A Dimension Independent General Partitioning Algorithm to support HPF (re)distribution directives.

Haleh Najafzadeh, Lenore Mullin
zadeh@cs.albany.edu,lenore@cs.albany.edu
Department of Computer Science
University at Albany, SUNYA

Abstract

A General Partitioning Algorithm can provide a powerful way of computing local and global addressing for different distributions on different processor topologies. Therefore, it can be used to design an efficient algorithm for run-time (re)distributions in array based languages such as High Performance Fortran(HPF). Efficient redistributions are essential in many compute and communication intensive algorithms such as the Fast Fourier Transform.

1 Introduction

High performance Fortran (HPF) was designed to provide a portable extension to Fortran 90 for writing data parallel applications.It includes features (directives) for mapping data to parallel processors, specifying data parallel applications, and methods for interfacing HPF programs to other programming paradigms. Directives are special Fortran comments that suggest implementation strategies or assert facts about a program to the compiler. Align is an HPF directive that describes how two arrays "line up" together. Distribute is an HPF directive that specifies a mapping of data objects to abstract processors in a processor arrangement. The *distribution* step of the HPF model technically applies to the template of an array. Distribution partitions the template among a set of abstract processors according to a given pattern. The patterns can be one of the following:

$$star, block, cyclic, block(n), cyclic(n)$$

A template is simply an abstract space of indexed positions; it can be considered as an "array of nothings". A template may be used as an abstract align-target that may then be distributed. The combination of alignment(from arrays to templates) and distribution(from templates to processors) determines the relationship of an array to the processors,this relationship is called *mapping* of an array. *Redistribute* and *realign* are HPF directives similar to the distribute and align directives but are considered executable. An array or template may be redistributed or realigned at any time, provided it has been declared DYNAMIC.
Many factors play a role in the choice of a partitioning scheme. The communication patterns of the computation may favor one manner of partitioning over another. For example, matrix multiplication often performs best with block partitioning, but LU decomposition often performs best with a block-cyclic(cyclic(m)) partitioning scheme [6]. In some algorithms like the multidimensional Fast Fourier Transform(FFT) [21], the Alternative Direction Implicit(ADI) method for solving two-dimensional diffusion equations, redistributions are necessary for some of the stages of the computation in order to minimize the communication overhead. In this paper we introduce a dimension independent general partitioning algorithm, which is used to calculate the local/global addresses efficiently at the time of redistribution. Redistribution involves two basic issues, (a) an efficient local/global address calculations, and (b) an efficient communications between processors. Communications costs involved in redistributions will be considered in future research. The general partitioning algorithm is designed using Psi Calculus methods. The algorithms in this paper assume that arrays are row major [1].

Section 2 surveys the literature on the distributions and redistributions problem.A brief overview of Psi Calculus based on [1] can be found in section 3. Section 4 contains the design of the algorithm and section 5 explains the algorithm first in a high level English language based format and then based on Psi Calculus methods. Section 6 discusses the implementation issues. Section 7 introduces a redistribution algorithm based on our General Partitioning Algorithm. Section 8 includes the conclusion.

2 Literature Overview

We briefly survey the literature on the distributions and redistributions problem.A brief overview of Psi Calculus based on [1] can be found in section 3. [1] defines formal methods for representing, creating, accessing, and transforming arrays abstractly and in a general manner.

McMahon [7], developed a dimension independent algorithm for block distribution using Psi Calculus. The work in this paper extends that work in order to find a general and dimension independent distribution algorithm.

Anderson, Amarasinghe and Lam [11] introduce an optimization algorithm which consists of 2 steps. First choosing the parallelization and computation assignment such that synchronization and data sharing are minimized. The second step then restructures the layout of the data in the shared address space with an algorithm that is based on a

[1] ordering is arbitrary [13]

new data transformation framework.Throughout this transformation the arrays are restructured so that all the data accessed by the same processor are contiguous in the shared address space. Our work is very much similar to what they do for the data transformation.We also predict the indices which will be contiguous to each other after distributions or redistributions . But we approach the entire problem from a different view using Psi Calculus methods. Also we apply the final results to determine efficient local/global address calculations that can be used for efficient redistributions. Hence the mathematics used to formulate and transfer a problem to a programming model employs one unifying method, the Psi Calculus and its associated algebra(MOA).

Ashwath Thirumalai and J.Ramanujam [10] present efficient algorithms for generating local memory access patterns for the various processors (node code) given the alignment of arrays to a template and the distribution of the template onto the processors.This solution is based on viewing the access sequence as an integer lattice, and uses closed form expressions for a suitable set of basis vectors for the lattice. Efficient code is generated at compile time that determines the access pattern with negligible runtime overhead.Similarly our general partitioning algorithm can help us compute the local and global addresses of an array distributed over processors simply by applying a mathematics of arrays(Ψ Calculus) which looks at any data or array as contiguous indices in memory. Then applying Gamma, and GammaPrime functions we can capture the local indices on each processor or their relative global offset with respect to the shape of the original array before distribution.

Walker and Otto [12] show how to schedule messages in order to change the array distribution from cyclic(r) on a P-processor linear grid to cyclic(kr) on the same grid. Desprez and Dongarra [14] extend this idea in order to solve the general redistribution problem; that is, moving from a cyclic(r) distribution on a P-processor grid to a cyclic(s) distribution on a Q-processor grid. Our algorithm is also capable of solving redistribution problems when moving from cyclic(r) distribution on a P-processor grid to a cyclic(s) distribution on a Q-processor grid is concerned.

Similarly Chung,Hsu and Bai [15] have presented a basic-cycle calculation technique to efficiently perform block-cyclic(s) to block-cyclic(t) redistribution. Their work like [12] assumes that the source and destination processor sets are the same. Also their algorithm only supports one-dimensional arrays.

In [8] a new approach "strip mining " is introduced for redistributing arrays, where portions of array elements were redistributed in sequence in order to overlap the communication overhead with the following computation . In [9] a spiral mapping of formal processors on to actual processors is introduced that reduces communication overhead for array redistribution by translating the global communication to the local communication in a certain processor group.

[16] and [17] describe efficient algorithms for block(m) to cyclic, cyclic to block(m) and the general cyclic(x) to cyclic(y) type distributions. Their algorithms try to minimize address calculation and communication, and make good use of the cache. They introduce two algorithms (GCD and LCM methods) in order to reduce the overhead due to address calculations for every single index, when y=kx. When there is no relationship between y and x, calculations must be done for every single indices on both send and receive sides. They have introduced a two-phase method to perform the general case. For now our algorithm simply does the computation for every single index, but as mentioned in this paper, this is not the most efficient way to build it. We will reduce this further to start, stop and stride calculations using the Psi correspondence theorem [3]. This is only possible when the memory access patterns that naturally decompose a problem, e.g. in the FFT, are used to define redistributions.

[18] introduces a multi-phase approach for performing array redistribution when source and target block-cyclic distributions satisfy certain conditions. The underlying idea is to perform the array redistribution as a sequence of redistributions so that the total cost of the sequence is lower than the cost of the direct redistributions. This approach has been extended to handle arbitrary source and target block-cyclic distributions.

[19] introduces a mathematical representation, PITFALLS, for regular data redistributions, which finds the intersections between source and target destinations. The intersection determines the send/receive processor/data sets, and based on these sets general redistribution algorithms can be devised.

3 Overview of Psi Calculus

Psi Calculus is a mathematics of arrays which defines formal methods for representing, creating, accessing, and transforming arrays abstractly and in a general manner. This calculus can be used to describe any physical or abstract problem that can be represented as an array, as well as to create and verify a design for a solution. Psi Calculus is used extensively throughout most of this paper.The general partitioning algorithm is designed using its methods. Using the reduction rules of Psi Calculus the design is transformed to a normal form, a generic model for implementation. A brief overview of Psi Calculus follows, mainly through the use of examples. The interested reader should refer to *Mathematics of Arrays* [1] for a complete and more detailed discussion of the entire Psi Calculus.

