TACC Technical Report IMP-02

The deep theory of the Integrative Model

Victor Eijkhout*

February 22, 2016

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

Abstract

The Integrative Model for Parallelism (IMP) is normally described in terms of distributions. However, this assumes an 'owner-computes' model of parallel computing. There is a more theory of IMP, from which the distribution formulation follows easily.

Here we present the deep theory of IMP. This is only for afficionados.

1 Deep theory of the IMP model

1.1 Kernels

We define a *kernel* in the Integrative Model for Parallelism (IMP) model, which is the semantic specification of certain parallel operations. Formally, a kernel is a limited type of directed bipartite graph: a kernel is a tuple comprising an input data set, an ouput data set, and a set of edges denoting elementary computations that take input items and map them to output items:

$$K = \langle \mathrm{In}, \mathrm{Out}, E \rangle$$

where

In, Out are data structures, and

E is a set of (a,b) elementary computations, where $a \in \mathrm{In}, b \in \mathrm{Out}$,

and 'elementary computations' are simple computations between a single input and output.

For instance, a parallel 'map' of a function f can be modeled by isomorphic In and Out objects, and

$$E = \{\text{`}y \leftarrow f(x)\text{'} : x \in \mathrm{In}, y \sim x\}$$

where $x \in \mathrm{In}$, and y its isomorphic counterpart. On the other hand, a 'reduce' with an associative operator $\oplus$ can be modeled by letting Out be a singleton set $\{y\}$ and

$$E = \{\text{`}y \leftarrow y \oplus x\text{'} : x \in \mathrm{In}\}$$

for all $x \in \mathrm{In}$ and y an element of Out.

Typically, the input and output sets should be thought of as objects on the order of distributed matrices or vectors, and a kernel is a unit of computation on the order of a matrix-vector product, or a single stage in a Fourier transform.

The fact that an elementary computation has only a single input may seem a limitation, since it would exclude for instance a binary addition from being modeled. Rather than formulating our model in terms of hypergraphs, combining multiple inputs and outputs, we will keep the simpler model where multiple edges reaching a single output element are interpreted as a reduction operation. We will address the semantics of this, and argue for the model's generality in section 1.4.

Later in this story we will design a programming system based on kernels. In that context we use the term 'kernel' for something like a function that can be applied multiple times. The above definition then corresponds to an *execution* of such a programming kernel.

1.2 Parallel kernels

To parallelize a kernel over P processing elements, we define

$$K = \langle K_1, \ldots, K_P \rangle, \qquad K_p = \langle \mathrm{In}_p, \mathrm{Out}_p, E_p \rangle,$$

describing the parts of the input and output data set and (crucially!) the work that are assigned to processor p. We will use a loose concept of 'processing element' here: it can correspond to a core, a thread, cluster node, et cetera. The only restrictions on the distributions of a kernel are

$$\mathrm{In} = \bigcup_p \mathrm{In}_p, \mathrm{Out} = \bigcup_p \mathrm{Out}_p, E = \bigcup_p E_p;$$

none of these distributions are required to be disjoint.

We now see an immediate advantage of the IMP model in that it allows talking about large numbers of elementary tasks in a single distributed kernel object. This allows us to reason in a unified manner about the multitude, potentially millions, of tasks that make up one conceptual tasks.

1.3 Work distribution

In many applications the work distribution is induced by the *owner computes* rule, and therefore by the distribution of the output. However, there may be a good reason for using a different distribution. For instance, Berkeley folks have proved various results showing communication and energy benefits to a more direct distribution of the work, and using replicated data storage [3, 5].

1.4 Normal form

Above we allowed multiple arcs $e \in E$ to have the same endpoint:

$$e = (a,b) \quad \text{and} \quad e' = (a,b).$$

To give a formal meaning to this we need to take a step back.

Let a 'context' C be the set of all defined objects in some stage of a computation. In terms of concepts introduced earlier, $C \supset \mathrm{In}$, and after a kernel is finished the context is $C' = C \cup \mathrm{Out}$. Thus, we can write $K\colon C \to C'$ instead of $K\colon \mathrm{In} \to \mathrm{Out}$.

Like $\mathrm{In}, \mathrm{Out}$, contexts are partitioned: $C = \cup_p C_p$. As with the $\mathrm{In}_p, \mathrm{Out}_p$ partitioning this is not disjoint; rather than viewing it as an assignment of 'objects belonging to a processor', we view this as 'a processor being able to see an object'.

