A Calculus for Parallel Computations over Multidimensional Dense Arrays

Roberto Di Cosmo*, Susanna Pelagatti†, Zheng Li*

* *Laboratory "Preuves, Programmes et Systémes",*
University of Paris VII 175, rue du Chevaleret, F-75013 Paris France
e-mail:{dicosmo,li}@pps.jussieu.fr and INRIA Rocquencourt, project Cristal

† *Dipartimento di Informatica, University of Pisa,*
Via F. Buonarroti, 2 I-56127 Pisa Italy e-mail:susanna@di.unipi.it

January 5, 2004

Abstract

We present a calculus to formalize and give costs to parallel computations over multidimensional dense arrays. The calculus extends a simple distribution calculus (proposed in some previous work) with computation and data collection. We consider an SPMD programming model in which process interaction can take place using point-to-point as well as collective operations, much in the style of MPI. The idea is give rigorous description of all stages of data parallel applications working over dense arrays: initial distribution (ie, partition and replication) of arrays over a set of processors, parallel computation over distributed data, exchange of intermediate results and final data gather. In the paper, beside defining the calculus, we give it a formal semantics, prove equations between different combinations of operations, and show how to associate a cost to operation combinations. This last feature makes possible to make quantitative cost-driven choices between semantically equivalent implementation strategies.

1 Introduction

Large number-crunching problems over multi-dimensional dense arrays have always been an important application area for parallel computing and, over the years, many libraries and tools have been built to ease the parallelization such problems (see for instance [5, 3, 21]). However, surprisingly little effort has been spent in trying to develop a systematic formal methodology to assist the development of such applications in a systematic way. In particular, when designing the efficient implementation of an application working on dense multi-dimensional arrays, a programmer needs to compare different strategies for distributing/moving data across processors, take into account many machine specific details and reason about the relative performance of a usually large space of options. This is usually done in an ad-hoc way, drawing complex data graphs on paper and trying to figure out the actual correctness of the strategy at hand. The problem is even worse when one tries to provide general and efficient implementations of high-level mechanisms describing in a compact way large families of computations over dense arrays (with an arbitrary number of dimensions), as we experienced when trying to incorporate the powerful P3L Map skeleton in the OcampP3L library [7]. The actual efficient implementation of the general Map requires us to compare different data

distribution and recollection strategies, to evaluate their costs and to prove their correctness in a formal way.

In this paper, we make a step towards the development of a formal framework to reason about dense multimdimensional arrays, with unlimited number of dimensions. In particular, we assume an SPMD model of computation, in which process interaction can take place using point-to-point as well as a small set of collective operations, much in the style of MPI and we define a calculus which allows to formalize all the steps of application, from the initial data distribution, to the final collection of results. The calculus is simple enough to allow a readable semantics, yet powerful enough to allow us to compare realistic implementation strategies.

The contributions and the structure of the paper are as follows. Section 2 motivates our work through a simple example. Then, Section 3 defines a model for multidimensional dense arrays with unlimited number of dimensions and for processors. In particular, we formalize the concept of communicator provided by MPI and use it to bound the scope of our collective array operations. Then, Section 4 defines our calculus targeted at the description of parallel computations for multidimensional dense arrays over a set of processors. We give a formal semantics to it in the style of denotational semantics, that allows to prove equations between implementation strategies. Then, we discuss two cost models for the calculus, one adopting BSP style of iteraction and one adopting MPI asynchronous style (Sec. 5). This will allow us to weight different implementation strategies and perform informed choices between semantically equivalent options. Finally, related work is reviewed in Section 7 and Section 8 concludes.

2 A motivating example

We now introduce some of the problems to be addressed through a simple example, which will be used as a running example in the following sections.

Consider the problem of matrix multiplication, that is we compute $C = B * A$, where B and A are two dense matrices $n \times k$ and $k \times m$ respectively. Devising a good algorithm for matrix multiplication involves devising a good strategy for breaking-up the underlying data –initial and final matrices– among processors, and distributing the computation load to each processor.

Matrix multiplication has been widely studied in the literature and many efficient parallel algorithms have been proposed (see for instance [19]). However, as our goal is to demonstrate the problems to be addressed, we choose a very simple algorithm which is illustrated in Figure 1. We assume a grid of $p \times p$ ($p = 3$ in the picture) processors each one computing a block of the result matrix C. Each processor computes a submatrix of the result C using a group of rows of A and a group of columns of B. In particular, processor (i, j) needs two columns $(2j,\ 2j + 1)$ of `A` and two rows $(2i,\ 2i + 1)$ of `B` and produces a 2×2 matrix of C ($C[2i : 2i + 1][2j : 2j + 1]$).

The implementation of such an algorithm requires: (1) the distribution of data to processors, (2) the computation of submatrixes by each processor and (3) the gather of submatrices to build the result matrix C.

Each step can be implemented in several ways. We need to formalize all the three steps in order to choose between alternative implementation strategies, as well as to compare the performance expected form different choices.

For instance, the initial distribution requires two multicast operations, where rows of `B` and columns of `A` are sent to subsets of the 9 processors.

There are several possible strategies to perform the initial multicast of A:

direct send data may be sent directly from the node holding A to all the processors (which requires the same column to be sent several times, to each processor needing it), or

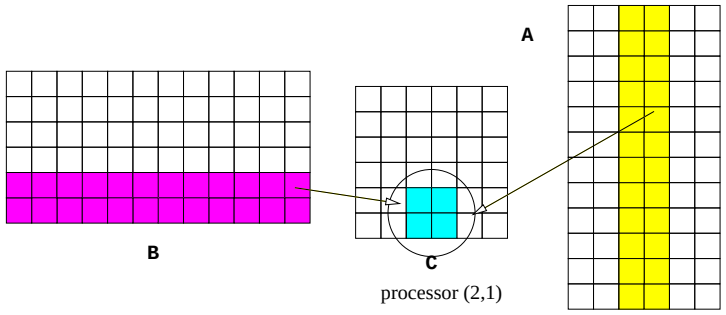
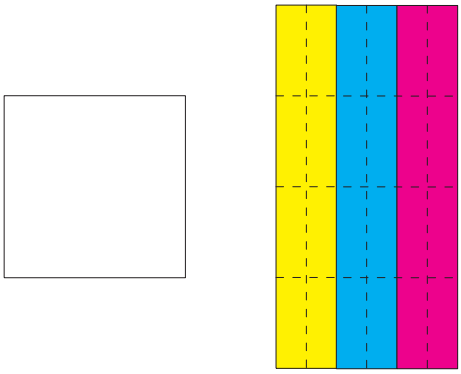


Figure 1: Matrix Multiplication: data needed by processor $(2, 1)$.

scatter&broadcast (Figure 2) we can first send each column (*only once*) to some well chosen processor (eg, columns $2i$ and $2i + 1$ to processor $(0, i)$) via a scatter operation, and then broadcast in parallel each column to each processor needing it (eg, processor $(0, i)$ broadcasts $A[][i]$ to all processors (j, i)), or

scatter&gather (Figure 3) we can first scatter A on the processor grid (eg, processor (i, j) gets $A[4i : 4(i + 1) - 1][2j : 2(j + 1) - 1]$) and then perform a set of concurrent local gather to get all needed parts from other processors on the same column.



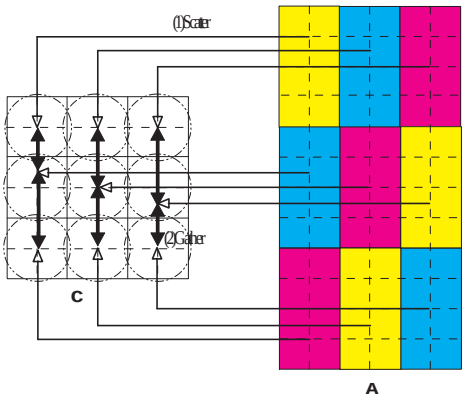


Figure 3: Scatter&gather multicast strategy.

3 A model for dense arrays

In this section, we extend the model for array distributions presented in [6] adding intermediate data collection, independent computations over processors and result gathering. We first recall some basic definitions and then introduce the operations.

3.1 Index domains and multidimensional arrays

We model arrays as functions from index domains to values.

We write $[l : h]$ for the integer interval with lower bound l and upper bound h, and we denote interval bounds as $inf([l : h]) = l$ and $sup([l : h]) = h$.

Definition 1 (Index domain) *An* index domain *I is a Cartesian product of integer intervals*

$$I = \prod_{i=1}^{n} [l_i : h_i]$$

also written $[l_i : h_i | i \in 1..n]$. We call n the dimensionality *of the index domain I.*

Definition 2 (Index) *An* index *$(i_1, i_2, \ldots, i_n)$ is a sequence of integer, with the value in each dimension i_m correspondingly falling into the interval $[l_m : h_m]$ of the index domain it belongs to. The n,* dimensionality *of the index, should agree with the* dimensionality *of its index domain.*

For convenience, an index $(i_1, i_2, \ldots, i_n)$ is often written in the form of vector like $\vec{i}$. Several vector operators are also imported.

- $z \times \vec{i} = \vec{j}$ $(z \in \mathbb{Z})$
  $\vec{i}, \vec{j}$ have the same dimensionality n and $\forall m \in [1, n]$. $j_m = z \times i_m$
- $\vec{i} + \vec{j} = \vec{k}$
  $\vec{i}, \vec{j}, \vec{k}$ have the same dimensionality n and $\forall m \in [1, n]$. $k_m = i_m + j_m$
- $\vec{i} \otimes \vec{j} = \vec{k}$
  $\vec{i}, \vec{j}, \vec{k}$ have the same dimensionality n and $\forall m \in [1, n]$. $k_m = i_m \times j_m$

Definition 3 (Shape of index domains) *Given an index domain $I = \Pi_{i=1}^{n}[l_i : h_i]$, function* ibase *returns its basic shift in each dimension from the origin $\vec{0}$, and function* idims *computes all its dimensions.*

$$\begin{aligned} ibase(I) &= (l_1, l_2, \ldots, l_n) \\ idims(I) &= (h_1 - l_1 + 1, h_2 - l_2 + 1, \ldots, h_n - l_n + 1) \end{aligned}$$

Outputs of both *ibase* and *idims* are integer sequences, so they can also be taken as vectors.