3.1 Representing Arrays

Psi Calculus represents arrays as abstract objects defined by their dimension and shape. The most common symbol for arrays is the Greek letter ξ, and the dimension of an array is represented as $\delta\xi$. The dimension can also be specified using a superscript. For example, a 4 dimensional array can be represented as ξ^4, and $\delta\xi^4 \equiv 4$. Arrays of dimension 1 are often referred to as vectors, and are represented in the common way ($\vec{i}, \vec{p}, \vec{d}$, etc.). Arrays of dimension of 0 are referred to as scalars and are represented either by the Greek letter σ or by alphabetical letters without an arrow. Θ denotes the empty vector.

The shape of an array is a vector whose length is equal to the dimension of the array and whose elements are the extents of each dimension. The Greek letter ρ is used to represent the shape of an array. For example, if

$$\xi^2 \equiv \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

then $\delta\xi^2 \equiv 2$ and $\rho\xi^2 \equiv <3\ 4>$. As another example, consider the array

$$\xi^4 \equiv \begin{bmatrix} \begin{bmatrix} 10 & 11 \\ 12 & 13 \end{bmatrix} & \begin{bmatrix} 14 & 15 \\ 16 & 17 \end{bmatrix} \\ \begin{bmatrix} 18 & 19 \\ 20 & 21 \end{bmatrix} & \begin{bmatrix} 22 & 23 \\ 24 & 25 \end{bmatrix} \\ \begin{bmatrix} 26 & 27 \\ 28 & 29 \end{bmatrix} & \begin{bmatrix} 30 & 31 \\ 32 & 33 \end{bmatrix} \end{bmatrix}$$

For this array, $\delta\xi^4 \equiv 4$, and $\rho\xi^4 \equiv <3\ 2\ 2\ 2>$. By definition, the shape of a scalar is the empty vector Θ.

Two other useful operators for representing arrays are π (pi) and τ (tau). The operator τ gives the number of components in an array.
For example, if $\vec{v} \equiv <8\ 2\ 6\ 1\ 11>$, then $\tau\vec{v} \equiv 5$. Tau is also useful, since for any array ξ, $\tau(\rho\xi) \equiv \delta\xi$. By definition, $\tau\sigma \equiv 1$ and $\tau\Theta \equiv 0$. The operator π gives the product of the elements of a vector.
For example, $\pi(<3\ 8\ 2>) \equiv 48$. π is also useful, since for any array ξ, $\pi(\rho\xi) \equiv$ the total number of elements of ξ, or $\tau\xi$. By definition, $\pi\Theta \equiv 1$. Note that both τ and π always return scalars. Therefore, using $\vec{v}$ above,

$$\begin{aligned} \tau\vec{v} &\equiv \pi\rho\vec{v} \equiv \pi <5> \equiv 5 \\ \tau\sigma &\equiv \pi\Theta \equiv 1 \\ \tau\Theta &\equiv \pi <0> \equiv 0 \end{aligned}$$

3.2 Creating Arrays

Two major operators for creating arrays are ι (iota) and $+\!\!+$ (cat). The expression ιn creates a vector of length n whose elements are the consecutive integers from 0 to $n-1$. For example, $\iota 5 \equiv <0\ 1\ 2\ 3\ 4>$. Iota is also defined to operate on arrays as well, but this form of iota will not be used in this paper. By definition $\iota 0 \equiv \Theta$.

The expression $\vec{a} +\!\!+ \vec{b}$ concatenates two vectors $\vec{a}$ and $\vec{b}$ together in order to create a new vector.
For example, $<3\ 6\ 2\ 4> +\!\!+ <2\ 2\ 2> \equiv <3\ 6\ 2\ 4\ 2\ 2\ 2>$. Cat is also defined for arrays with dimension greater than 1, but is beyond the scope of this work. When concatenating scalars, they are considered to be 1 element vectors, while any vector concatenated with the empty vector simply returns the original vector.

3.3 Accessing Arrays

There are quite a few operators for accessing array elements, the most important of which are ψ (psi), $\triangle$ (take), and $\triangledown$ (drop). Psi is used to access single elements, single rows, single planes,... of arrays, and $\triangle$ and $\triangledown$ are used to access contiguous elements, contiguous rows, contiguous planes ... of arrays. When using ψ, arrays are considered 0 origin, so that the 0th row of an array is what most would consider the 1st row. Take allows you to specify which pieces of an array you want; drop is the inverse of $\triangle$, allowing you to specify which pieces of an array you do not want. Probably the best way to explain these operators is through example noting that all data access operations are defined in terms of ψ.

Consider the array

$$\xi_a \equiv \begin{bmatrix} 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 \\ 22 & 23 & 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 & 32 & 33 \end{bmatrix}$$

- Psi:
  $<1>\ \psi\xi_a \equiv <16\ 17\ 18\ 19\ 20\ 21>$
  which is the first row of ξ_a, and
  $<2\ 4>\ \psi\xi_a \equiv 26$
  which is the fourth element of the second row of ξ_a

- Take:
  $<2>\ \triangle\xi_a \equiv \begin{bmatrix} 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 \end{bmatrix}$
  which is the first two rows of ξ_a,
  $<3\ 2>\ \triangle\xi a \equiv \begin{bmatrix} 10 & 11 \\ 16 & 17 \\ 22 & 23 \end{bmatrix}$
  which is the first 2 elements of the first 3 rows of ξ_a, and
  $<2\ -3>\ \triangle\xi a \equiv \begin{bmatrix} 13 & 14 & 15 \\ 19 & 20 & 21 \end{bmatrix}$
  which is the last 3 elements of the first 2 rows of ξ_a
- Drop:
  $<3>\ \triangledown\xi_a \equiv \begin{bmatrix} 28 & 29 & 30 & 31 & 32 & 33 \end{bmatrix}$
  which is everything but the first 3 rows of ξ_a,
  $<2\ 3>\ \triangledown\xi a \equiv \begin{bmatrix} 25 & 26 & 27 \\ 31 & 32 & 33 \end{bmatrix}$
  which is everything but the first 3 elements of everything but the first 2 rows of ξ_a, and
  $<-3\ -2>\ \triangledown\xi a \equiv \begin{bmatrix} 10 & 11 & 12 & 13 \end{bmatrix}$
  which is everything but the last 2 elements of everything but the last 3 rows of ξ_a
- Gamma:
  gamma[i,v] [2] gives the index into the ravelled array of shape v for an index i (assumed valid for the array of shape v). $\gamma(\{2,3\}, \rho\xi_a) \equiv 15$
- GammaPrime:
  GammaPrime[a,v] gives the cartesian coordinate index equivalent to the index a into a ravelled array of shape v (the index a is assumed valid for the ravelled array of shape v. $\gamma\prime(15, \rho\xi_a) \equiv \{2,3\}$

3.4 Transforming Arrays

Three transformation operators will be used in this paper: Reshape (reshape), `rav` (ravel), and $\circledslash$ (transpose). Reshape is used to give an array a different shape. Ravel is used to flatten any array into a 1 dimensional vector, and is important because it allows arrays to be represented mathematically as how they would be stored in physical linear memory. Transpose is probably a familiar operator already, but in Psi Calculus it has a much more general definition which will be discussed shortly. These operators will be explained via example, using the array ξ_a from the previous example.