We need this concept of a distributed context to give meaning to the distribution $p \mapsto E_p$. We say that $e = (a,b) \in E_p$ if, apart from a, all quantities needed for the e-computation are in C_p.

Now we can consider a sequence of kernels

$$C^{(0)} \xrightarrow{K^{(0)}} C^{(1)} \cdots C^{(s-1)} \xrightarrow{K^{(s-1)}} C^{(s)}$$

where, as observed above

$$0 \leq t < s\colon C^{(t+1)} = C^{(t)} \cup \mathrm{Out}^{(t)}.$$

Let us now consider the special case where a sequence of instructions $e^{(t)} \in E_p^{(t)}$ can be described by;

$$(\alpha_t, \beta_t) \in E_p^{(t)} \quad \text{and} \quad \begin{cases} \alpha_t \in C^{(0)} \\ \beta_t \in \mathrm{Out}_p^{(t)} \end{cases}$$

That is, the instructions are executed on processor p, and contribute to the respective output sets on p, but none of the intermediate results is directly an input: all inputs come from the original context.

We can now summarize this computation as

$$\beta^{(s-1)} = f(C_p^{(0)}, \alpha^{(0)}, \ldots, \alpha^{(s-1)}).$$

If the computation in f is associative and we never need the intermediate results $\beta^{(t)}, t < s$, we see that we can collapse all the kernels into one that has arcs

$$(\alpha_0, \beta^{(s-1)}), \ldots, (\alpha_{s-1}, \beta^{(s-1)})$$

and the interpretation of this is no longer ambiguous.

However, describing operations such as a reduction in terms of single element-by-element reductions is slightly cumbersome, so we introduce a 'normal form'; see figure 1. A kernel is on normal form if either

1. it contains no arcs (α_1, β), (α_2, β) with the same endpoint, where $\alpha_1 \in \mathrm{In}_p$, $\alpha_2 \in \mathrm{In}_q$ and $p \neq q$, or
2. for any pair (α_1, β), (α_2, β) with the same endpoint, both sources and the endpoints are on the same processor: $\alpha_1, \alpha_2 \in \mathrm{In}_p$ and $\beta \in \mathrm{Out}_p$.

We see that IMP kernels on normal form have a strict separation and alternation between episodes of process/thread-local computation and communication. For an actual global reduction operation this is inefficient by a factor of 2 [2], but we can handle this as a special case. For many other reduction-type operations this normal form simplifies discussion greatly.

This normal form also means that it is not possible to have a stream of instructions that operates on a stream of incoming data. However, we do not feel this as a restriction; rather, we regard it as

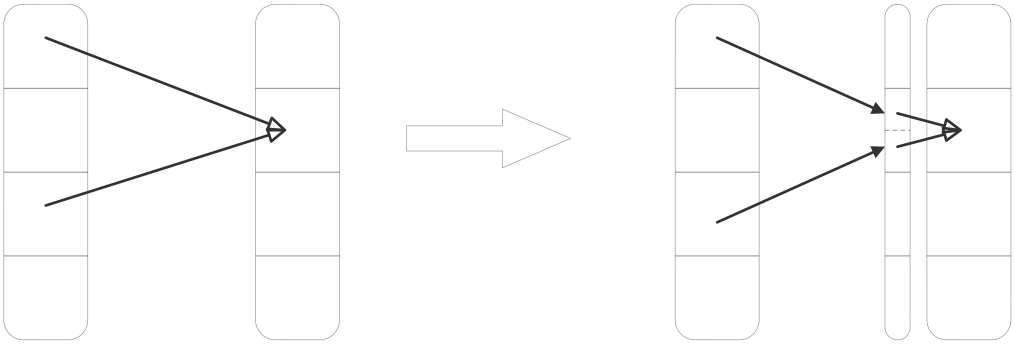

Figure 1: Transformation of a reduction to a normal form consisting of copy-to-local and local reduction

enforcing data aggregation in the programming system. Such aggregation is usually considered a Good Thing.

We also note the similarity of these episodes with Bulk Synchronous Parallelism (BSP) supersteps; however, our model does not feature any explicit synchronization: tasks can fire asynchronously in a *dataflow* [9] manner. (See section **??** for more details.) In fact, kernels appearing in a cycle (section **??**) will have tasks that are of necessity not simultaneous.

The above described separation of communication and local computation makes sense in distributed memory systems. There is an argument to be made that even in shared memory parallelism this mode of computing has its uses. In coherent shared memory context, a processor can obtain a copy of data into its local cache by simply touching the address for read or write: the coherence protocol will move the data from wherever it resides, whether that be memory or a different local cache.