Definition 4 (Shift of index domains) *Given an index domain* $I = \prod_{i=1}^{n}[l_i : h_i]$ *and a shift vector* $\vec{s} = (s_1, s_2, \ldots, s_n)$*, the function* ishift *is defined as*

$$ishift^{\vec{s}}(I) = \prod_{i=1}^{n}[s_i : s_i + h_i - l_i]$$

which shifts the index domain I *to a new position where its basic shift from origin equals to* $\vec{s}$*.*

Projections extract subdomains of an index domain.

Definition 5 (Projection of index domains) *Given a domain* $I = \prod_{i=1}^{n} D_i$*, and an injective mapping* $\sigma : k \to n$*, with* $k < n$*, that pinpoints the dimensions maintained by the projection, we define the* projection of I according to σ *as*

$$proj_{\sigma}(I) = \prod_{i=1}^{k} D_{\sigma(i)}$$

Now we are ready to define arrays and operations on arrays.

Definition 6 (Array) *An array* A *is a* partial *map* $A : I \to V_{\perp}$ *defined on an index domain* I*, returning either a value in* V *or the "undefined" element* $\perp$*. For brevity, we will just write* V *instead of* $V_{\perp}$ *in the following. Obviously* $A(\vec{i})$ *returns the corresponding value of index* $\vec{i}$*, given that* $\vec{i} \in I$*.*

On an array $A : I \to V$ *we define the usual operations* $dom(A) = I$ *and* $codom(A) = V$*, and the* dimensionality *of* A *is the* dimensionality *of* I*.*

Similarly, we can define the shape functions for arrays.

Definition 7 (Shape of arrays) *Given an array* $A : I \to V$*, function* abase *returns the basic shift of the its index domain from the origin, and function* adims *computes all the dimensions of its index domain. More precisely,*

$$abase(A) = ibase(dom(A)) = ibase(I)$$
$$adims(A) = idims(dom(A)) = idims(I)$$

Definition 8 (Shift of arrays) *Given an array* $A : I \to V$ *and a vector* $\vec{s}$ *which has the same dimensionality, the function* $ashift^{\vec{s}}(A)$ *produces an array defined on the index domain* $ishift^{\vec{s}}(I)$ *and for each* $\vec{i} \in I$

$$ashift^{\vec{s}}(A)(\vec{i} + \vec{s} - abase(A)) = A(\vec{i})$$

Notation 1 (Block Selection) *We will use a* block selection *notation: if* $A : I \to V$*, with* $I = (\prod_{i=1}^{n} D_i)$*, then* $A' = A[l_i : h_i | i \in 1..n]$ *is the array defined by the restriction of* A *to the index domain* $I' = (\prod_{i=1}^{n}[l_i : h_i])$*.* This is a selection which preserves global indexes*, as elements of* A' *have the same index they had in* A*.*

Figure 4 shows domains and codomains of arrays A, B and C of our running example. Block selection notation is used to denote groups of rows, groups of columns and submatrices.

We often need to 'glue' together two arrays that define different parts of an index domain (eg, two rows of a matrix). This is captured by array pasting.

Definition 9 (Array Pasting) *Given two* conformant *(ie, defined on the same index domain) arrays* $A : I \to V$*,* $B : I \to V$*, the* pasting $A \oplus B : I \to V$ of A and B *is defined as*

$$A \oplus B(\vec{i}) = \begin{cases} A(\vec{i}) \text{ if } A(\vec{i}) \text{ is defined and } B(\vec{i}) = \perp \\ B(\vec{i}) \text{ if } B(\vec{i}) \text{ is defined and } A(\vec{i}) = \perp \\ \perp \text{ otherwise} \end{cases}$$

Notice that array pasting is a commutative operation, but it is not associative. This works well for our purposes, however, if associativity is needed, the definition should be extended introducing an explicit error value to use on the points where both arrays are defined.

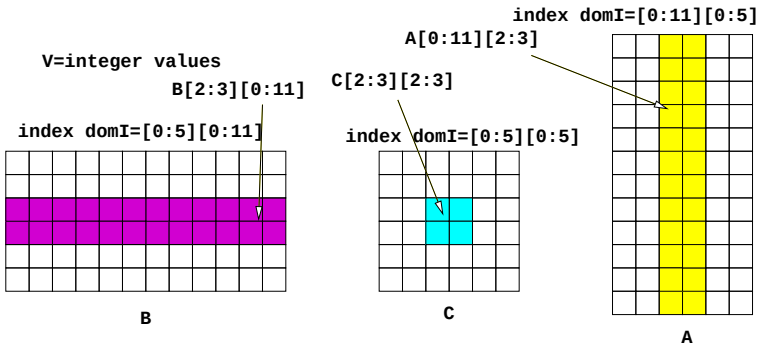


Figure 4: Index domains, arrays and block selection.

3.2 Processors and communicators

We can now model a set of processors, indexed on a domain I, as an array, which is the formal counterpart of an MPI (Cartesian) communicator.

Definition 10 (Cartesian communicator) *On the set P of all processors, one can define a* Cartesian communicator *C as a function $C : I^C \to P$, where I^C is an index domain. This allows to structure the processors as an array (we will often just speak of processor arrays). A Cartesian communicator defines an index function $index^C : P \to I^C$ which assigns to each processor its index.*

3.3 Array manipulation

We now define the basic operations on arrays we are interested in. These operations are concerned with extracting subarrays, composing arrays and injecting arrays into arrays of larger *dimensionality*.

3.3.1 Array projections

A *projection* is a mapping from an index domain to another of smaller *dimensionality*, obtained by erasing some dimensions.

Notation 2 (Complement of an injective mapping) *In what follows, given an injective mapping $\sigma : [1 : k] \to [1 : n]$, we will write $\bar{\sigma}$ for the injective mapping $[1 : n - k] \to [1 : n]$ enumerating in increasing order the complement of the codomain of σ.*

For instance, if we consider the injective mapping $\sigma : [1 : 2] \to [1 : 5]$ defined as $\sigma(1) = 2$ and $\sigma(2) = 3$, then $\bar{\sigma} : [1 : 3] \to [1 : 5]$ and $\bar{\sigma}(1) = 1$, $\bar{\sigma}(2) = 4$, $\bar{\sigma}(3) = 5$.

Definition 11 (Array Projection) *Given an array $A : I \to V$, $I = (\Pi_{i=1}^{n} D_i)$, an injective mapping $\sigma : k \to n$, with $k < n$, and a vector $\vec{v} \in \Pi_{i=1}^{n-k} D_{\bar{\sigma}(i)}$ we define the projection of A along σ and $\vec{v}$, $proj_{\sigma,\vec{v}}(A) : proj_\sigma(I) \to V$ as*

$$proj_{\sigma,\vec{v}}(A)(i_1, \ldots, i_k) = A(a_1, \ldots, a_n)$$

where $a_j = i_{\sigma^{-1}(j)}$ if $\sigma^{-1}(j)$ is defined, and $a_j = \vec{v}[\bar{\sigma}^{-1}(j)]$ if $\bar{\sigma}^{-1}(j)$ is defined.

Notation 3 (Projections of domains) *We write $dom_\sigma(A)$ for $proj_\sigma(dom(A))$*

In practice, σ pinpoints the dimensions that we want to maintain during projection and $\vec{v}$ fixes the indexes for the dimensions which will be erased by projection. Figure 5 shows the projection of array A in our running example, where we have pinpointed the first dimension, and fixed the index of the second one to 3.

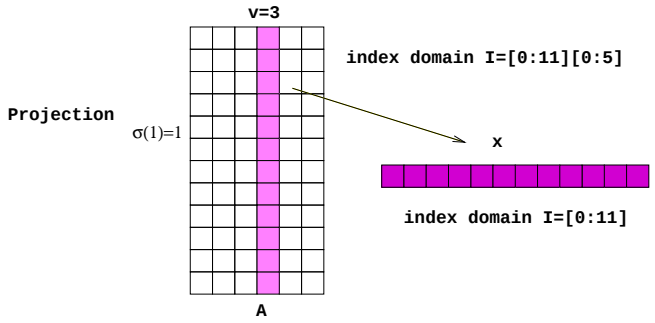


Figure 5: Projecting array A along $x = proj_{\sigma,\vec{v}}(A)$, $\sigma : 1 \to 2$, $\vec{v} = (3)$.

3.3.2 Array injections

The second operation we are interested in is *injection* which 'inserts' an array into another one of larger dimensionality (which we will call the *universe*).

Definition 12 (Array Injection) *Given an array $A : I \to V$, where $I = (\Pi_{i=1}^{k} D_i)$, an injective mapping $\sigma : k \to n$, with $k < n$, $U = (\Pi_{i=1}^{n} E_i)$ with $E_{\sigma(i)} = D_i$ for $i \in 1..k$, and a vector $\vec{v} \in \Pi_{i=1}^{n-k} E_{\bar{\sigma}(i)}$, we define the injection of A into U along σ and $\vec{v}$ as the function $inj^{U}_{\sigma,\vec{v}}(A) : U \to V$ as*

$$inj^{U}_{\sigma,\vec{v}}(A)(e_1, \ldots, e_n) = \begin{cases} A(e_{\sigma(1)}, \ldots, e_{\sigma(k)}) \text{ if } e_{\bar{\sigma}(i)} = \vec{v}[i], i \in 1..n-k \\ \perp \text{ otherwise} \end{cases}$$

σ selects a projection of the universe U which is *conformant* (ie, has the same index domain) of the array A to be injected. Then injection inserts A in this projection, leaving the rest of U undefined. Figure 6 shows how to inject back vector x (extracted in Figure 5) back into array A.

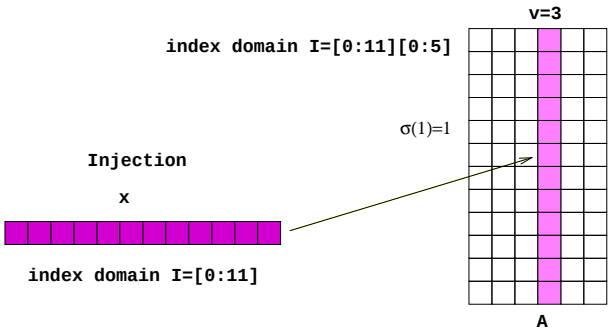


Figure 6: Injecting vector x into array A, $inj^{A}_{\sigma,\vec{v}}(x)$, $\sigma : 1 \to 2$, $\vec{v} = (3)$

Projection and injection are related operations, as they allow us to decompose and recompose an array defined on an index domain via arrays defined on domains of smaller *dimensionality*. More formally, the following theorem easily holds.