Ravel: rav $\xi_a \equiv <10\ 11\ 12 ... 31\ 32\ 33>$

Reshape: $<2\ 3\ 4>\ \hat{\rho}\xi_a \equiv \begin{bmatrix} \begin{bmatrix} 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 \end{bmatrix} \\ \begin{bmatrix} 22 & 23 & 24 & 25 \\ 26 & 27 & 28 & 29 \\ 30 & 31 & 32 & 33 \end{bmatrix} \end{bmatrix}$

Notice that $\hat{\rho}$ preserves the lexicographical ordering of the array. In other words, ravelling the original array and

[2] Gamma in this paper refers to $gamma_{row}$, $gamma_{column}$ is defined in [13]

ravelling the reshaped array will give the same result, assuming that both arrays have the same number of elements[3].

$$\text{Transpose: } <1\ 0>\ \bigcirc\!\!\!\!\backslash\ \xi_a \equiv \begin{bmatrix} 10 & 16 & 22 & 28 \\ 11 & 17 & 23 & 29 \\ 12 & 18 & 24 & 30 \\ 13 & 19 & 25 & 31 \\ 14 & 20 & 26 & 32 \\ 15 & 21 & 27 & 33 \end{bmatrix}$$

where $<1\ 0>$ is a permutation vector describing how the dimensions of ξ_a are to be reordered, and therefore how the array is to be restructured. Although this example shows a normal transpose of a 2 dimensional array, Psi Calculus defines transpose for arrays of any dimension by using a permutation vector that has length equal to the dimension of the array to be transposed. It should be noted that as the dimension of the array increases, the ways in which an array can be transposed increases. A 3 dimensional array can be transposed 6 (3!) different ways, a 4 dimensional array can be transposed 24 (4!) different ways, and so on.

3.5 Omega

As you may have noticed, most of the operators discussed so far have been defined over the primary axis. The operator Ω allows these operations to be applied over the other dimensions as well. How Ω works is difficult to understand and would require a rather lengthy explanation. Therefore, the discussion of Ω will be limited to the specific manner in which it is used in this paper, namely to apply $+\!\!+$ over the 0th dimension (i.e. over scalars). For example,

$$Ravel(<2\ 3\ 4>\ +\!\!+\Omega_{<0\ 0>} <5\ 6\ 7>) \equiv <2\ 5\ 3\ 6\ 4\ 7>$$

by applying concatenation to the individual elements of $<2\ 3\ 4>$ and $<5\ 6\ 7>$. Understanding how Ω accomplishes this is not important. Recognizing expressions and understanding the final result are pertinent to the comprehension of later Psi Calculus theorems, proofs, and derivations contained in this paper.

4 Design of the Algorithm

The design of the algorithm first begins with a high-level mathematical representation of the problem expressed in an algebra reducible by the Psi Calculus. From this design, an operational normal form (generic way to build a program) will be derived and presented in future papers. This denotes the pseudo code for an arbitrary implementation. Throughout this paper, a few common symbols will be used to represent various abstract arrays and their shapes. Some are as follows, and others will be introduced later on.
$\xi_p \equiv$ the abstract virtual processor topology
$\vec{s}_p \equiv$ the shape of the processor topology $(\rho\xi_p)$
$\xi_a \equiv$ the data array we wish to partition
$\vec{s}_a \equiv$ the shape of the data array $(\rho\xi_a)$
$\xi_{new} \equiv$ the final partitioned array
$\vec{s}_{new} \equiv$ the shape of the final partitioned array $(\rho\xi_{new})$

4.1 The Overall Shape: $\vec{s}_{new}$

$\vec{s}_{new}$ is the mathematical representation of the shape of the data array ξ_a partitioned onto a virtual processor topology ξ_p. Eventually, though, this must map to a real set of processors and actual data.

[3]It is possible to reshape an array so that it has either less or more elements than before, but this is unnecessary for the understanding of this paper

4.2 The Intermediate Shape: $\vec{s}_{init}$

Unfortunately, one can not simply reshape ξ_a by $\vec{s}_{new}$ in order to obtain the high-level mathematical description of the final partitioned array, ξ_{new}. Reshape does not preserve the locality of the data elements and their relation to one another, since reshape is designed to preserve the lexicographical ordering instead. As an example, consider a 2 dimensional array ξ^2, such that

$$\xi^2 \equiv \begin{bmatrix} 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 \\ 22 & 23 & 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 & 32 & 33 \end{bmatrix}$$

where $\rho\xi^2 \equiv <4\ 6>$. If $\vec{s}_p \equiv <2\ 3>$,and the distribution vector is $<block, block>$, then $\vec{s}_{new} \equiv <2\ 3\ 2\ 2>$, and

$$\vec{s}_{new}\,\hat{\rho}\,\xi^2 \equiv \begin{bmatrix} \begin{bmatrix} 10 & 11 \\ 12 & 13 \end{bmatrix} & \begin{bmatrix} 14 & 15 \\ 16 & 17 \end{bmatrix} & \begin{bmatrix} 18 & 19 \\ 20 & 21 \end{bmatrix} \\ \\ \begin{bmatrix} 22 & 23 \\ 24 & 25 \end{bmatrix} & \begin{bmatrix} 26 & 27 \\ 28 & 29 \end{bmatrix} & \begin{bmatrix} 30 & 31 \\ 32 & 33 \end{bmatrix} \end{bmatrix}$$

However, the locality of the elements of ξ^2 is not preserved within the 2 dimensional arrays. What we want is for ξ_{new} to contain actual sub-matrices of ξ^2:

$$\xi^2_{new} \equiv \begin{bmatrix} \begin{bmatrix} 10 & 11 \\ 16 & 17 \end{bmatrix} & \begin{bmatrix} 12 & 13 \\ 18 & 19 \end{bmatrix} & \begin{bmatrix} 14 & 15 \\ 20 & 21 \end{bmatrix} \\ \\ \begin{bmatrix} 22 & 23 \\ 28 & 29 \end{bmatrix} & \begin{bmatrix} 24 & 25 \\ 30 & 31 \end{bmatrix} & \begin{bmatrix} 26 & 27 \\ 32 & 33 \end{bmatrix} \end{bmatrix}$$

Note that the shapes of both of the above arrays are the same; only the ordering of the elements is different.

A solution to this problem is that we must first determine an intermediate shape for ξ_a which we will represent from here on as $\vec{s}_{init}$.

For the array ξ^2 seen previously, $\vec{s}_{init} \equiv <2\ 2\ 3\ 2>$, and

$$\begin{aligned} \xi^2_{init} &\equiv \vec{s}_{init}\,\hat{\rho}\,\xi^2 \\ &\equiv \begin{bmatrix} \begin{bmatrix} 10 & 11 \\ 12 & 13 \\ 14 & 15 \end{bmatrix} & \begin{bmatrix} 16 & 17 \\ 18 & 19 \\ 20 & 21 \end{bmatrix} \\ \\ \begin{bmatrix} 22 & 23 \\ 24 & 25 \\ 26 & 27 \end{bmatrix} & \begin{bmatrix} 28 & 29 \\ 30 & 31 \\ 32 & 33 \end{bmatrix} \end{bmatrix} \end{aligned}$$

4.3 The Transpose Vector: $\vec{t}_v$

After we have reshaped the data array by $\vec{s}_{init}$, we then need to restructure it so that the elements of the shape of the processor array $\vec{s}_p$ are separated from the elements of the shape of the partitions. By restructuring the array in this way, we will have constructed an array with shape of $\vec{s}_{new}$ whose elements are ordered in the correct manner. This restructuring is accomplished using transpose with a permutation vector $\vec{t}_v$ that needs to satisfy the relation $\vec{s}_{init}[\vec{t}_v] \equiv \vec{s}_{new}$. For ξ^2_{init} above, $\vec{t}_v \equiv <0\ 2\ 1\ 3>$, and

$$\begin{aligned} \xi^2_{new} &\equiv \vec{t}_v\ \bigcirc\!\!\!\!\backslash\ \xi^2_{init} \\ &\equiv \begin{bmatrix} \begin{bmatrix} 10 & 11 \\ 16 & 17 \end{bmatrix} & \begin{bmatrix} 12 & 13 \\ 18 & 19 \end{bmatrix} & \begin{bmatrix} 14 & 15 \\ 20 & 21 \end{bmatrix} \\ \\ \begin{bmatrix} 22 & 23 \\ 28 & 29 \end{bmatrix} & \begin{bmatrix} 24 & 25 \\ 30 & 31 \end{bmatrix} & \begin{bmatrix} 26 & 27 \\ 32 & 33 \end{bmatrix} \end{bmatrix} \end{aligned}$$

which is the correct representation of the final partitioned array.

