribution, which is seen to equal the γ-distribution.

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 3.3 and [2]) that

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

If follows, 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 [5]. For sparse matrix applications the signature function is isomorphic to the adjacency graph; for collective operations, $\beta = \gamma$ often holds; et cetera.

3 Formal definition

3.1 Data parallel computation

The Integrative Model for Parallelism (IMP) is a theory of data parallel functions. By this we mean functions where each element of a distributed output object is computed from one or more elements of one or more distributed input objects.

- Without loss of generality we limit ourselves to a single input object.
- Since all output elements can be computed independently of each other, we call this a 'data parallel' function. In our context this has no connotations of SIMD or synchronization; it merely expresses independence.

Formally, a data parallel computation is the use of a function with a single output to compute the elements of a distributed object:

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

where k is some integer.

Since we will mostly talk about indices rather than data, we define $\text{Ind} \equiv 2^N$ and we describe the structure of the data parallel computation through a 'signature function':

$$\text{Signature} \equiv N \rightarrow \text{Ind}.$$

In our motivating example, where we computed $y_i = f(x_{i-1}, x_i, x_{i+1})$, our signature function was

$$\sigma_f \equiv i \mapsto \{i-1, i, i+1\}.$$

- The signature function can be compactly rendered in cases of a stencil computation.
- In general it describeds the bi-partite graph of data dependencies. Thus, for sparse computations it is isomorphic to the sparse matrix, and can be specified as such.
- In certain cases, the signature function can be most compactly be rendered as a function recipe. For instance, for 1D multigrid restriction it would be given as $\sigma(i) = \{2i, 2i+1\}$.
- For collectives such as an 'allreduce', the signature function expresses that the output is a function of all inputs: $\forall_i \colon \sigma(i) = N$.

3.2 Distributions

We now formally define distributions as mappings from processors to sets of indices:

$$\text{Distr} \equiv \text{Proc} \to \text{Ind}.$$

Traditionally, distributions are considered as mappings from data elements to processors, which assumes a model where a data element lives uniquely on one processor. By turning this definition around we have an elegant way of describing:

- Overlapping distributions such as halo data, where data has been gathered on a processor for local operations. Traditionally, this is considered a copy of data 'owned' by another processor.
- Rootless collectives: rather than stating that all processors receive an identical copy of a result, we consider them to actually own the same item.
- Redundant execution. There are various reasons for having operation executed redundantly on more than one processor. This can for instance happen in the top levels of a coarsening multilevel method, or in redundant computation for resilience.

We now bring together the concepts of signature function and distribution:

1. We can extend the signature function concept, defined above as mapping integers to sets of integers, to a mapping from sets to sets: with the obvious definion that, for $\sigma \in \text{Signature}, S \in \text{Ind}$:

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

   In our motivating example,

   $$\sigma\big([i_{\min}, i_{\max}]\big) = [i_{\min} - 1, i_{\max} + 1].$$

2. We then extend this to distributions with the definition that for $\sigma \in$ Signature and $u \in$ Distr

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

We now have the tools for our grand result.

3.3 Definition and use of β-distribution

Consider a data parallel operation $y = f(x)$ where y has distribution γ, and x has distribution α. We call a local operation to be one where every processor has all the elements of x needed to compute its part of y. By the above overloading mechanism, we find that the total needed input on processor p is $\sigma\big(\gamma(p)\big)$.

This leads us to define a *local operation* formally as:

Definition 1 *We call a kernel $y = f(x)$ a local operation if x has distribution α, y has distribution γ, and*

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

That is, for a local operation every processor already owns all the elements it needs for its part of the computation.

Next, we call $\sigma_f(\gamma)$ the 'β-distribution' of a function f:

Definition 2 *If γ is the output distribution of a computation f, we define the β-distribution as*

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

Clearly, if $\alpha \supset \beta$, each processor has all its needed inputs, and the computation can proceed locally. However, this is often not the case, and considering the difference between β and α gives us the description of the task/process communication:

Corollary 1 *If α is the input distribution of a data parallel operation, and β as above, then processor q is a predecessor of processor p if*

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

Proof. The set $\beta(p)$ describes all indices needed by processor p; if the stored elements in q overlap with this, the computation on q that produces these is a predecessor of the subsequent computation on p.

This predecessor relation takes a specific form depending on the parallelism mode. For instance, in message passing it takes form of an actual message from q to p; in a Directed Acyclic Graph (DAG) model such as OpenMP tasking it becomes a 'task wait' operation.

Remark 1 *In the context of Partial Differential Equation (PDE) based applications, our β-distribution corresponds loosely to the 'halo' region. The process of constructing the β-distribution is implicitly part of such packages as PETSc [4], where the communication resulting from it is constructed in the* `MatAssembly` *call. Our work takes this ad-hoc calculation, and shows that it can formally be seen to underlie a large part of scientific parallel computing.*

4 Practical importance of this theory

The above discussion considered operations that can be described as 'generalized data parallel'. From such operations one can construct many scientific algorithms. For instance, in a multigrid method a red-black smoother is data parallel, as are the restriction and prolongation operators.

In the IMP model these are termed 'kernels', and each kernel gives rise to one layer of task dependencies; see section 2.1. Taking together the dependencies for the single kernels then gives us a complete task graph for a parallel execution; the edges in this graph can be interpreted as MPI messages or strict dependencies in a DAG execution model.

Demonstration software along these lines has been built, showing performance comparable to hand-coded software; see [3].

5 Summary

In this paper we have given a rigorous mathematical definition of data distributions and the signature function of a data parallel operation. Our notion of data distribution differs from the usual interpretation in that we map processors to data,

rather than reverse. The signature function appears implicitly in the literature, for instance in stencil languages, but our explicit formalization seems new.

These two concepts effect a separation of concerns in the description of a parallel algorithm: the data distribution is an expression of the parallel aspects of, while the signature function is strictly a description of the algorithm. The surprising result is that these two give rise to a concept we define as the 'β-distribution'; it can be derived from data distribution and signature function, and it contains enough information to derive the communication / synchronization aspects of the parallel algorithm.

Demonstrating the feasibility of programming along these lines, we mention our Integrative Model for Parallelism (IMP) system, which implements these ideas, and is able to perform competitively with traditionally coded parallel applications.

Acknowledgement

This work was supported by NSF EAGER grant 1451204. The code for this project is available at https://bitbucket.org/VictorEijkhout/imp-demo.

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] Victor Eijkhout. IMP distribution theory. Technical Report IMP-01, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014.

[3] Victor Eijkhout. Report on NSF EAGER 1451204. Technical Report IMP-19, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014.

[4] W. D. Gropp and B. F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference, IEEE 1994*, pages 87–93.

[5] 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.