Theorem 1 (Covers of A) *For any array $A : I \to V$, it is the case that*

$$A = \bigoplus_{\vec{v} \in dom_{\bar{\sigma}}(A)} inj^{dom(A)}_{\sigma,\vec{v}}(proj_{\sigma,\vec{v}}(A)) \qquad (1)$$

3.4 Array distribution

Now we turn to a more complex operation, which is the *block distribution* of an array A over the values of another array B. This assigns a subrange of $dom(A)$ to each index of $dom(B)$ and is modelled by an index domain mapping.

Definition 13 (Dependent index distribution)
Given an index domain $I = (\Pi_{i=1}^{m} D_i)$, and an index domain $J = (\Pi_{i=1}^{n} E_i)$, a dependent index distribution *is a dependent function $gs :: J \to P(I)$ that associates to each index in J a slice of indexes in I (an element of $P(I)$, the powerset of I). More precisely, using dependent type notation [25], one would write*

$$gs :: \Pi\ \vec{j} \in J.\ \prod_{i=1}^{m} D'_i(\vec{j})$$

to stress that to each index of J is assigned a subrange of the indexes in I.

This way of defining subranges of I produces subcubes of I that are of the same *dimensionality* as I, even if it is quite possible that a subcube only consists of a single datum (if all index ranges are reduced to a single point). This suffices for modelling our block distribution strategies, but one may imagine to allow changes in dimensions too, to model more complex distributions.

According to the definition, we have $gs(\vec{j}) : \Pi_{i=1}^{m} D'_i(\vec{j})$. It's very clear here that the function gs associates each $\vec{j}$ in J with a sub domain of another index domain I. Practically, it is used to describe the *association* between each processor index $\vec{j}$ and a corresponding sub domain from I on which a original array is defined.

Example 1 (Mappings underlying some distribution strategies)

broadcast *The* broadcast *of an array $A : I \to V$ is modeled by the constant index distribution $gs_{bcst} : \lambda\vec{j}.I$.*

block scatter *The* block scatter *of A over J is modeled as follows. Given $I = (\Pi_{i=1}^{m} D_i)$ of the same dimensionality as the index domain $J = (\Pi_{i=1}^{m} E_i)$, and given a vector of block sizes $[h_1, \ldots, h_m]$, the block-scatter strategy for I over J is the function*

$$gs_{bs}(j_1, \ldots, j_m) = [inf(I_k) + j_k * h_k : inf(I_k) + (j_k + 1) * h_k - 1 | k \in 1..m]$$

Note that gs, the dependent index distribution function, describes merely a *map relation* between processor indexes and sub domains of array. The function itself does not specify any operational actions. So it could be used to describe not only the relation between processor indexes and the sub domain *to be* distributed on them (which usually concerns with distribution *operations*), but also the association between processor indexes and the sub domains *already* distributed on them (which usually concerns with distribution *states*). It depends on the definitions of the operator functions which it is exploited.

We can now model the *distribution operation* of an array A on a communicator C using a given index distribution gs.

Definition 14 (Distribution over a communicator) *Given an array $A : I \to V$, a communicator $C : J \to P$ and an index distribution $gs : J \to P(I)$, we define the* distribution *of A over C according to gs (written in form $\mathbf{distr}_C^{gs}(A)$), as an array defined over J in which each element of index $\vec{j}$ is a sub-array of A restricted to the sub index domain $gs(\vec{j}) : \Pi_{i=1}^{m} D'_i(\vec{j})$, if $C(\vec{j})$ is defined, and $\perp$ otherwise. More formally:*

$$\mathbf{distr}_C^{gs}(A)(j_1, \ldots, j_n) = \begin{cases} A[gs(j_1, \ldots, j_n)] & if\ C(j_1, \ldots, j_n) \neq \perp \\ \perp & otherwise \end{cases}$$

Example 2 (Some well known distributions) *Given an array $A : I \to V$ and a communicator $C : J \to P$, we formalize the following distributions.*

broadcast *The* broadcast of A over C *is described by* $\mathbf{distr}_C^{\lambda\vec{j}.I}(A)$.

block scatter *If the index domain* $I = (\Pi_{i=1}^m D_i)$ *has the same dimensionality as the index domain* $J = (\Pi_{i=1}^m E_i)$*, and* $[h_1, \ldots, h_m]$ *is a vector of block sizes, the* block-scatter distribution of A over C *is the function*

$$\mathbf{distr}_C^{\lambda(j_1,\ldots,j_m).[inf(I_k)+j_k*h_k:inf(I_k)+(j_k+1)*h_k-1|k\in 1..m]}(A)$$

array multicast *Finally, we can formalize an* array multicast *operation, which sends the same data to a group of processors. As an example, to say we send row* i *of a matrix* $A[l_k : h_k | k \in 1..2]$ *to each processor in row* i *of a communicator* C *defined on the same index set, we would write* $\mathbf{distr}_C^{\lambda i,j.[i:i][l_2:h_2]}(A)$.

Example 3 (Distributions in our running example) *Back to our running example. If* R *is our* 3×3 *communicator of processors, the* multicast of B over R *is*

$$\mathbf{distr}_R^{\lambda i,j.[2i:2i+1][0:11]}(B)$$

the multicast of A over R *is*

$$\mathbf{distr}_R^{\lambda i,j.[0:11][2j:2j+1]}(A)$$

and block scatter of C over R *is* $\mathbf{distr}_R^{\lambda i,j.[2i:2i+1][2j:2j+1]}(C)$

In what follows, we will often use the term *global array* to denote an array in the usual sense (so, a function from an index domain to a set of values), while we will use the term *distributed array* to mean an array that has been spread blockwise across a set of processors. Such a distributed array has the same name on all the processors, but each processor only owns a piece of the array, whose dimensions depend on the processor itself. To access a member of *distributed array*, we must specify a processor index $\vec{j}$ first, then an array index $\vec{i}$ *inside* the sub domain distributed on this processor, and then we get the final value. More formally

Definition 15 (Distributed array) *Given an index domain* $I = (\Pi_{i=1}^m D_i)$*, an index domain* $J = (\Pi_{i=1}^n E_i)$ *and a* dependent index distribution $gs :: J \to P(I)$*, a distributed array* A *is defined as a dependent function*

$$A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$$

so that for each index $\vec{j} \in J$*,* $A(\vec{j}) : gs(\vec{j}) \to V$ *returns a function from an index domain* $gs(\vec{j})$ *(sub domain of* I*) to value domain* V*. Note that,* $A(\vec{j})$ *is in fact a global array again.*

Coming back to the definition of *distribution operation*, a distribution operation **distr** takes a global array $(A : I \to V)$ as input and produces a distributed array $(\mathbf{distr}_{comm}^{gs}(A) : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V)$ with the arguments gs and C specified. This fits well to our common sense.

3.4.1 Remarkable identities over distributions

We identify now a few remarkable identities on generic array distributions (remember Notation 3).

Proposition 2 (Distributivity of distr and $\oplus$)

$$\bigoplus_{i\in I} \mathbf{distr}_{C_i}^{gs}(A) = \mathbf{distr}_{\bigoplus_{i\in I} C_i}^{gs}(A) \tag{2}$$

if all the $dom(C_i)$ *are conformant (ie, have the same index domain, Def. 9) and* $\bigoplus_{i\in I} C_i$ *is well defined (no superposition, all* C_i *are defined on different indexes).*

Proposition 3 (Absorption of injections)

$$proj_{\sigma,\vec{v}}(\mathbf{distr}^{gs}_{inj^{dom(C)}_{\sigma,\vec{v}}(D)}(A)) = \mathbf{distr}^{proj_{\sigma,\vec{v}}(gs)}_{D}(A) \tag{3}$$

$$\mathbf{distr}^{gs}_{inj^{dom(C)}_{\sigma,\vec{v}}(D)}(A) = inj^{dom(C)}_{\sigma,\vec{v}}(\mathbf{distr}^{proj_{\sigma,\vec{v}}(gs)}_{D}(A)) \tag{4}$$

Both propositions are established by an easy case analysis on the domain of definition of the two members of equations. Finally, we establish that if a communicator C is decomposed in non overlapping parts, distribution over parts can be done independently using an appropriate restriction of mapping gs.

Proposition 4 (Partition) *For $C : (\Pi_{i=1}^{n} D_i) \to P$, $\sigma : m \to n$ $(m \leq n)$*

$$\mathbf{distr}^{gs}_{C}(A) = \bigoplus_{\vec{v} \in dom_{\bar{\sigma}}(C)} inj^{dom(C)}_{\sigma,\vec{v}}(\mathbf{distr}^{proj_{\sigma,\vec{v}}(gs)}_{proj_{\sigma,\vec{v}}(C)}(A)) \tag{5}$$

where $proj_{\sigma,\vec{v}}(gs)(i_1,\ldots,i_m) = gs(a_1,\ldots,a_n)$ with $a_{\sigma(k)} = i_k$, $a_{\bar{\sigma}(k)} = \vec{v}[k]$

Proof.

$$\begin{aligned}
\mathbf{distr}^{gs}_{C}(A) &= \mathbf{distr}^{gs}_{\oplus_{\vec{v} \in dom_{\bar{\sigma}}(C)} inj^{dom(C)}_{\sigma,\vec{v}}(proj_{\sigma,\vec{v}}(C))}(A) \text{ by Eq. 1} \\
&= \bigoplus_{\vec{v} \in dom_{\bar{\sigma}}(C)} \mathbf{distr}^{gs}_{inj^{dom(C)}_{\sigma,\vec{v}}(proj_{\sigma,\vec{v}}(C))}(A) \text{ by Prop. 2} \\
&= \bigoplus_{\vec{v} \in dom_{\bar{\sigma}}(C)} inj^{dom(C)}_{\sigma,\vec{v}}(\mathbf{distr}^{proj_{\sigma,\vec{v}}(gs)}_{proj_{\sigma,\vec{v}}(C)}(A)) \text{ by Prop. 3} \qquad \square.
\end{aligned}$$

3.5 Array gathering

Data distributed over processors can be collected back by the *gather operation*:

Definition 16 (Gather over a communicator) *Given a communicator $C : J \to P$, an index domain I and an distributed array A defined over J, such that for each $\vec{j} \in J$, $A(\vec{j}) : gs(\vec{j}) \to V$ is a global array defined over a subset of I. We define the* gather *of A over C and I as $\mathbf{gath}^{I}_{C}(A) : I \to V$, such that for $\vec{i} \in I$*

$$\mathbf{gath}^{I}_{C}(A)(\vec{i}) = \begin{cases} A(\vec{j})(\vec{i}) \text{ if } \exists \text{ only one } \vec{j} \in J \text{ for which } A(\vec{j})(\vec{i}) \text{ is defined} \\ \perp \text{ otherwise} \end{cases}$$

It's quite clear that *gather* operation takes an input of distributed array $(A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V)$ and produces a global array $(\mathbf{gath}^{I}_{C}(A) : I \to V)$. Note that, gs here doesn't means the distributed array is generated by a concrete distribution operation $\mathbf{distr}^{gs}_{C}(A)$ (it is quite possible to be formed though a sequence of different distribution and gathering operations). As we mentioned in previous section, gs is just an pure function describing the association between each processor index and certain index domain corresponding, no matter such association is to be generated by some operation or somehow already formed.