4.4 The final high-level design: ξ_{new}

Our final high-level design is therefore

$$\xi_{new} \equiv \vec{t}_v \bigcirc\!\!\!\!\backslash (\vec{s}_{init}\widehat{\rho}\xi_a)$$

Although the example presented in previous sections concerns only a 2 dimensional array with the $< block, block >$ distribution, and 2 dimensional processor topology, this high-level design describes any type of partitioning of any dimensional data array onto any dimensional processor topology, and does not require that the data array and processor topology have the same dimension. In HPF, this is not the case. We believe that there is a correlation between the HPF's distribution directives and reshape/transpose functions. In our future papers we will discuss the proof to the theorem:*Every distribution directive specified by HPF can be represented as reshape/transpose.*

5 The Algorithm:

5.1 The OVERALL representation of Algorithm

The algorithm contains 3 basic parts:

1. Finding the intermediate array shape $\vec{S}_{init}$.
2. Finding the new array shape $\vec{S}_{new}$.
3. Finding the permutation vector $\vec{t}_v$. The new array can be obtained by transposing the initial array using permutation vector.

The reason for creating an intermediate array is that we want to preserve the locality of the elements of the original array A in the new array. Restructuring the shape of the new array can be accomplished using transpose with a permutation vector that needs to satisfy the relation $\vec{S}_{init}[\vec{t}_v] = \vec{S}_{new}$.

5.1.1 Finding the shape of the intermediate array $\vec{S}_{init}$:

For finding the shape of $\vec{S}_{init}$, we need to have some information like:

$\vec{S}_a$ = the shape of the data array.

$\vec{S}_p$ = the shape of the processor array.

$\vec{S}_d$ = the distribution description vector, where each distribution will have its code.For now lets assume that the codes for different distributions are:

distribution	code
$\star$	0
block	1
cyclic	2
block(n)	3
cyclic(n)	4

For example $\vec{S}_d$ for the case when distribution is

$$< block, \star, cyclic(n), block(n), cyclic >$$

will be $< 1, 0, 4, 3, 2 >$.
The dimensionality of $\vec{S}_a$ and $\vec{S}_p$ can be different, but the only restriction is that:
For-all $0 \leq i < \tau\vec{S}_p$; $\vec{S}_p[i] \leq \vec{S}_a[i]$ and $\vec{S}_p[i]$ mod $\vec{S}_a[i] = 0$. Also dimensionality of $\vec{S}_a$ and $\vec{S}_d$ must be the same, for dimensions with no distribution, there will be a value of 0 in $\vec{S}_d$.
For finding this vector we need to know how many elements of information we will need for each distribution type.

distribution type	elements
$\star$	1
block	2
cyclic	2
block(n)	2
cyclic(n)	3

let $ELEMENTS$ =(sum of all the elements in the $\vec{S}_d$).

NOTE: There is no constraint on the value of n in block(n) and cyclic(n). n can have different values from index to index in the body of block or cyclic. The only restriction is that for computation involved in each index, the value of the corresponding n for that index must be used.

Walk along $\vec{S}_d$, $\vec{S}_a$,$\vec{S}_p$ and find the shape for the intermediate array $\vec{S}_{init}$.

1. Let $t = (\delta\xi_a \lfloor \delta\xi_p)$.
2. let $0 \leq i < ELEMENTS$, $f = 0$; (f will keep track of indexes for $\vec{S}_p$).
3. For $0 \leq j < t$ choose $\vec{S}_a[j]$ and $\vec{S}_p[f]$ and $\vec{S}_d[j]$
   let A $\equiv \vec{S}_{init}[i]$
   let B $\equiv \vec{S}_{init}[i+1]$
   let C $\equiv \vec{S}_{init}[i+2]$

A	B	C	$\vec{S}_d[j]$	i	j	f
$\vec{S}_a[j]$			(0)	i=i+1	j++	f
$\vec{S}_p[f]$	$\frac{\vec{S}_a[j]}{\vec{S}_p[f]}$		(1)	i=i+2	j++	f++
$\frac{\vec{S}_a[j]}{\vec{S}_p[f]}$	$\vec{S}_p[f]$		(2)	i=i+2	j++	f++
$\frac{\vec{S}_a[j]}{n}$	n		(3)	i=i+2	j++	f++
$\frac{\vec{S}_a[j]}{n\vec{S}_p[f]}$	$\vec{S}_p[f]$	n	(4)	i=i+3	j++	f++

4. append (t $\nabla\vec{S}_a$) to the rest of $\vec{S}_{init}$.

5.1.2 How to find the shape for the new array ($\vec{S}_{new}$)

: This is also very similar to $\vec{S}_{init}$ but the order of the steps we take are different.

1. Let $t = (\delta\xi_a \lfloor \delta\xi_p)$.
2. let $0 \leq i < ELEMENTS$, $f = 0$.
3. For $0 \leq j < t$ Calculate the values for the indices of $\vec{S}_{new}$.

$\vec{S}_{new}[i]$	$\vec{S}_d[j]$	i	j	f
none	(0)	i	j++	f
$\vec{S}_p[f]$	(1)	i=i++	j++	f++
$\vec{S}_p[f]$	(2)	i=i++	j++	f++
$\frac{\vec{S}_a[j]}{n}$	(3)	i=i++	j++	f++
$\vec{S}_p[f]$	(4)	i=i++	j++	f++

4. At this point set j to be 0,f to be 0, but i holds its value, and append the values to $\vec{S}_{new}$ as they are computed in the following table.

$\vec{S}_{new}[i]$	$\vec{S}_{new}[i+1]$	$\vec{S}_d[j]$	i	j	f
$\vec{S}_a[j]$	none	(0)	i++	j++	f
$\frac{\vec{S}_a[j]}{\vec{S}_p[f]}$	none	(1)	i++	j++	f++
$\frac{\vec{S}_a[j]}{\vec{S}_p[f]}$	none	(2)	i++	j++	f++
n	none	(3)	i++	j++	f++
$\frac{\vec{S}_a[j]}{n\vec{S}_p[f]}$	n	(4)	i= i+2	j++	f++

5. append (t $\nabla \vec{S}_a$) to the rest of $\vec{S}_{new}$

5.1.3 Finding the permutation vector for the transpose.

For finding this vector we need to know how many elements of information we will need for each distribution type.

Let $t = (\delta\xi_a \lfloor \delta\xi_p)$.

Let $ELEMENTS$ =(sum of all the elements in the $\vec{S}_d$).

In this case we need to operate in 2 stages:

1. For $0 \leq i < t$ where $0 \leq j < ELEMENTS$, walk along the $\vec{S}_d$ and for different distribution type, the following table specifies on which specific bit of $\vec{t}_v$ what value should go.

$\vec{t}_v[j]$	$\vec{t}_v[j+1]$	$\vec{t}_v[j+2]$	$\vec{S}_d[i]$	i	j
none	none	none	(0)	i++	j+1
i	none	none	(1)	i++	j+2
none	i	none	(2)	i++	j+2
i	none	none	(3)	i++	j+2
none	i	none	(4)	i++	j+3

2. l = t, and For $0 \leq i < t$ where $0 \leq j < ELEMENTS$, walk along the $\vec{S}_d$ and for different distribution type, the following table specifies on which specific bit of $\vec{t}_v$ what value should go.

$\vec{t}_v[j]$	$\vec{t}_v[j+1]$	$\vec{t}_v[j+2]$	$\vec{S}_d[i]$	i	j	l
l	none	none	(0)	i++	j+1	l++
none	l	none	(1)	i++	j+2	l++
l	none	none	(2)	i++	j+2	l++
none	l	none	(3)	i++	j+2	l++
l	none	l+1	(4)	i++	j+3	l+2

3. Assuming that j holds its value from last step,

   For m = l; $m < ELEMENTS$; $\vec{t}_v$[j++] = m.