Programming this way is certainly easy to code; however, it has many disadvantages. Explicitly control of data movement in a shared memory context is desirable enough that it has led to the development of 'cache-oblivious' techniques [8] and 'communication avoiding' algorithms [4]. Both techniques are laborious. Our model, on the other hand, has the promise of being able to control data movement explicitly without too much programmer tedium.

In terms of the LRSX characterization (section 1.7), we see that a general kernel containing all four types will have to be split into, in sequence, a *pre* communication kernel for the R and X instructions, a computation kernel for all four types, and a *post* communication kernel for the S and X instructions. In most practical cases a kernel will have exclusively R or S type instructions, in which case we reduce it to local computation plus a kernel that only contains the data movement.

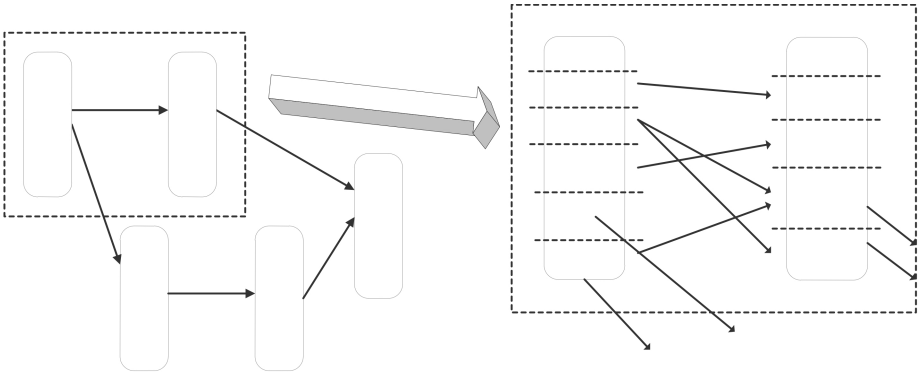

Figure 2: Illustration of kernel and task dataflow

1.4.1 Active messages

As a further observation, we can define two normal forms: the R-normal form which has only L and R operations, and the S-normal form which has only L and S. If we have a kernel on R-normal form it can easily be converted to S-normal form. If K is on R-normal form and $e = (a,b) \in E^K$ with $a \in p, b \in q$, then $e \in E_q$. By assigning such e to E_p we convert the kernel to S-normal form while keeping the computation intact.

The R-normal form corresponds to a 'pull' model of data movement; often the dominant style of MPI coding, as well as the implicit model behind PGAS languages. The S-noral form, on the other hand, is ambiguous to interpret if we take remote procedure call into account. An instruction e being $e \in E_p$ can mean being physically issued on processor p, but it can also be interpreted as issued on p to be executed elsewhere. Under this latter interpretation, the S-normal form is seen to cover 'active messages'.

1.5 From kernels to algorithms

Next we consider constructing a full algorithm by composing kernels. This gives us an 'abstract algorithm': a description of data dependencies without regard for architectural details.

This composition process can be interpreted as graph construction, or as function composition if we interpret kernels in a functional manner; $\text{Out} \leftarrow K(\text{In})$.

This corresponds to a high level form of *dataflow*; figure 2 shows the connections between kernels, and the finer-grained connections between tasks in a kernel.

By now the reader may have a mental image of kernels as synchronously operating, as in BSP. This is not necessarily the case: we are not imposing any type of synchronization, and individual processing elements can operate governed only by the availability of their In data. But even more radically, we have not ruled out circular connections between kernels, which can imply serial dependencies between the tasks in a single kernel; see section **??**.

1.6 Derivation of data movement

Based on the fact that the computations in E_p are executed on processor p we can now define the input and output data for these computations:

$$\begin{aligned} \mathrm{In}(E_p) &= \{a\colon (a,b) \in E_p\}, \\ \mathrm{Out}(E_p) &= \{b\colon (a,b) \in E_p\}. \end{aligned}$$

These correspond to the input elements that are needed for the computations on processor p, and the output elements that are produced by those computations. These sets are related to $\mathrm{In}_p, \mathrm{Out}_p$ but are not identical: in fact we can now characterize the communication involved in an algorithm as

$$\begin{cases} \mathrm{In}(E_p) - \mathrm{In}_p & \text{data to be communicated to } p \text{ before computation on } p \\ \mathrm{Out}(E_p) - \mathrm{Out}_p & \text{data computed on } p, \text{ to be communicated out afterwards} \end{cases}$$

We see that some simple cases are covered by our model: the common 'owner computes' case corresponds to

$$\mathrm{Out}(E_p) = \mathrm{Out}_p,$$

that is, each processor computes the elements of its part of the output data structure and no data is communicated after being computed. If additionally $\mathrm{In}(E_p) = \mathrm{In}_p$, we have an embarrassingly parallel computation because no communication occurs before or after computation.