3.5.1 Remarkable identities over gathering

We identify a few remarkable identities on distribution and gathering.

If we *scatter* an array $A : I \to V$ over a communicator C (ie, we partition A on processors without replications) and then gather it again on the same index domain we obtain the same array. That is

$$\mathbf{gath}^{I}_{C}(\mathbf{distr}^{gs}_{C}(A)) = A \tag{6}$$

if $cod(gs) \subseteq P(I)$ is a *partition* of I.

3.6 Block computations

Now that we have the combinators necessary to distribute and gather data to and from a processor grid, we turn in this section, to the specialized combinators that we provide in our calculus to perform computations on the blocks of data that have already been distributed on the processor grid. This set contains the usual `map`, `reduce` and `scan` operations, now working over blocks of data, together with a special combinator `btran` that allows to perform operations that actually change the dimensions of the blocks.

Definition 17 (Block map over a communicator) *Given a communicator $C : J \to P$, a distributed array $A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$ and a function $f : (L \to V) \to (L \to S)$, and $\forall \vec{j} \in J.\ adims(A(\vec{j})) = idims(L)$, we define the* block map *of A over C as* $\mathbf{bmap}_C^f(A) : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to S$

$$\mathbf{bmap}_C^f(A)(\vec{j}) = ashift^{abase(A(\vec{j}))}(\ f(\ ashift^{ibase(L)}(A(\vec{j})))\ )$$

The *bmap* operation takes an input of distributed array over a communicator and produces another distributed array. It's quite similar to the common *map* function, but the basic unit given to the function f is now a full block, so it allows to write programs that need access to all the element of the block to compute the result (like in the case of the Life game).

There are cases where we need not only to access the full block, but also to produce as a result a block of different dimensions. A typical example where this need arises is the taks of performing a reduce over a distributed array (for example, sum-up all the elements of the distributed array): in that case, one wants to first perform locally the reduction, that changes the dimension of the data, as it takes an array and produces an integer (the sum of the elements), and then a reduction among the processors to get the final result.

We introduce now the *btran* (block transformation) combinator, that allows to perform this first phase of a reduction.

Definition 18 (Block transform over a communicator) *Given a communicator $C : J \to P$, a distributed array $A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$, a function $f : (L \to V) \to (K \to S)$, $\forall \vec{j} \in J.\ adims(A(\vec{j})) = idims(L)$, we define the* block transform *operation of A over C as* $\mathbf{btran}_C^f(A) : \Pi\ \vec{j} \in J.\ gs'(\vec{j}) \to S$

$$\mathbf{btran}_C^f(A)(\vec{j}) = f(\ ashift^{ibase(L)}(A(\vec{j})))$$

After a *btran*, we can finalyze the reduce operation by means of a *bred* combinator.

Definition 19 (Block reduce over a communicator) *Given a communicator $C : J \to P$, an distributed array $A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$ and an associative binary operator $\odot : (L \to V) \to (L \to V) \to (L \to V)$, and $\forall \vec{j} \in J.\ adims(A) = idims(L)$, we define the* block reduce *operation of A over C with operator $\odot$ as* $\mathbf{bred}_C^{\odot}(A) : L \to V$ *such that*

$$\mathbf{bred}_C^{\odot}(A) = \bigodot_{\vec{j} \in J, A(\vec{j}) \neq \perp} ashift^{ibase(L)}(A(\vec{j}))$$

The *bred* function takes an input of distributed array and produces a global array. It's very similar to the common *reduce* function on elements except the basic unit is block.

Definition 20 (Block scan over a communicator) *Given a communicator $C : J \to P$, an distributed array $A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$ and an associative binary operator $\odot : (L \to V) \to (L \to V) \to (L \to V)$, and $\forall \vec{j} \in J.\ adims(A) = idims(L)$, we define the* block scan *operation of A over C with operator $\odot$ as* $\mathbf{bscan}_C^{\odot}(A) : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$ *such that*

$$\mathbf{bscan}_C^{\odot}(A)(\vec{k}) = ashift^{abase(A(\vec{k}))}(\bigodot_{\vec{j} \in J, A(\vec{j}) \neq \perp, \vec{j} \sqsubseteq \vec{k}} ashift^{ibase(L)}(A(\vec{j})))$$

where $\sqsubseteq$ is the lexicographic order on indexes.

The input of *bscan* operation is a distributed array and the result is also a distributed array of the same type.

3.6.1 Remarkable identities

Most well known equalities for map, reduce and scan [2, 11] hold in our setting. For instance, if $A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$, $f : (I \to V) \to (I \to S)$ and $g : (I \to S) \to (I \to T)$, we can formulate **bmap**() fusion as

$$\mathbf{bmap}_C^g(\mathbf{bmap}_C^f(A)) = \mathbf{bmap}_C^{f \circ g}(A) \tag{7}$$

3.7 Formalizing multicast in our running example

3.7.1 Multicast = scatter&broadcast

We can now use our model to express the collective operations composing second multicast strategy introduced for our running example.

Intuitively, a multicast operation induces an equivalence class on a communicator, two processors being equivalent if they receive the same data. Then, we can decompose it into a scatter operation to some representative processor in the equivalence class, followed by a set of broadcasts from the representative to the whole class. Let's start describing a columnwise multicast distribution of a matrix $A : [l_1 : h_1][l_2 : h_2] \to V$ onto a communicator $C : [l'_1 : h'_1][l'_2 : h'_2] \to P$ of the same dimensionality, with a block-size of s, expressed as the following distribution:

$$\mathbf{distr}_C^{\lambda i,j.[l_1:h_1][l_2+j*s:l_2+(j+1)*s-1]}(A)$$

We see clearly from the expression of the distribution that there is a constant part $[l_1 : h_1]$ that may give raise to a broadcast operation (all processors in the same column get the same slice of the original matrix). To make this explicit, we choose as the restriction function $\sigma : 1 \to 2$, defined as $\sigma(1) = 1$ (in general, we would chose a restriction function removing every index not used in the index expression of the multicast). Then, by definition, $\bar{\sigma} : 1 \to 2$ and $\bar{\sigma}(1) = 2$. Now, we can apply the partition identity, using σ and $\bar{\sigma}$, to obtain mechanically

$$\mathbf{distr}_C^{\lambda i,j.[l_1:h_1][j*s:(j+1)*s-1]}(A) = \bigoplus_{v \in [l'_2:h'_2]} inj_{\sigma,v}^{dom(C)}(\mathbf{distr}_{\lambda i \in [l'_1:h'_1].C[i][v]}^{\lambda i.[l_1:h_1][v*s:(v+1)*s-1]}(A))$$

The equality produces a superposition of distributions whose index function is *constant*, so we have decomposed the block scatter operation of A over C into a superposition of independent broadcast operations of disjoint subsets of the argument array A onto the independent communicators $\lambda i \in [l'_1 : h'_1].C[i][v]$. Now, we can easily pick a representative processor in each communicator by choosing a canonical index, for example the minimum, and write a block scatter distribution that would assign to each representative processor a different portion of A:

$$\begin{aligned}
&\mathbf{distr}_{proj_{\bar{\sigma},inf(dom_\sigma C)}(C)}^{proj_{\bar{\sigma},inf(dom_\sigma C)}(\lambda i,j.[l_1:h_1][j*s:(j+1)*s-1])}(A) \\
&= \mathbf{distr}_{proj_{\bar{\sigma},l'_1}(C)}^{proj_{\bar{\sigma},l'_1}(\lambda i,j.[l_1:h_1][j*s:(j+1)*s-1])}(A) \\
&= \mathbf{distr}_{\lambda j.C[l'_1][j]}^{\lambda j.[l_1:h_1][j*s:(j+1)*s-1]}(A)
\end{aligned}$$

and this last one is clearly a scatter operation (all communicator indexes are used in the index expression, and they select independent blocks of A). Now that we can describe scatter and broadcasts we need a calculus to compose them in a more complex strategy.

3.7.2 Multicast = scatter&gather

We now model the third multicast strategy introduced for our running example. We want to multicast columnwise the matrix $A : [l_1 : h_1][l_2 : h_2] \to V$ onto a communicator $C : [l'_1 : h'_1][l'_2 : h'_2] \to P$ of the same dimensionality. The idea here is

- scatter the matrix A on C using a block scatter strategy with block-size $s_1 \times s_2$:

$$\mathbf{distr}_C^{\lambda i,j.[l_1+i*s_1:l_1+(i+1)*s_1-1][l_2+j*s_2:l_2(j+1)*s_2-1]}(A)$$

- on each processor $C[i][j]$, gather all the blocks of A scattered on processors on the same column (ie, on communicator $\lambda i \in [l'_1 : h'_1].C[i][j]$) in order to have locally all the columns $A[l_1 : h_1][l_2+j*s_2 : l_2(j+1)*s_2-1]$, which is formalized by the following gather

$$\mathbf{gath}_{\lambda i \in [l'_1:h'_1].C[i][j]}^{[l_1:h_1][l_2+j*s_2:l_2+(j+1)*s_2-1]}(A')$$

  where A' (defined over $[l'_1 : h'_1][l'_2 : h'_2]$) is the array of arrays resulting from the block scatter operation.