5.1.4 Some of the Rules Involved in this Algorithm

1. Every index in the processor array must divide the corresponding index in the data array.
2. If we have an empty processor array or data array, an error message should be posted.
3. The processor array can have any dimensionality. The algorithm will only use as many indices of the processor array as needed. If there are more processor array indices than data array indices the extra indices will be ignored. If there aren't as many indices as the data array indices, then even though distribution has been specified for other dimensions of data array, they will be treated as if there is no distribution for those indices(star distribution).
4. If there is only a scalar as data element then the final partition will be an array of shape $< 1, 1 >$, since we will use only 1 processor and in that processor we will have only that one scalar.
5. The dimensions of data array,distribution array, and the array which has all the information about the value of **n** for the case of block(n) and cyclic(n) must be equal, if not an error message should be posted.
6. If the dimension of distribution array is 0 for any reason, an error message should be posted to the user.
7. For the case when we have a vector of data, things should be handled exactly the same way as an array of data.
8. For the case when the dimension of the data array is 1 and the length of shape of array is also 1, meaning that only one element is in the array then only one processor will be used.
9. If we have a scalar instead of the processor array, then that value will be used to partition the very first non star index of the data array.
10. If the array has only one element, in that case the only value for the corresponding processor index is 1, other than that, any other number greater than 1 will not divide 1, which is the number of elements in the array.
11. If the dimension of the processor array is less than the dimension of the data array, then for all the indices that there is no corresponding processor index, the distribution will be considered as star.
12. For any processor index the value must be greater than zero.
13. For indices with block(n) or cyclic(n) distributions the value of $\mathbf{n} * \mathbf{p}$ must divide the corresponding index of the data array and it must be greater than zero, where p is the corresponding index on the processor array.

Example:

Let $\vec{S}_a =< 2, 4, 12, 6, 8, 9, 4 >$

Let $\vec{S}_d =< 1, 0, 4, 3, 2, 0, 0 >$

Let $\vec{S}_p =< 2, 3, 3, 2 >$

$n = 2$, $\delta\xi_a = 7$, $\delta\xi_p = 4$ and t = 4.

Then $\vec{S}_{init} =< 2, \frac{2}{2}, 4, \frac{12}{2*3}, 3, 2, \frac{6}{2}, 2, \frac{8}{2}, 2, 9, 4 >$

Then in step one of the algorithm for $\vec{S}_{new}$,
$\vec{S}_{new} =< 2, 3, \frac{6}{2}, 2 >$.

Then in step two $\vec{S}_{new} =< 2, 3, \frac{6}{2}, 2, \frac{2}{2}, 4, \frac{12}{2*3}, 2, 2, \frac{8}{2}, 9, 4 >$.

Then in stage one of the algorithm for $\vec{t}_v$, $\vec{t}_v =< 0, , , , 1, , 2, , , 3, , >$

Then in stage two $\vec{t}_v =< 0, 4, 5, 6, 1, 7, 2, 8, 9, 3, 10, 11 >$

5.2 The Psi Calculus representation of the Algorithm

Since we want to use Psi Calculus to derive the algorithm to an operational normal form, the algorithm should be in a format understandable by Psi Calculus. The following 2 sections describe the algorithm expressible in a monolithic array algebra which is reducible by the Psi Calculus.

5.2.1 The algorithm for finding the shape of the intermediate array $\vec{S}_{init}$

Inputs to this algorithm:

- $\vec{d}$ = distribution vector.(shape is n(rank of array to be distributed))
- $\vec{p}$ = processor vector. (any shape greater than 0 is valid, if shape is less than n, then we fill the blanks by value of 1, also for indices with star distribution , first we shift the right hand side of that index to the right by 1, and then insert a value of 1 into that index.)
- $\vec{m}$ = possible **n** values for block(n) and cyclic(n). In case of star , cyclic and block the value of n by default is 1. Indices in $\vec{m}$ representing block(n) or cyclic(n) will be assigned with the value of n, and indices with star ,cyclic and block will be assigned with the value of 1.
- $\vec{a}$ = Represents the shape of the array. (shape of $\vec{a}$ is n).

Note: Rank of $\vec{d}$ and $\vec{a}$ and $\vec{m}$ must be equal.

A uniform matrix : For this algorithm we need to have a uniform matrix called UM which looks like the following:

$\frac{a_0}{p_0*m_0}$	p_0	m_0	$\frac{a_0}{m_0}$
$\frac{a_1}{p_1*m_1}$	p_1	m_1	$\frac{a_1}{m_1}$
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
$\frac{a_{n-1}}{p_{n-1}*m_{n-1}}$	p_{n-1}	m_{n-1}	$\frac{a_{n-1}}{m_{n-1}}$

Each row of this matrix represents one index in the distribution vector. So the first column represents the first index, 2nd represents the 2nd index and up-to n of them, where n is the rank of the distribution vector.

Any variable that appears in each row also represents the variable from the corresponding index of that variables vector.

For example a_0, p_0, m_0 represent a[0],p[0],m[0]. We will be building this matrix in our algorithm before using it.In order to build this matrix which will help us reach the end results, we need to consider additional information.

- **takevectmatrix**: is a matrix of $< n, 1 >$ where each row represents the number of elements to be **taken** from each row of *UM* matrix. This is a constant matrix where each of its indices represents a corresponding distribution. For example the first row of this matrix holds the value if the distribution is **0**.
- **psivectmatrix**: Is a matrix of $< n, 4 >$. Each row represents one particular permutation vector which will be used later on the *UM* matrix. This will help to change the order of elements on each row.

0	0	1	2	3
1	1	0	2	3
2	0	1	2	3
3	3	2	0	1
4	0	1	2	3

psivectmatrix

0	1
1	2
2	2
3	2
4	3

takevectmatrix

Steps to create array UM :

Let dd = reshape[{n,1},$\vec{d}$].

- To create the first column(A1)

$$\mathrm{A1} = \vec{a}\ {}_{/}\Omega_{<0\ 0>}\ \vec{m}$$

This will take care of $\frac{a_i}{m_i}$ now we need to build $\frac{a_i}{m_i*p_i}$.

$$\mathrm{A11} = \mathrm{A1}\ {}_{/}\Omega_{<0\ 0>}\ \vec{p}$$

- Concatenate this column to $\vec{p}$ to get 2 columns of array UM.

$$\mathrm{A2} = \mathrm{A11}\ {}_{+\!\!+}\Omega_{<0\ 0>}\ \vec{p}$$

- To get the third column of array UM :

$$\mathrm{A3} = \mathrm{A2}\ {}_{+\!\!+}\Omega_{<1\ 0>}\ \vec{m}$$

- To get the fourth column of array UM :

$$\mathrm{UM} = \mathrm{Result1} = \mathrm{A3}\ {}_{+\!\!+}\Omega_{<1\ 0>}\ \mathrm{A11}$$

- Therefore let:
  UM = Result1 := $((((\vec{a}\ {}_{/}\Omega_{<0\ 0>}\ \vec{m})\ {}_{/}\Omega_{<0\ 0>}\ \vec{p})\ {}_{+\!\!+}\Omega_{<0\ 0>}\ \vec{p})\ {}_{+\!\!+}\Omega_{<1\ 0>}\ \vec{m})\ {}_{+\!\!+}\Omega_{<1\ 0>}(\vec{a}\ {}_{/}\Omega_{<0\ 0>}\ \vec{m})$.
- We need to know what should be the permutation vector for each row of our UM matrix. For that reason we need to build the **psimatrix** for this particular distribution, where each row of this matrix specifies the permutation to be used by **vecpsi** function on the corresponding row of UM matrix.

$$\mathrm{psimatrix} = \mathrm{dd}\ {}_{\Psi}\Omega_{<1\ 2>}\ \mathrm{psivectmatrix}$$

- We need to know what should be the take matrix for this distribution with rank n. As mentioned before this matrix will be used at the very end step to determine the useful information to be taken from each row of UM.

$$\mathrm{takematrix} = \mathrm{dd}\ {}_{\Psi}\Omega_{<1\ 2>}\ \mathrm{takevectmatrix}$$

- We need to permute array UM such that each row can be permuted according to matrix psivectmatrix based on its distribution.

$$\text{Array-intermediate} = \mathrm{psimatrix}\ {}_{vecpsi}\Omega_{<1\ 1>}\ \mathrm{UM}$$

- Then the final array can be computed as follows:
  $\forall i\ 0 \le i < \tau(\rho(takematrix))$

$$\mathrm{Result} = \mathrm{Result} +\!\!+ ((< i > \Psi\ \mathrm{takematrix})\ \triangle\ (< i > \Psi\ \text{Array-intermediate})).$$

5.2.2 The algorithm for finding the shape of the Transpose Vector

Input to this algorithm : $\vec{d}$ = distribution vector (shape is n=rank of array to be distributed).