1.7 LRSX diagrams

We now characterize the computations in E_p based on the localization of their input and output data. We split

$$E_p = L_p \cup R_p \cup S_p \cup X_p$$

where

$$L_p = \{(a,b) \in E_p\colon a \in \mathrm{In}_p, b \in \mathrm{Out}_p\}$$

are the operations that take local input data and touch only local elements of the output data structure, and therefore can be performed without communication. The set

$$R_p = \{(a,b) \in E_p\colon a \in \mathrm{In}(E_p) - \mathrm{In}_p, b \in \mathrm{Out}_p\}$$

are the operations that compute local output elements but use data not present on p, and therefore require communication prior to the computation. This 'communication before computation' is by

far the most common parallel programming paradigm. The set $\mathrm{In}(E_p) - \mathrm{In}_p$ is often called the 'ghost region' or *halo region*.

Next,

$$S_p = \{(a,b) \in E_p \colon a \in \mathrm{In}_p, b \in \mathrm{Out}(E_p) - \mathrm{Out}_p\}$$

is a less common set of operations that take local input data but produce data for other processors, so communication will occur after the computation. The set $\mathrm{Out}(E_p) - \mathrm{Out}_p$ is not commonly encountered in the literature, an exception being [6]; it appears in algorithms on matrices distributed by columns.

Finally,

$$X_p = \{(a,b) \in E_p \colon a \in \mathrm{In}(E_p) - \mathrm{In}_p, b \in \mathrm{Out}(E_p) - \mathrm{Out}_p\}$$

are operations that take non-local input and produce non-local output, so they require communication both before and after the computation. Such operations occur if dense matrix operations are mapped to a 2D processor grid; they also appear in zonal methods for N-body problems [1]. However, we will not show any examples in this paper.

The sets L_p, R_p, S_p, X_p are illustrated in figure 3. This figure is suggestive of a matrix-vector prod-

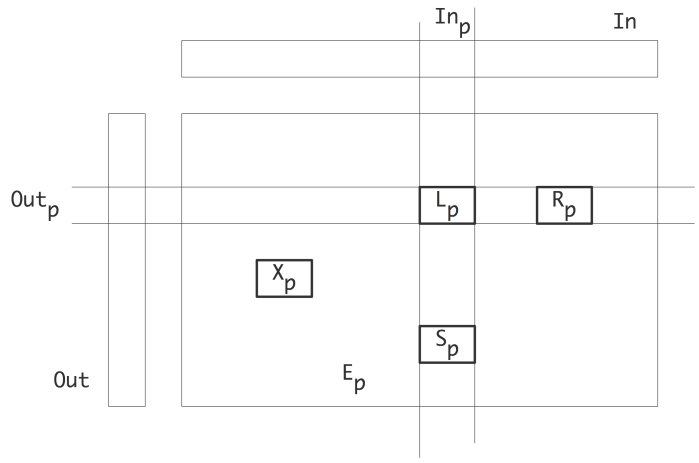

Figure 3: The sets L_p, R_p, S_p, X_p related to the input and output data structures

uct, but its strict interpretation is that of a cross-tabulation of all instructions in E_p: given the distributions of the input and output data, to which of the four sets L_p, R_p, S_p, X_p do instructions in E_p belong. Another intuitive interpretation of this figure is as an adjacency matrix where position (i, j) is nonzero if processor j communicates with i. However, note that the LRSX diagram has additional information about instructions being executed.

It is clear that the instructions in R_p, S_p, X_p require data communication. To make source and target of communications more explicit we define

$$\begin{array}{ll} R_{q\to p} & = \{e \in R_p \colon e = (a,b) \wedge a \in \mathrm{In}_q\}, \\ S_{p\to q} & = \{e \in S_p \colon e = (a,b) \wedge b \in \mathrm{Out}_q\}, \\ X_{q\to p\to r} & = \{e \in X_p \colon e = (a,b) \wedge a \in \mathrm{In}_q \wedge b \in \mathrm{Out}_r\}, \end{array}$$

that is, $R_{q\to p}$ contains those R_p instructions with input on processor q, $S_{p\to q}$ contains those S_p instructions for which the output belongs on q, and $X_{q\to p\to r}$ contains instruction with input on q, output on r, executed on p.