Then the overall column multicast will require one gather for each processor. The following points are worth noticing.

gatherall all processors on the same column need to perform exactly the same gather (it does not depend on i), resulting in the same array $A[l_1 : h_1][l_2 + j * s_2 : l_2 + (j + 1) * s_2 - 1]$. In this case we can use a generalized version of gather (ie, `allgather` provided in MPI) which accounts for cases in which *all* processors in the communicator need to gather the same value. It's the same for another case in which *all* processors in the communicator need to do the same reduce. We will introduce suitable operations $\mathbf{allg}_C^I(A)$ and $\mathbf{allr}_C^I(A)$ in our calculus to account for this common optimization.

parallelism gather operations involving processors on different columns need not to be serialized. Thus, all gathers on different columns can be executed in parallel.

We will formalize all the steps after introducing our calculus in the following section.

4 A calculus for dense arrays

Using as semantical model the array operations introduced in Sec 3, we can now introduce a simple calculus that can describe formally the evolution over time of a distributed dense array. For the sake of brevity, we do not give the formal semantics of the full language (notably, to the standard functional part of it), but only of the special features modeling distributed data.

Definition 21 (Dense array calculus) *The language of the* dense array calculus *is composed of the following syntactic categories:*

Index Domains
idom ::= a cartesian product of integer intervals
Communicators
processors ::= a set of processors
comm ::= functions from indexes to processors
Array expressions
name ::= a string
index ::= a sequence of integers
aexp ::= name@(index,comm) | $[e|l_1 \leq i_1 \leq h_1, \ldots, l_n \leq i_n \leq h_n]$
Index distributions

gs ::= functions from indexes to index domains | $proj_{\sigma,index}(gs)$

Array operations

$$
\begin{aligned}
d ::= \ & \mathbf{distr}^{gs}_{comm}(aexp, name) \quad | \\
& \mathbf{gath}^{idom}_{comm}(name, index) \quad | \\
& \mathbf{bmap}^{function}_{comm}(name) \quad | \\
& \mathbf{bred}^{binop}_{comm}(name, index) \quad | \\
& \mathbf{bscan}^{binop}_{comm}(name) \quad | \\
& \mathbf{btran}^{function}_{comm}(name) \quad | \\
& \mathbf{allg}^{idom}_{comm}(name) \quad | \\
& \mathbf{allr}^{binop}_{comm}(name) \quad | \\
& d;d \quad | \\
& ||(d_1, \ldots, d_n)
\end{aligned}
$$

The *aexp*, which stands for a global array, can be expressed in two styles. The first style $name@(index, comm)$ presents the distributed part of *name* array on *index* processor under certain communicator *comm*; the second style directly gives a array function $e : Index \to Value$ with its domain defined as (or restricted to) $l_1 \leq i_1 \leq h_1, \ldots, l_n \leq i_n \leq h_n$.

Also notice that we purposely left unspecified the base functional language used to write functions like *gs*, *comm* or *binop*. The choice of this language is not essential for our purposes; we just assume some such language (with appropriate semantics) is given.

This calculus is given semantics in a denotational style [18] using the structures we have introduced in the previous section.

Definition 22 (Semantic domains) *We take as base semantic domains a set of processor, array names and integer index ranges. Then our state is simply a function associating an array to each processor and array name.*

$$
\begin{aligned}
Proc &= \textit{flat domain of processors} \\
Names &= \textit{flat domain of names} \\
Index &= \textstyle\bigcup_k N^k \\
State &= Proc \to Names \to Index \to Value
\end{aligned}
$$

We assume the usual auxiliary function $ext : S \to S \to S$ *to extend a state with new bindings.*

Note that we do not use the processor index but the processor itself as the element of a state, because it is possible to have more than one communicator in a system and under different communicators a processor usually has different indexes. For instance, we can do a gather operation to get an array under communicator A and then distribute it with another communicator B — It's obviously more powerful and convenient to have this style. So $Proc$ is independent but $ProcIndex$ is dependent (on communicator). The *State* can only be modelled as $Proc \to Names \to Index \to Value$, not $ProcIndex \to Names \to Index \to Value$. Meanwhile in all mathematical array manipulation defined in previous section, we use the $ProcIndex$ not the $Proc$, because only the index is operable while processor not. This doesn't matter because there holds a basic precondition that for each operation there is one and only one communicator for it, and in each communicator($Comm$) the processor index ($ProcIndex$) could be uniquely mapped to one processor ($Proc$). So we have $Proc = Comm(ProcIndex)$. To access array values with processor index under a certain communicator, we can just apply $State : Exp$ sequently to $Comm(ProcIndex)$ and $Name$:. In short, to describe(define) a state we use $Proc$, to access array we use $ProcIndex$ with current communicator.

Definition 23 (Semantic interpretation functions) *The semantic interpretation functions (Fig. 7) come in two flavors:* $\mathcal{E}[\![e]\!]s$ *associates a value to an expression* e *in a given state* s *(this value may be an integer, a function, etc.), while* $\mathcal{C}[\![d]\!]s$ *computes the new state* s' *which is* s *as modified by the execution of the operation* d*.*

$$\mathcal{E}[\![name@(index, comm)]\!]\mathrm{s} = s\ (\mathcal{E}[\![comm]\!]s\ index)\ name \quad (1)$$

$$\mathcal{E}[\![[e|l_1 \le i_1 \le h_1, \ldots, l_n \le i_n \le h_n]]\!]\mathrm{s} \quad (2)$$
$$= \lambda i_1 \ldots i_n.(\mathcal{E}[\![e]\!]s)(i_1)(i_2)\ldots(i_n)$$

$$\mathcal{E}[\![proj_{\sigma,index}(gs)]\!]\mathrm{s} = proj_{\mathcal{E}[\![\sigma]\!]s,index}(\mathcal{E}[\![gs]\!]s) \quad (3)$$

$$\mathcal{C}[\![\mathbf{distr}^{gs}_{comm}(aexp, name)]\!]\mathrm{s} = \text{let } a, g, c = \mathcal{E}[\![aexp]\!]\mathrm{s},\mathcal{E}[\![gs]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (4)$$
$$\text{in let } d = \mathbf{distr}^{g}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (\exists j.(c(j) = p)) \wedge (n = name)$$
$$\text{then } d\ (c\ p) \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![\mathbf{gath}^{idom}_{comm}(name, index)]\!]\mathrm{s} = \text{let } i, c = \mathcal{E}[\![idom]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (5)$$
$$\text{in let } a = \lambda j.\ s\ (c\ j)\ name$$
$$\text{in let } g = \mathbf{gath}^{i}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (p = c\ index) \wedge (n = name) \text{ then } g \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![\mathbf{bmap}^{function}_{comm}(name)]\!]\mathrm{s} = \text{let } f, c = \mathcal{E}[\![function]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (6)$$
$$\text{in let } a = \lambda j.\ s\ (c\ j)\ name$$
$$\text{in let } b = \mathbf{bmap}^{f}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (\exists j.(c(j) = p)) \wedge (n = name)$$
$$\text{then } b\ (c\ p) \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![\mathbf{bred}^{binop}_{comm}(name, index)]\!]\mathrm{s} = \text{let } b, c = \mathcal{E}[\![binop]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (7)$$
$$\text{in let } a = \lambda j.\ s(c\ j)\ name$$
$$\text{in let } r = \mathbf{bred}^{b}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (p = c\ index) \wedge (n = name) \text{ then } r \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![\mathbf{bscan}^{binop}_{comm}(name)]\!]\mathrm{s} = \text{let } b, c = \mathcal{E}[\![binop]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (8)$$
$$\text{in let } a = \lambda j.\ s\ (c\ j)\ name$$
$$\text{in let } sc = \mathbf{bscan}^{b}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (\exists j.(c(j) = p)) \wedge (n = name)$$
$$\text{then } sc\ (c\ p) \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![\mathbf{btran}^{function}_{comm}(name)]\!]\mathrm{s} = \text{let } f, c = \mathcal{E}[\![function]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (9)$$
$$\text{in let } a = \lambda j.\ s\ (c\ j)\ name$$
$$\text{in let } t = \mathbf{btran}^{f}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (\exists j.(c(j) = p)) \wedge (n = name)$$
$$\text{then } t\ (c\ p) \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![\mathbf{allg}^{idom}_{comm}(name)]\!]\mathrm{s} = \text{let } i, c = \mathcal{E}[\![idom]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (5a)$$
$$\text{in let } a = \lambda j.\ s\ (c\ j)\ name$$
$$\text{in let } g = \mathbf{gath}^{i}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (\exists j.(c(j) = p)) \wedge (n = name) \text{ then } g \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![\mathbf{allr}^{binop}_{comm}(name)]\!]\mathrm{s} = \text{let } b, c = \mathcal{E}[\![binop]\!]\mathrm{s},\mathcal{E}[\![comm]\!]\mathrm{s} \quad (7a)$$
$$\text{in let } a = \lambda j.\ s\ (c\ j)\ name$$
$$\text{in let } r = \mathbf{bred}^{b}_{c}(a)$$
$$\text{in let } diff = \lambda\ p, n.$$
$$\text{if } (\exists j.(c(j) = p)) \wedge (n = name) \text{ then } r \text{ else } \perp$$
$$\text{in } (ext\ s\ diff,\ diff)$$

$$\mathcal{C}[\![d_1; d_2]\!]\mathrm{s} = \text{let } s', diff' = \mathcal{C}[\![d_1]\!]\mathrm{s} \quad (10)$$
$$\text{in let } s'', diff'' = \mathcal{C}[\![d_2]\!]\mathrm{s'}$$
$$\text{in } (s'',\ ext\ diff'\ diff'')$$

$$\mathcal{C}[\![||(d_1, \ldots, d_n)]\!]\mathrm{s} = \text{let } s_i, diff_i = \mathcal{C}[\![d_i]\!]\mathrm{s} \quad (11)$$
$$\text{in } (ext\ s\ \bigoplus_{1\le i\le n} diff_i,\ \bigoplus_{1\le i\le n} diff_i)$$

Figure 7: Semantic rules

We now discuss briefly the meaning of each rule. Rule (1) gives the distributed part of array *name* on the *index* processor under certain communicator *comm*. Rule (2) extracts subarrays using block selection (remember Notation 1). $proj_{\sigma,index}(gs)$ "restricts" function gs to a subarray of the definition domain (Rule (3)).