Algorithm computes:

- NSD = number of Non Star Distributions.
- n = rank of our array or total number of elements in $\vec{d}$.

Some constant matrices:

- psivectmatrix: which specifies the kind of permutation for each one of different distributions. The rows of this matrix will be used to permute elements on some final array. In this final array depending on the type of distribution of each row the corresponding permutation will be used. The indices of this array represent different distributions. 0: star, 1: block, 2: cyclic, 3: block(m), 4:cyclic(m).
- takevectmatrix: which represents the number of elements to be taken from each row of our final matrix. each index represents a distribution.

0	-1
1	-2
2	-2
3	-2
4	3

takevectmatrix

0	1	0	2
1	1	0	2
2	1	2	0
3	1	0	2
4	1	0	2

psivectmatrix

- let $\vec{dd}$ = reshape[{n,1},$\vec{d}$].

5.2.3 Create matrix FM(Final Matrix) :

At this point we will be creating 3 matrices of shape $< n, 1 >$, after some operations on these individual matrices, we will concatenate them to create a matrix of shape $< n, 3 >$ denoted by FinalMatrix(FM).

- Build the 3rd column of FM.

  column3 = reshape[{n,1},iota{n}]

  Initialize all elements of column3 to be NSD.
- Build the 2nd column of FM.

  column2 = reshape[{n,1},iota{n}]

  then initialize every element of column2 with constant (0).
- Build the 1st column of FM.

  column1 = reshape[{n,1},iota{n}]

  Initialize every row of column1 with constant (**-1**).
- We need to update column2. For every index with cyclic(m) distribution, we change the value of the element in that index from 0 to be 1.
- We need to update column1,We will need a vector(increment1). This vector increment1 can be built by first creating a vector of shape $< n >$ initialized to **0**.Then a counter to set to 0, incremented when the indices have non-star distribution.

  column1 = increment1 $_{+}\Omega_{<0\ 1>}$ column1
- There are some changes to be made in both column2 and column3 in parallel. The following code explains the necessary changes:
  temp = column2[1];
  column2[1] = column3[1];
  column3[1] = column3[1] + temp;
  For(i=2,i $\leq$ n,i++) {

  temp = column2[i];

  column2[i] = column3[i-1]+temp;

  column3[i] = column2[i]+1;

  }
- Having all three columns ready, concatenate them to create one big matrix of shape $< n, 3 >$.

  Matrix1 = column1 $_{+\!\!+}\Omega_{<1\ 1>}$ column2

  Matrix = Matrix1 $_{+\!\!+}\Omega_{<1\ 1>}$ column3
- Having this Matrix, we need to permute the indices in each row according to the psivectmatrix. Depending on the type of distribution of each row, that particular psivectmatrix row will be chosen for a permutation vector. First we need to find out for our distribution what will be the overall permutation matrix.

  psimatrix = dd $_{\Psi}\Omega_{<1\ 2>}$ psivectmatrix
- Then apply this matrix to **Matrix** to permute the whole matrix accordingly.

  Result1 = psimatrix $_{vecpsi}\Omega_{<1\ 1>}$ Matrix
- We need to see from each row of Result1 how many elements should be taken. For finding that first we need to create the take matrix for our distribution vector.

  takematrix = dd $_{\Psi}\Omega_{<1\ 2>}$takevectmatrix
- In our final step we will apply this takematrix to our Result matrix and concatenate the useful taken information to get the desired transpose vector.
  $\forall i\ 0 \leq i < \tau(\rho(takematrix))$

  Result = Result $+\!\!+$ (($< i >$ Ψ takematrix) $\triangle$ ($< i >$ Ψ Result1)).

An example of computing the shape vector for intermediate array

- let $\vec{d}$= {c,star,c(m),b(n),b}.
- Encoding $\vec{d}$ will result in $\vec{d}$= {2,0,4,3,1}.
- let $\vec{a}$ = {16,16,16,16,16}
- let $\vec{p}$ = {2,1,2,2,2}
- let $\vec{m}$ = {1,1,2,2,1}
- Compute the matrix UM(Uniform Matrix) for this distribution.
- Compute psimatrix and takematrix.

2	0	1	2	3
0	0	1	2	3
4	0	1	2	3
3	3	2	0	1
1	1	0	2	3

psimatrix

2	2
0	1
4	3
3	2
1	2

takematrix

- Apply the permutation vectors from psimatrix on UM to obtain Array-intermediate:

2	8	2	1	16
0	16	1	1	16
4	4	2	2	8
3	8	2	4	2
1	2	8	1	16

Array-intermidiate

2	8	2	1	16
0	8	2	1	16
4	4	2	2	8
3	4	2	2	8
1	8	2	1	16

UM

- Take the right number of elements from each row of Result1 according to the number specified on corresponding row of takematrix, and concatenate results of these operations :

  Result = { 8,2,16,4,2,2,8,2,2,8 }

An example of computing the Transpose Vector.

- let $\vec{d}$= {star,b,c,c(m),b(n),star,b,c}.
- Encoding $\vec{d}$ results in $\vec{d}$= {0,1,2,4,3,0,1,2}.

- count number of non star distributions and assign it to NSD. NSD = 6.
- build 3 column matrices. Update column1 using the increment1 vector.

- increment1 = {0,1,2,3,4,0,5,6}
- Modify both column3 and column2 in parallel as it explains in the above algorithm for computing the transpose vector,concatenating all 3 columns will result in matrix *Matrix.*

column1	column2	column3	Matrix		
-1	0	6	-1	6	6
0	0	6	0	6	7
1	0	6	1	7	8
2	1	6	2	9	10
3	0	6	3	10	11
-1	0	6	-1	11	12
4	0	6	4	12	13
5	0	6	5	13	14

- Compute the psimatrix and takematrix for this distribution of {0,1,2,4,3,0,1,2}
- Apply the permutation vectors from psimatrix on Matrix will give us *Result1*.

	psimatrix				takematrix		Result1		
0	1	0	2	0	-1	0	6	-1	6
1	1	0	2	1	-2	1	6	0	7
2	1	2	0	2	-2	2	7	8	1
4	1	0	2	4	3	4	9	2	10
3	1	0	2	3	-2	3	10	3	11
0	1	0	2	0	-1	0	11	-1	12
1	1	0	2	1	-2	1	12	4	13
2	1	2	0	2	-2	2	13	14	15

- Then applying the takematrix on this Result1 matrix and concatenating the taken values from each row will give us **Transpose Vector**:

  {6,0,7,8,1,9,2,10,3,11,12,4,13,14,5}

6 Implementation

All of the preceding definitions and theories, as well as the complete general partitioning algorithm are modeled in Mathematica using a Psi Calculus Mathematica package. This package was extremely helpful in coming up with the general partitioning algorithm, since most of the mistakes originally made in the definitions were caught when modeling them in Mathematica.