2 Examples

Among the many research questions about the IMP model, maybe foremost are questions of deadlock and determinacy that arise when we chain kernels together into algorithms with circular dependencies as shown above.

We distinguish two types of circularity. First of all, algorithms such as a Conjugate Gradient solver have a circular dependency, for instance from the output of the matrix-vector product in one iteration to the input in the next. We expect little added value from considering the totality of all iterations, so we will limit ourselves to expressing and analyzing a single iteration and the data traffic in it, which does not involve kernel cycles.

Secondly, certain algorithms such as solving a triangular system have true circularities as illustrated in figure 4: the computation of one solution component becomes the input for one or more next components. This circularity is in fact surprisingly easy to model in IMP; see our technical report [7]. The formal treatment of deadlock in such cycles is still a research topic.

Consider a simple prefix sum.

$$y_i = \sum_{j<i} y_j.$$

First of all since the input and output of a kernel need to be distinct, we split this as two interlinked operations

$$\begin{cases} y_i = \sum_{j<i} t_j & \text{where} \\ t_j = y_j \end{cases}$$

Next, since our normal form requires all reductions to be local, so the reduction $y_i \sum_{j<i} t_j$ requires t_* to be local. We solve this by make t two-dimensional and replicating it. We write

$$\begin{cases} y_i = \sum_{j<i} t_{ij} & \text{where} \\ t_{ij} = y_j \quad \forall_i \end{cases} \tag{1}$$

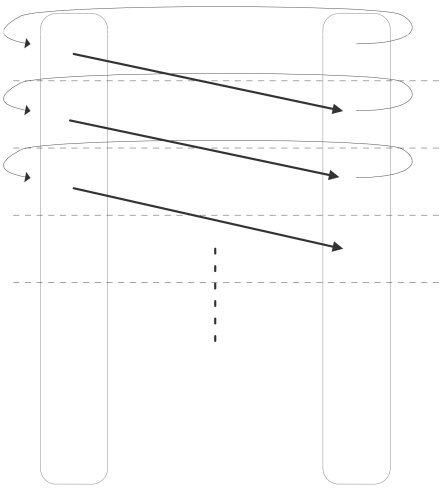

Figure 4: Sequential dependencies between tasks deriving from circular composition of kernels

This can be written in terms of distributions as

$$\begin{cases} y(u) &= \sum_{\mathrm{scan},2} t(u,*) \\ t(u,*) &= y(*) \end{cases} \tag{2}$$

In the scalar formulation of equation (1) we recognize, translating scalar operaterations to tasks, that the task graph is acyclic; in the distribution formulation (2) we see this expressed in the presence of a 'scan' operation.

2.0.1 Two-sided pipelines: Gauss-Seidel

Can we formulate the GS method, with the left and right communication?

References

[1] Kevin J. Bowers, Ron O. Dror, and David E. Shaw. Zonal methods for the parallel execution of range-limited n-body simulations. *Journal of Computational Physics*, 221(1):303 – 329, 2007.

[2] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 2007.

[3] James Demmel, Andrew Gearhart, Oded Schwartz, and Benjamin Lipshitz. Perfect strong scaling using no additional energy. Technical Report UCB/EECS-2012-126, EECS Department, University of California, Berkeley, May 2012.

[4] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *IEEE International Parallel and Distributed Processing Symposium*, 2008.

[5] Michael Driscoll, Evangelos Georganas, Penporn Koanantakool, Edgar Solomonik, and Katherine Yelick. A communication-optimal N-body algorithm for direct interactions. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 213.

[6] Victor Eijkhout and Roldan Pozo. Lapack working note 77, Basic concepts for distributed sparse linear algebra operations. Technical Report CS-94-240, Computer Science Department, University of Tennessee, Knoxville, 1994.

[7] Victor Eijkhout, Barry Smith, and Ritu Arora. A global model for parallel programming. Technical Report TR-2011-03, Texas Advanced Computing Center, The University of Texas at Austin, 2011. http://www.tacc.utexas.edu/~eijkhout/Articles/2011-eijkhout-globalmodel.pdf.

[8] M. Frigo, Charles E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, 1999.

[9] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. Appl Math.*, 14:1390–1411, 1966.