The semantics of a distribution $\mathbf{distr}^{gs}_{comm}(aexp, name)$ (Rule (4)) modifies the state by distributing global array *aexp* to all processors in communicator *comm* according to distribution function gs. All the distributed parts will be stored on respective processors under the same array name *name*. A gather $\mathbf{gath}^{idom}_{comm}(name, index)$ collects all distributed parts of array *name* from processors in the communicator *comm* and then stores the comprised result to the processor of *index* under the same array *name* (Rule (5)). A block reduce $\mathbf{bred}^{binop}_{comm}(name, index)$ assigns the result of reducing distributed array of *name* via *binop* to array *name* on *index* (Rule (7)). $\mathbf{bmap}^{function}_{comm}(name)$, $\mathbf{bscan}^{binop}_{comm}(name)$ and $\mathbf{btran}^{function}_{comm}(name)$ assign to array of *name* on all processors in communicator *comm* the result of respective computation (Rules (6)(8)(9)).

There are many cases in which the result of a gather/reduce operation on a communicator C is needed by all processors in C. In this case, we can perform a gather followed by a broadcast operation on C. However, this is not the best solution, as it is usually possible to leave the result of a gather/reduce operation on all the nodes in C paying a small extra overhead in the implementation. Thus it is convenient to have two extra operations for collection: allgather and allreduce, which is exactly what MPI does. In our calculus, we introduce the two operations $\mathbf{allg}()$ and $\mathbf{allr}()$ (Rules $(5a)(7a)$) to account for this particular optimizations.

Finally, the composition of two operations (;) executes the two operations one after the other, while the parallel composition (||) *simultaneously* superposes the effect of operations (Rules (10)(11)).

4.1 Identities

Now we have the means of describing a state associated to the processors, and we can model the evolution over time of this state, so we can write identities involving the sequential composition of operations manipulating dense arrays.

Using the identities on arrays and distributions proven in the previous sections, we establish:

Theorem 5 (Distribution decomposition)

$$\mathbf{distr}^{gs}_{C}(A'@(index', C'), A) = \mathbf{distr}^{proj_{\bar{\sigma},\vec{w}}(gs)}_{proj_{\bar{\sigma},\vec{w}}(C)}(A'@(index', C'), A); \underset{\vec{v}\in dom_{\bar{\sigma}}(C)}{||} \mathbf{distr}^{proj_{\sigma,\vec{v}}(gs)}_{proj_{\sigma,\vec{v}}(C)}(A@(\vec{v}\uparrow_{\sigma}\vec{w}, C), A) \tag{8}$$

where $\vec{w}$ is a vector of values used, as described in the previous section, to pinpoint the set of canonical representative processors in the equivalence class, and $\vec{v}\uparrow_{\sigma}\vec{w}$ gives the index in C of the canonical representative processor having index $\vec{v}$ in $proj_{\sigma,\vec{v}}(C)$

The following identities can be easily proved from definitions:

- an allgather can be always seen as gather followed by a broadcast for any *index* that C *index* $\neq \perp$

$$\mathbf{allg}^{I}_{C}(A) = \mathbf{gath}^{I}_{C}(A, index); \mathbf{distr}^{\vec{\lambda j}.I}_{C}(A@(index, C), A) \tag{9}$$

- an allgather is equivalent to a set of parallel gather for array $A : I \to V$,

$$\underset{p\in codom(C)}{||} \mathbf{gath}^{I}_{C}(A) = \mathbf{allg}^{I}_{C}(A) \tag{10}$$

which follows trivially from definition

- a scatter followed by an allgather on a given communicator is equivalent to a broadcast on the same communicator
  for array $A : I \to V$,

$$\mathbf{distr}_C^{gs}(A'(index', C'), A); \mathbf{allg}_C^I(A) = \mathbf{distr}_C^{\vec{\lambda j}.I}(A'(index', C'), A) \tag{11}$$

  which follow directly from Eq. 10 and Eq. 6, if $codom(gs) \subseteq P(I)$ is a *partition* of I.

- parallel and sequential composition can be exchanged as follows (par-seq exchange)
  for any d_1, d_2, d_3, d_4 we have that

$$||\{(d_1; d_2), (d_3; d_4)]\}$$

  has the same semantics as

$$||\{d_1, d_3\}; ||\{d_2, d_4]\}$$

  Notice that the second case adds extra-synchronization in the implementation as both activities in the second parallel composition should wait until the first two have finished. Thus, we can always transform

$$||\{(d_1; d_2), (d_3; d_4)\} \Rightarrow ||\{d_1, d_3\}; ||\{d_2, d_4]\} \tag{12}$$

  converse transformation can only be done if pairs (d_1,d_4) and (d_2,d_3) affect disjoint parts of the status.

4.2 Running example: proving equivalences

We can now formalize distributions in our running example (Sec. 3.7.1) and prove their semantic equivalence.

4.2.1 Multicast = scatter&broadcast

We can now state formally that the multicast operation can be expressed as the *sequence* of the scatter operations computed there and a set of broadcasts, originating from the canonical representative of each equivalence class, sending out the local slice of A received from the scatter operation:

$$\mathbf{distr}_C^{\lambda i,j.[l_1:h_1][l_2+j*s:l_2+(j+1)*s-1]}(A'@(index', C'), A) =$$

$$\mathbf{distr}_{\lambda j \in [l_2':h_2'].C[l_1][j]}^{\lambda j.[l_1:h_1][l_2+j*s:l_2+(j+1)*s-1]}(A'@(index', C'), A);$$

$$||_{v \in [l_2':h_2']} \mathbf{distr}_{\lambda i \in [l_1':h_1'].C[i][v]}^{\lambda i.[l_1:h_1][l_2+v*s:l_2+(v+1)*s-1]}(A@([l_1][v], C), A)$$

Thus, the equivalence between this strategy and direct send follows directly from Theorem 5.

4.2.2 Multicast = scatter&allgather

We can now state formally that the multicast operation can be expressed as the *sequence* of a scatter block distribution plus a set of parallel allgather:

$$\mathbf{distr}_C^{\lambda i,j.[l_1+i*s_1:l_1+(i+1)*s_1-1][l_2+j*s_2:l_2+(j+1)*s_2-1]}(A'@(index', C'), A);$$

$$||_{j \in [l_2':h_2']} \mathbf{allg}_{\lambda i \in [l_1':h_1'].C[i][j]}^{[l_1:h_1][l_2+j*s_2:l_2+(j+1)*s_2-1]}(A)$$

We now prove the equivalence of this strategy with the scatter plus broadcast strategy. Distribution function $\lambda i, j.[l_1+i*s_1 : l_1+(i+1)*s_1-1][l_2+j*s_2 :$

$l_2+(j+1)*s_2-1]$ is a partition of index domain $[l_1:h_1][l_2:h_2]$. So this distribution is a scatter operation. We can decompose a scatter operation in a group of scatters on disjoint parts of the communicator. Let us consider communicators $\lambda i \in [l'_1:h'_1].C[i][v]$, $v \in [l'_2:h'_2]$ which group all processors in the same column. These form a partition of communicator C. For each $v \in [l'_2:h'_2]$, $proj_{\sigma,v}(\lambda i, j.[l_1+i*s_1 : l_1+(i+1)*s_1-1][l_2+j*s_2 : l_2+(j+1)*s_2-1])$, $\sigma : 1 \to 2$, $\sigma(1) = 1$ is a partition of communicator $\lambda i \in [l'_1:h'_1].C[i][v]$, so we can decompose our scatter operation as a first scatter on a subset of the communicator followed by a set of parallel scatter on the rest [16]. In particular, we can first scatter our matrix onto the same set of canonical representative used by scatter&broadcast distribution (that is on row $\lambda j.C[l'_1][j]$) and then a set of parallel scatter along the columns, formally

$$\mathbf{distr}^{\lambda j.[l_1:h_1][l_2+j*s_2:l_2+(j+1)*s_2-1]}_{\lambda j.C[l'_1][j]}(A'@(index', C'), A);$$

$$||_{j\in[l'_2:h'_2]}\ \mathbf{distr}^{\lambda i.[l_1+i*s_1:l_1+(i+1)*s_1-1][l_2+j*s_2:l_2+(j+1)*s_2-1]]}_{\lambda i\in[l'_1:h'_1].C[i][j]}(A@([l'_1][j], C), A);$$

$$||_{j\in[l'_2:h'_2]}\ \mathbf{allg}^{[l_1:h_1][l_2+j*s_2:l_2(j+1)*s_2-1]}_{\lambda i\in[l'_1:h'_1].C[i][j]}(A)$$

we can then apply par-seq exchange

$$\mathbf{distr}^{\lambda j.[l_1:h_1][l_2+j*s_2:l_2+(j+1)*s_2-1]}_{\lambda j.C[l'_1][j]}(A'@(index', C'), A);$$

$$||_{j\in[l'_2:h'_2]}\ \{\mathbf{distr}^{\lambda i.[l_1+i*s_1:l_1+(i+1)*s_1-1][l_2+j*s_2:l_2+(j+1)*s_2-1]]}_{\lambda i\in[l'_1:h'_1].C[i][j]}(A@([l'_1][j], C), A);$$

$$\mathbf{allg}^{[l_1:h_1][l_2+j*s_2:l_2+(j+1)*s_2-1]}_{\lambda i\in[l'_1:h'_1].C[i][j]}(A)\}$$

and from Eq. 11 we obtain a set of parallel broadcast as in the previous distribution.

Thus, we have formalized our three strategies and proved they are all equivalent from a semantic point of view.

After deriving costs for operations in the following section, we will compare their properties with respect to performance.

5 Cost models

We want now to to associate a cost to array operations, so we first fix some details on the execution model. We assume a set of processors arranged as a Cartesian communicator and communicating via message passing. We consider two message passing styles: *bulk synchronous* (such as in BSP[22]) and *asynchronous*, (such as in MPI). Since the interaction style influences the cost model, we discuss the two cases in two separate subsections. In particular, Section 5.1 discusses the costs in BSP and Section 5.2 discusses the costs in the asynchronous case.

In both cases we first cost single operations, then discuss how to combine costs to evaluate composition of operations via the sequential/parallel operators. Finally, we apply costs to our running example and give a quantitative comparison of multicast strategies.