6.1 The Operational Normal Form

In order to implement the partitioning algorithm in code, the Psi Calculus expression

$$\xi_{new} \equiv \vec{t}_v \oslash (\vec{s}_{init} \widehat{\rho} \xi_a)$$

must first be derived to a operational normal form. The derivation, although important, is quite tedious and is not presented in the main body of this paper, but can be seen in full in [20]. The result of this derivation is given here:

$$\begin{aligned} \rho(\vec{t}_v \oslash \vec{s}_{init} \widehat{\rho} \xi_a) \quad &\equiv \quad \vec{s}_{init}[\mathtt{gu}(\vec{t}_v)] \\ &\qquad \text{by definition of } \oslash \end{aligned}$$

Since

$$\xi_{new} \equiv \vec{t}_v \oslash (\vec{s}_{init} \widehat{\rho} \xi_a)$$

then

$$\begin{aligned} <\vec{\imath} +\!\!+ \vec{\jmath}> \psi\, \xi_{new} \quad &\equiv \quad <\vec{\imath} +\!\!+ \vec{\jmath}> \psi\, \vec{t}_v \oslash (\vec{s}_{init} \widehat{\rho} \xi_a) \\ &\equiv \quad (<\vec{\imath} +\!\!+ \vec{\jmath}>)[\vec{t}_v] \psi\, (\vec{s}_{init} \widehat{\rho} \xi_a) \\ &\qquad \text{by definition of } \oslash \\ &\equiv \quad (<\vec{\imath} +\!\!+ \vec{\jmath}>)[\vec{t}_v]) \psi\, \xi_{init} \end{aligned}$$

which implies that

$$\begin{aligned} &(\mathtt{rav}\, \xi_{new}) \quad [\gamma(<\vec{\imath} +\!\!+ \vec{\jmath}>; \vec{s}_{new})] \\ &\equiv \quad (\mathtt{rav}\, \xi_{init})[\gamma(<\vec{\imath} +\!\!+ \vec{\jmath}> [\vec{t}_v]; \vec{s}_{init})] \\ &\qquad \text{by definition of } \gamma \\ &\equiv \quad (\mathtt{rav}\, \xi_a)[\gamma(<\vec{\imath} +\!\!+ \vec{\jmath}> [\vec{t}_v]; \vec{s}_{init})] \\ &\qquad \text{by definitions of } \widehat{\rho} \text{ and } \mathtt{rav} \end{aligned}$$

In this equation, $\vec{\imath}$ represents the indices into the virtual processor topology and $\vec{\jmath}$ represents the indices in the data array local to each processor. This expression tells us, for all possible values of $\vec{\imath}$ and $\vec{\jmath}$, what coefficients each element of these vectors is to be multiplied in order to obtain the lexicographical coordinates (i.e. the memory location offset from the beginning of ξ_a) of the element of ξ_a that $<\vec{\imath} +\!\!+ \vec{\jmath}>$ represents. And if the derivation is correct, this will be valid for any dimensional data array partitioned over any dimensional processor topology.The subexpressions for the coefficients have not been shown here, since they are variable due to different distributions. For the same reason this derivation can not be expanded further unless the distribution vector is known.

In order to implement this in a program, one could loop through all possible values of $\vec{\imath}$ and $\vec{\jmath}$, multiplying each element of $<\vec{\imath} +\!\!+ \vec{\jmath}>$ by the correct coefficient in order to obtain the memory location of all the elements of the partitions of ξ_a for each processor. This will work, but it's not very efficient, and all the benefits of the generality of the algorithm would be lost to the excessive computational cost. When distributions are known that reflect memory access pattern of important scientific codes, e.g. FFT, it is possible to further reduce our design to starts, stops and stride [3] that enhance performance. Further research will attempt to categorize classes of memory access pattern and their subsequent distributions that prevail in compute intensive programs.

7 Redistribution using local/global address computation.

General partitioning algorithm can be used to compute the local/global array addressing of redistributions. In order to redistribute an array we need to know the processors on which data of this array reside on. These processor ids will be used to send and receive data in order to rearrange data among processors and create the new distribution.

7.1 Send Part:

On each processor P_j, partition the elements in to separate queues based on their destination :
For i = 0 to (N-1), where N is the size of the local sub-array,

compute the global address of this location, then convert that global address to the local address with respect to a new distribution. From this last computation we can find the processor id **j**, for which this element is going to be sent to. Store this element in $Queue_j$.
In order to do all these computations we need to compute S_{init}, S_{new}, t_v for the old distribution and $S_{init'}$, $S_{new'}$, $t_{v'}$ for the new distribution . These vectors will also be used for receive part.
These vectors provide us with a tool to swing from one distribution mode to the other, and capture the information related to the local/global address of elements in each processor for each distribution.
Find the shape of the sub-array on this processor, call it **S**.
let n = size of distribution vector.
let Cn = number of cyclic(n) distributions in distribution vector.
let newN = n + Cn
Since we have assumed that things divide evenly, therefore **S**, n , Cn and newN, can be used for all the processors.
On each processor P_j:

- Find the index position of that processor on the virtual processor topology call it $index_{p_j}$.
- For i = 0 to (N-1) where N is the size of the local sub-array on P_j:
  - Compute the Cartesian coordinate index equivalent to the index $index_i$ into the ravelled array of shape **S** using the **GammaPrime** function. (The values computed here are the same for every processor, so one set of computation can be used for all processors).
    let $index_i$ = GammaPrime(i,**S**).
  - Concatenate $index_{p_j}$ to $index_i$ call it INDEX.
    let INDEX = Concatenate($index_{p_j}$,$index_i$)
  - Apply the reverse of the permute function to INDEX using t_v. Permute function takes a permutation vector
    $t_v = \{t_i, t_j, ..., t_n\}$, and an index vector ,$index_i$, for each element, i.e.,
    $index_i = \{idx_i, idx_j, ..., idx_n\}$. The permute function will permute the index positions on each vector as following.
    $index_i[t_i] = index_i[0]$
    $index_i[t_j] = index_i[1]$
    . . . .
    . . . .
    $index_i[t_n] = index_i[n]$
    On the contrary reverse permute(Rpermute) function, does the reverse of permute function, meaning that it will permute the index positions on each vector as following(Rpermute is just the same as vecpsi in Psi Calculus).
    $index_i[0] = index_i[t_i]$
    $index_i[1] = index_i[t_j]$
    . . . .
    . . . .
    $index_i[n] = index_i[t_n]$

    let $RtrIndex = Rpermute(index_i, t_v)$.
  - Compute the Global position of this element on the whole array by finding the index in to the ravelled array of shape S_{init} for an index of RtrIndex.
    let GLOBAL = Gamma(RtrIndex,S_{init}).
  - With the GLOBAL position, apply the reverse steps to get the local position of this element in the new distribution namely $S_{new'}$.
    Let $tmpindex = GammaPrime(GLOBAL, S_{init'})$.
  - Let $newindex = permute(tmpindex, t_{v'})$.
    Using newindex which is a Cartesian coordinate in to the new distribution, we can compute the processor number from this Cartesian coordinate.
  - $processorID = \tau'(drop(newindex, (-newN))$ This will give the processor id that this particular element should be sent to.
    NOTE: τ' is a function which simply converts the binary values to decimal.

7.2 Receive Part:

There are different queues received from different processors. On each processor P_j, for i = 0 to (N-1) where N is the size of the local sub-array, compute the global address of this location, then convert that global address to the local address of previous distribution. From this last computation the processor id **j** for which this element is coming from can be found. Therefore the first element on the $Queue_j$ will be placed in this location.
Find the shape of the sub-array on this processor, call it **S**.
let n = size of the distribution vector.
let Cn = number of cyclic(n) distributions in distribution vector.
let newN = n + Cn
Since we have assumed that things divide evenly, therefore **S**, n , Cn and newN, can be used for all the processors.
On each processor P_j:

- Find the index position of that processor on the virtual processor topology call it $index_{p_j}$.
- For i = 0 to (N-1) where N is the size of the local sub-array P_j:
  - Compute the Cartesian coordinate index equivalent to the index $index_i$ into the ravelled array of shape **S** using the **GammaPrime** function. (Since we have assumed that things divide evenly, therefore this value computed, can be used for all the processors.)
    let $index_i$ = GammaPrime(i,**S**).
  - Concatenate $index_{p_j}$ to $index_i$ call it INDEX.
    let INDEX = concatenate($index_{p_j}$, $index_i$)
  - Apply the reverse of permute function to INDEX using $t_{v'}$.
    let $RtrIndex = Rpermute(index_i, t_{v'})$
  - Compute the Global position of this element on the whole array by finding the index into the ravelled array of shape $S_{init'}$ for an index of RtrIndex.
    let GLOBAL = Gamma(RtrIndex,$S_{init'}$).
  - Having the GLOBAL position, apply the reverse steps to get the local position of this element in the old distribution namely S_{new}.
    let $tmpindex = GammaPrime(GLOBAL, S_{init})$.
  - $oldindex = permute(t_v, tmpindex)$.
    Using the oldindex which is a Cartesian coordinate in to the old distribution, the processor number from this Cartesian coordinate can be computed.
  - $processorID = \tau'(drop(oldindex, (-newN))$ This will give the processor id that this particular element should receive from. Therefore the queue from which the element of this location will be read from.