5.1 A cost model based on BSP

In BSP, computation is organized in supersteps in which processors first perform local computation and then exchange data in a global communication step. BSP's cost model assumes that the bottleneck in communication performance is at the processors, rather than in the network itself. If processor i transmits h_i words during a communication phase, the total time for the phase is close to $g * \max_i\ h_i$. To give the cost of a whole superstep, two more parameters are used: s which models the computation units executed by a

distribution	fan-out source node	max fan-in
broadcast	$\|A\| * N$	$\|A\|$
block scatter	$\|A\|$	$\prod_k h_k$
multicast	$\|A\| * M$	$\|A\|/(N/M)$

Table 1: Where N is the number of processors, M is the cardinality of the group of processors which get the same data.

processor in the unit of time, and l which models the cost of a global synchronization barrier to inform all the processors that the superstep has ended. A superstep costs $s * \max_i c_i + g * \max_i \ h_i + l$, where c_i are the computation units executed at processor i. The BSP model has been investigated experimentally and has been proved accurate to within a few percent across a range of architectures and applications [20, 12].

The value of h_i depends on the communication ability at each processor. During a communication step, we call fan-in f_i^{in} the number of words received by processor i and fan-out f_i^{out} the number of words sent. If i can both send and receive simultaneously $h_i = \max\{f_i^{in}, f_i^{out}\}$, otherwise $h_i = f_i^{in} + f_i^{out}$. Here, we assume bidirectional communication ability.

5.1.1 Costing distributions

The cost of executing a BSP superstep which consists of a single distribution operation $\mathbf{distr}_C^{gs}(A)$ can be computed from the fan-in and the fan-out of each processor in C according to gs. For the sake of simplicity, we assume the size of each element in A to be one word. If this is not the case, it suffices to multiply the fan-in, and fan-out for the size in words of each element. Since A usually resides on a single processor P, the maximum fan-out is always given by the fan-out at P. f_P^{out} is given by the number of elements the domain of A ($\#dom(A)$) multiplied by the *replication factor* that is the number of data sent to more than one processor.

If $gs : J \to \mathcal{P}(I)$, each processor corresponds to an index in J. Given an index domain $I = (\Pi_{i=1}^m D_i)$, and an index domain $J = (\Pi_{i=1}^n E_i)$,

$$gs(j_1, \ldots, j_n) = \prod_{k=1}^{m} I_k \subseteq I$$

the replication factor is given by the size of the disjoint union of the index domains spanned by the distribution (restricted to the domain of definition of A), and dividing it by the effective size of (the domain of definition of) A

$$r_{gs} = \frac{\#\biguplus_{j \in J}(gs(j) \cap dom(A))}{\#dom(A)}$$

thus the maximum fan-out is $|A| * r_{gs}$. On the other hand, the fan-in at processor j depends on the set $gs(j) \subseteq I$ and in particular it is equal the number of A elements in $gs(j)$. In Table 1, we give fan-in/out for some common distributions.

Thus, the cost of a superstep only distributing data according to $\mathbf{distr}_C^{gs}(A)$ is

$$T(\mathbf{distr}_C^{gs}(A)) = \max\{\max_i\{\#(gs(i) \cap dom(A)\}, |A| * r_{gs}\} * g + l \qquad (13)$$

we take the max between the fan-out at P and the max fan-in at destinations.

5.1.2 Costing collections

The cost of a gather operation ($\mathbf{gath}_C^I(A)$) is given by the fan-in at destination node, which is the number of elements of A with index in I ($|\#dom(A) \cap I|$).

Thus,

$$T(\mathbf{gath}_C^I(A)) = |\#dom(A) \cap I| * g + l$$

Notice that gather and allgather have the very same cost as gathering in parallel on all nodes does not increase the maximum fan-in at destination.

Consider now reduce and allreduce operations. Let $t_\odot(n)$ be the cost of computing $\odot$ on an array of size n on a single processor. There are several strategies to compute reduce in the literature. Two of the most common are:

binary tree we can emulate a binary tree in $\log p$ supersteps (where $C : J \to P$, $p = |J|$ is the number of processors available in communicator C), and at each step half of the processes receive data from the other half and reduce them locally.

gather and bmap we can gather all data in a single processor and reduce all of them locally

In our calculus, the second strategy corresponds to a different combination of operations and can be costed in a different way. Thus, we give costs to block reduce assuming a binary tree execution strategy:

$$T(\mathbf{bred}_C^\odot(A)) = \log p * (t_\odot(n) + n * g + l)$$

where n is the size of operands of $\odot$ (if $A : \Pi\ \vec{j} \in J.\ gs(\vec{j}) \to V$, then $n = \max_{\vec{j} \in J} \#dom(A(\vec{j})) \leq |I|$). Since at each step the number of active processors halves, we can exploit idle processors and compute all reduce on all processors. Thus, costs of reduce and allreduce are again the same.

5.1.5 Running example: costing multicasts

We can now compare the cost of the three distribution strategies of the example in (Sec. 3.7.1). We assume we are multicasting an array A on $N \times N$ processors with size of replication groups equal to N.

direct send implements the multicast with N^2 separate sends each one sending out $|A|/N$ elements. Cost is $|A| * N * g + l$

scatter&broadcast first scatters A on N processors and then broadcasts in parallel each block on N groups of N processors. Cost is

$$|A| * g + l + \frac{|A|}{N} * N * g + l = 2 * |A| * g + 2 * l$$

scatter&allgather first scatters A on all available processors and the performs N concurrent allgather each gathering $|A|/N$ data. Cost is

$$|A| * g + l + \frac{|A|}{N} * g + l$$

so we can conclude that scatter&broadcast is better than direct send whenever $(N-2) * |A| * g \geq l$, while scatter&allgather always performs better than scatter&broadcast. Thus, we have managed to reason on semantically equivalent strategies in the BSP cost scenario.

5.2 A cost model using MPI communication style

Here, we assume a processor network in which any two processors can send messages of size m to each other simultaneously in time $t_s + m * t_w$, where t_s is the time to start-up a communication and t_w is the per-word transfer time. A processor is allowed to send/receive messages on only one of its links at a time. We also assume that local copies take negligible time.

In this model, processors can exchange information any time, possibly blocking if the partner is not ready. This model, although quite simplistic, it has been widely used to compute costs for programs using communication libraries such as MPI/PVM on a wide range of parallel machines[10]. It must be noted, however, that assuming fixed values for t_s and t_w it is not realistic in extremely dynamic contexts such as heterogenous, non dedicated platforms (such us those used for GRiD computing).

5.2.1 Costing distributions

Consider the cost of a single distribution $\mathbf{distr}_C^{gs}(A)$. From gs we compute the replication factor r_{gs} as in Eq. 13. Then, the source node sends a sequence of $p-1$ messages (p is the number of processors in the communicator, including the source), thus cost is

$$T(\mathbf{distr}_C^{gs}(A)) = (p-1) * t_s + (|A| * r_{gs}) * t_w$$

as we need to startup $p-1$ communications, and need to transfer globally $|A| * r_{gs}$ words of data.

5.2.2 Costing collections

The cost of a gather is:

$$T(\mathbf{gath}_C^I(A)) = |\#dom(A) \cap I| * n * t_w + (p-1) * t_s$$

as we need to establish $p-1$ communications to transfer $|\#dom(A) \cap I|$ data of n words each. The cost of allgather is the same as a gather operation, because we need to exchange $(p-1) * (p-1)$ messages and at each step $p/2$ pairs of processors can exchange data in a bidirectional way. Regarding block reduce, the cost is:

$$T(\mathbf{bred}_C^{\odot}(A)) = \log p * (t_{\odot}(n) + n * t_w)$$

5.2.3 Costing computations

The cost of bmap is the same computed for BSP (it does not include communications!)

$$T(\mathbf{bmap}_C^f(A)) = t_f(\max_{\vec{j}\in J} \#dom(A(\vec{j})))$$

while bscan is

$$T(\mathbf{bscan}_C^{\odot}(A)) = \log p * (3 * t_{\odot}(n) + n * t_w)$$

5.2.4 Costing compositions

The cost of a sequential composition $d_1; d_2$ is $T(d_1) + T(d_2)$, if we assume d_1 must be completed before starting d_2. If for specific pairs of operations, part of the computation of d_2 can start before d_1 has finished, we should add appropriate cost rules to our model. The cost of a parallel composition $d_1||d_2$ is $\max\{T(d_1), T(d_2)\}$, as our simple model assumes communications never collide.

5.2.5 Running example: costing multicasts

Again, we compare the costs of the three distribution strategies in Section 3.7.1 assuming we are multicasting an array A on $N \times N$ processors with size of replication groups equal to N.

direct send implements the multicast with N^2 separate sends each one sending out $|A|/N$ elements. Cost is $N^2 * t_s + |A| * N * t_w$

scatter&broadcast first scatters A on N processors and then broadcasts in parallel each block on N groups of N processors. Cost is

$$2 * (N-1) * t_s + \frac{|A|}{N}(2 * N - 1) * t_w$$

scatter&allgather first scatters A on all available processors and the performs N concurrent allgather each gathering $|A|/N$ data. Cost is

$$(N^2 + N - 1) * t_s + \frac{|A|}{N}(N+1) * t_w$$

so under this cost model scatter&allgather is worse than scatter&broadcast when t_s is sufficiently larger than t_w. Direct send in turn is worse than the other two in any case.

6 More examples: broadcast and scatter

In this section, we apply our calculus to study how a broadcast on a communicator C can be decomposed in a sequence of broadcast on 'smaller' (ie, lower size) communicators forming a partition of C. A similar result is then discussed for the scatter operation. For both decompositions we prove the semantic equivalence and compare performance costs according to the BSP model.

6.1 Broadcast = broadcast + broadcast

Consider a communicator C and a partition $C_1, \ldots, C_k$ of C. A broadcast operation on C can be decomposed in a first broadcast from the root to a set of representative processors $P = \{p_1, \ldots, p_k\}$ ($p_i \in C_i$) and then from each representative p_i to the C_i it belongs to (see Figure 8 for an example in two dimensions).

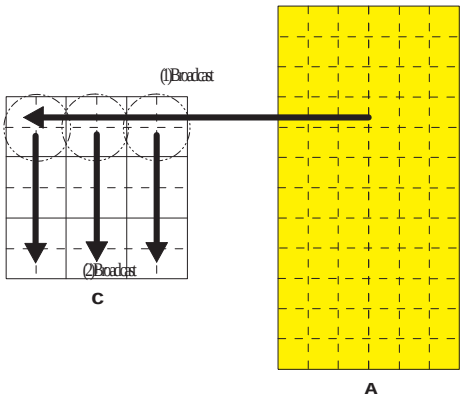


Figure 8: Decomposition of a broadcast in 'broadcast + broadcast'

Similar to the proof of multicast strategy in section 3.7.1, we can formalize this strategy in two dimensions according to proposition 4 as follows:

$$\mathbf{distr}_{C}^{\lambda i,j.[l_1:h_1][l_2:h_2]}(A) = \bigoplus_{v\in[l'_2:h'_2]} inj_{\sigma,v}^{dom(C)}(\mathbf{distr}_{\lambda i\in[l'_1:h'_1].C[i][v]}^{\lambda i.[l_1:h_1][l_2:h_2]}(A))$$

We partition the initial communicator by columns($\sigma : 1 \to 2$ and $\sigma(1) = 1$), inside of which is obviously a set of independent broadcasts of the whole array A. Choosing a canonical index to get the distribution for representatives of each equivalent class, we get

$$\begin{aligned}
&\mathbf{distr}_{proj_{\bar{\sigma},inf(dom_\sigma C)}(C)}^{proj_{\bar{\sigma},inf(dom_\sigma C)}(\lambda i,j.[l_1:h_1][l_2:h_2])}(A)\\
&= \mathbf{distr}_{proj_{\bar{\sigma},l'_1}(C)}^{proj_{\bar{\sigma},l'_1}(\lambda i,j.[l_1:h_1][l_2:h_2])}(A)\\
&= \mathbf{distr}_{\lambda j.C[l'_1][j]}^{\lambda j.[l_1:h_1][l_2:h_2]}(A)
\end{aligned}$$

Clearly, this one is also a broadcast operation of full array A, which proves the semantic equivalence of the two strategies.

Discussing BSP costs Implementing a broadcast of array A on $N \times N$ processors : (1) a one step array broadcast and (2) a two step strategy which first broadcast A on N processors and then broadcast in parallel each block on N groups of N processors. The cost of (1) is $|A| * N * N * g + l$, and the cost of (2) is $|A| * N * g + l + |A| * N * g + l = 2 * |A| * N * g + 2 * l$ So when $(N - 2) * N * |A| * g \geq l$, the two step strategy is more economic, which is obviously the most common case.

6.2 Scatter = scatter + scatter

We can apply the same strategy discussed in the previous section to the decomposition of a scatter operation on partition of a communicator C (see Figure 9).

The same array A with a scatter distribution would be written as

$$\mathbf{distr}_{C}^{\lambda i,j.[i*p:(i+1)*p-1][j*s:(j+1)*s-1]}(A)$$

Then we have

$$\begin{aligned}
&\mathbf{distr}_{C}^{\lambda i,j.[i*p:(i+1)*p-1][j*s:(j+1)*s-1]}(A)\\
&= \bigoplus_{v\in[l'_2:h'_2]} inj_{\sigma,v}^{dom(C)}(\mathbf{distr}_{\lambda i\in[l'_1:h'_1].C[i][v]}^{\lambda i.[i*p:(i+1)*p-1][v*s:(v+1)*s-1]}(A))
\end{aligned}$$

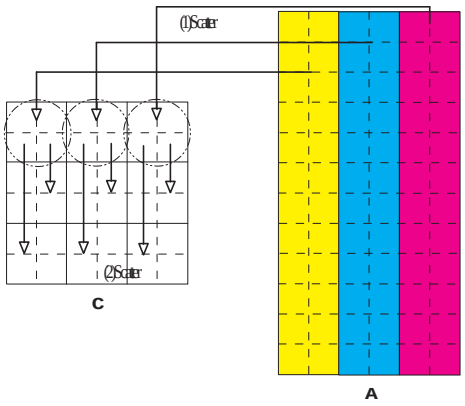


Figure 9: Decomposition of a scatter in 'scatter + scatter'

According to the indexes expression $\lambda i.[i{*}p : (i{+}1){*}p{-}1][v{*}s : (v{+}1){*}s{-}1]$ we know, each distribution (for each value of v) is a independent scatter operation on disjoint subset of array A. Now we simply pick the minimum index in each communicator $C[i][v]$ and suppose all data distributed to $C[i][v](i \in [l'_1 : h'_1])$ is distributed to the representative first. Now we can get this distribution as

$$\begin{aligned}
&\mathbf{distr}^{proj_{\bar{\sigma},inf(dom_\sigma C)}(\lambda i,j.[l'_1{*}p{:}h'_1{*}p{-}1][j{*}s{:}(j{+}1){*}s{-}1])}_{proj_{\bar{\sigma},inf(dom_\sigma C)}(C)}(A)\\
&= \quad \mathbf{distr}^{proj_{\bar{\sigma},l'_1}(\lambda i,j.[l_1:h_1][j{*}s:(j{+}1){*}s{-}1])}_{proj_{\bar{\sigma},l'_1}(C)}(A)\\
&= \quad \mathbf{distr}^{\lambda j.[l_1:h_1][j{*}s:(j{+}1){*}s{-}1]}_{\lambda j.C[l'_1][j]}(A)
\end{aligned}$$

From the indexes we can see it is also a disjoint scatter. Practically, the array scatters data partitions to a set of representatives, then each representative scatters to the whole subclass.

Discussing BSP costs Implementing a scatter of array A on $N \times N$ processors : (1) a one step array scatter and (2) a two step strategy which first scatter A on N processors and then scatter in parallel each block on N groups of N processors. The cost of (1) is $|A| * g + l$, and the cost of (2) is

$$|A| * g + l + \frac{|A|}{N} * g + l = |A| * g * (1 + \frac{1}{N}) + 2 * l$$

Obviously, the two step strategy always costs more than the one step scatter.

7 Related work

The goal of our calculus is to give a sound basis to experienced program developers for their design choices in implementing applications or supports working on multidimensional dense arrays, with an arbitrary number of dimensions. Thus, we do not consider automatic optimization of programs working on dense arrays nor we want to derive such programs automatically from some sequential/functional description.

This is why, we believe, we did not really find a suitable calculus ready for us in the vast literature dedicated to the problem of modeling the geometry of data, which has been addressed by several researchers in the field of automatic parallelization techniques[1], optimization and transformation of data parallel programs[24, 23, 4, 15].

On the other hand, formalisms aimed at a the systematic development of parallel programs using arrays, tend to limit the dimensionality (usually to 1,2) and/or do not address the problem of associating costs to operations [2, 17, 14].

In the following, we briefly recall a few previous works which are more closely related to ours. The PEI formalism [24] extends the model proposed in Crystal [4] and models multidimensional arrays using data fields. A data field is a function defined on a subcube in Z^n (its *index domain*). Data distributions can be expressed using three operations: *change of base*, which modifies the index domain of a data field (leaving the multiset of values unchanged), *geometrical operations*, which rearrange data, and *functional operations*, which apply the same function to all elements in a data field. The authors introduce a set of notable equivalences among combinations of operations. The idea is to develop parallel programs starting from an initial (possibly inefficient) representation, and rewriting it using semantic-preserving transformation rules until a 'satisfactory' implementation of the original idea is found. Although geometric operations of PEI are rather similar to our distributions, the semantic model we propose allows a more accurate representation of data distribution on actual processors. In particular, we are able to model hierarchical subsets of processors (communicators) much in the style of actual communication libraries such as MPI. This makes feasible to compute accurate costs for distributions and to take quantitative decisions.

D.06ur distr i

December 1994.

[2] R.S. Bird. Lectures in constructive functional programming. In *Constructive Methods in Computer Science*, volume F-55 of *NATO ASI*, pages 151–216. Springer-Verlag, 1989.

[3] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. Technical report, University of Tennessee, 1996.

[4] M. Chen, Y. Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308. ACM Press, 1991.

[5] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia*, pages 120–127. IEEE Computer Society Press, 1992.

[6] Roberto Di Cosmo and Susanna Pelagatti. A cost calculus for dense array distributions. In *Proc. of HLPP 2003*, 2003. To appear on *Parallel Processing Letters*.

[7] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the ocamlp3l experiment. *The ML Workshop*, 1998.

[8] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103. Springer-Verlag:Berlin, 1996.

[9] S.J. Fink, S.R. Kohn, and S.B. Baden. Efficient run-time support for irregular block-structured applications. *J. Parallel and Distributed Computing*, 50(1-2):61–8, April-May 1998.

[10] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, Reading, Mass., 1995.

[11] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In *13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99)*, pages 492–499, 1999.

[12] J.M.D. Hill, S.R. Donaldson, and D.B. Skillicorn. Stability of communication performance in practice: from to cray t3e to networks of workstations. Technical Report PRG-TR-33-97, Oxford University computing Laboratory, October 1997.

[13] Jonathan M. D. Hill. Collective communications in the Oxford BSP toolset. Note, part of the `bsplib` documentation., 1997.

[14] C.B. Jay, M.I. Cole, M. Sekanina, and P.A. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of Euro-Par'97 Parallel Processing*, volume 1300 of *LNCS*, pages 650–661. Springer, 1997.

[15] R. Lechtchinsky, M. Chackravarty, and G. Keller. Costing nested array codes. *Parallel Processing Letters*, 12(2):249–266, 2002.

[16] Zheng Li. Efficient implementation of MAP skeleton for the OcamlP3L system. DEA Report, University of PARIS VII, July 2003.

[17] Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.

[18] Peter D. Mosses. *Handbook of theoretical computer science*, volume B, chapter Denotational Semantics. MIT Press, Cambdridge, MA, 1991.

[19] M.J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, New York, 1994.

[20] J. Reed, K. Parrot, and T. Lanfear. Portability, predictability and performance for parallel computing: BSP in practice. *Concurrency Practice and Experience*, 8(10):799–812, December 1996.

[21] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. — Ninf — : Network based information library for globally high performance computing. In *Proc. of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe*, February 1996.

[22] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about bsp. *Scientific Programming*, 6(3):249–274, 1997.

[23] M. Südholt. Data distribution algebras: a formal basis for programming using skeletons. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 19–38. North-Holland, June 1994.

[24] E. Violard, S. Genaud, and G.-R. Perrin. Refinement of data-parallel programs in PEI. In R. Bird and L. Meertens, editors, *IFIP Working Conference on Algorithmic Language and Calculi*. Chapman & Hall, February 1997.

[25] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.