7.3 Example for Redistribution.

suppose we have an original array **A** of $<4,4>$

$$A \equiv \begin{bmatrix} 10 & 11 & 12 & 13 \\ 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 \\ 22 & 23 & 24 & 25 \end{bmatrix}$$

first we distribute our original array **A** onto 4 processors with distribution $<b,b>$.

$$p0 \begin{bmatrix} 10 & 11 \\ 14 & 15 \end{bmatrix} \quad p1 \begin{bmatrix} 12 & 13 \\ 16 & 17 \end{bmatrix}$$

$$p2 \begin{bmatrix} 18 & 19 \\ 22 & 23 \end{bmatrix} \quad p3 \begin{bmatrix} 20 & 21 \\ 24 & 25 \end{bmatrix}$$

Then we redistribute them to be $<c,c>$ on to 4 processors

$$p0 \begin{bmatrix} 10 & 12 \\ 18 & 20 \end{bmatrix} \quad p1 \begin{bmatrix} 11 & 13 \\ 19 & 21 \end{bmatrix}$$

$$p2 \begin{bmatrix} 14 & 16 \\ 22 & 24 \end{bmatrix} \quad p3 \begin{bmatrix} 15 & 17 \\ 23 & 25 \end{bmatrix}$$

To redistribute from $<b,b>$ to $<c,c>$ we need to know for every index of a sub-array on each processor starting from processor p0 to p3, from which processor that index is suppose to receive from and to which processor that index is supposed to send to. For example lets consider the sub-array on processor p1. This processor needs to send to p0,p1,p2,p3 and receive from p0,p1 ,p2 and p3. Lets find this by our computation. For all these computations the assumption is that the index starts from 0. For processor p1, we know that the sub-array has shape of $<2,2>$ and that the index position of this processor on the processor grid is $index_{p_j} \equiv \{0,1\}$. Then starting from the first element in the sub-array find this element's local cartesian coordinates on the sub-array with shape $\mathbf{S} = <2,2>$, $index_i \equiv \gamma'(i, \mathbf{S})$. For each element concatenate its local cartesian coordinate to the end of index position of processor p1 which is {0,1}, call it INDEX. In fact INDEX is composed of the position of processor p1 on the processor grid, and local cartesian coordinates of an element on this processor. The transpose vector for $<b,b>$ is t_v={0,2,1,3} and for $<c,c>$ is $t_{v'}$ ={2,0,3,1}. S_{init} = {2,2,2,2 } $S_{init'}$ = {2,2,2,2}. The value of newN = n + cN, where n(rank of distribution vector $<b,b>$ or $<c,c>$) is 2 and there is no cyclic(n) distribution, therefore the value of cN is 0, $newN = 2 + 0 = 2$.

To Send:

Send-algorithm(t_v, S_{init}, $t_{v'}$, $S_{init'}$){
j = get_my_rank
p_shape = shape of the processor array
$index_{p_j} = \gamma'(j, p\_shape)$
n = size of the distribution vector
cN = number of cyclic(n) distributions
newN = n + cN
for (i=0,i<N,i++){
$index_i = \gamma'(i, \mathbf{S})$
INDEX = concatenate($index_{p_j}$,$index_i$)
RtrIndex= Rpermute(INDEX,t_v)
GLOBAL=γ(RtrIndex,S_{init})
tmpindex = $\gamma'(GLOBAL, S_{init'})$
newindex = $Permute(tmpindex, t_{v'})$
procID = τ'(drop(newindex,-newN))
Enqueue($Queue_{procID}$,A[$index_i$])
}
}

For the 4 elements of sub-array on p1 using $index_{p_1}$, e.g.,{0,1}, we will be computing what shows in table 1.

To Receive:

Receive-algorithm(t_v, S_{init}, $t_{v'}$, $S_{init'}$){
j = get_my_rank
p_shape = shape of the processor array
$index_{p_j} = \gamma'(j, p\_shape)$
n = size of the distribution vector
cN = number of cyclic(n) distributions
newN = n + cN
for (i=0,i<N,i++){
$index_i = \gamma'(i, \mathbf{S})$
INDEX = concatenate($index_{p_j}$,$index_i$)
RtrIndex= Rpermute(INDEX, $t_{v'}$)
GLOBAL=γ(RtrIndex,$S_{init'}$)
tmpindex = $\gamma'(GLOBAL, S_{init})$
newindex = $Permute(tmpindex, t_v)$
procID = τ'(drop(newindex,-newN))
A[$index_i$] = Dequeue($Queue_{procID}$)
}
}

For the 4 elements of sub-array on p1 using $index_{p_1}$, e.g.,{0,1} we will be computing what shows in table 2.

8 Conclusion

In this paper we presented a *dimension independent general partitioning algorithm* which can be used to compute the local or global addressing of arrays efficiently.This algorithm may be used either statically or dynamically , e.g., redistribution. These computations can be done without large intermediate tables and descriptors on remote processors.Computations are based on a uniform algorithm and a few constant matrices. Due to the generality, the same code applies to all cases as opposed to designs which rely on multiple case statements to support variances in design, hence the code is more compact. In fact since this algorithm is based on the Psi Calculus, mathematical formulas and theorems proven correct, they can be reduced to an operational normal form based on the reduction rules of Psi Calculus. The only functions left from the algorithm after reduction, are Gamma ,GammaPrime and Psi functions which are all address calculating functions. These simple designs could easily be realized in hardware. This will help to speed the computation processing.

References

[1] Lenore M. Restifo Mullin, A Mathematics of Arrays. Ph.D. dissertation, Syracuse University, December 1988.

[2] Lenore M. R. Mullin, Daria R. Dooling, Erik A. Sandberg, Scott A.Thaibault, Formal Method for Scheduling, Routing and Communication Protocol. P 234-244, proceedings of 2nd international symposium on high performance distributed computing, July 1993.

[3] L. Mullin and M. Jenkins, Effective Data Parallel Computation using the Psi Calculus, Concurrency-Practice and Experience, sept 1996.

send-val	i	$index_i$	INDEX	RtrIndex	GLOBAL	tmpindex	newindex	drop(newindex,-2)	procID
12	0	{ 0, 0 }	{0,1,0,0}	{0,0,1,0}	2nd(12)	{0,0,1,0 }	{0,0,0,1 }	{0,0}	p0
13	1	{ 0, 1 }	{0,1,0,1}	{0,0,1,1}	3rd(13)	{0,0,1,1 }	{0,1,0,1 }	{0,1}	p1
16	2	{ 1, 0 }	{0,1,1,0}	{0,1,1,0}	6th(16)	{0,1,1,0 }	{1,0,0,1 }	{1,0}	p2
17	3	{ 1, 1 }	{0,1,1,1}	{0,1,1,1}	7th(17)	{0,1,1,1 }	{1,1,0,1 }	{1,1}	p3

Table 1: For send part this table summarizes the steps to convert the local indices of a subarray resulted from a $< b, b >$ distribution on processor p1 to the global indices in the original array A. And then during global to local address translation, converts the global indices from A, to the local indices on different processors, based on the $< c, c >$ distribution.The processor number to which each element of sub-array on processor p1 should be sent to is computed.

rec-val	i	$index_i$	INDEX	RtrIndex	GLOBAL	tmpindex	newindex	drop(newindex,-2)	procID
11	0	{ 0, 0 }	{0,1,0,0}	{0,0,0,1}	1st(11)	{0,0,0,1 }	{0,0,0,1}	{0,0}	p0
13	1	{ 0, 1 }	{0,1,0,1}	{0,0,1,1}	3rd(13)	{0,0,1,1 }	{0,1,0,1}	{0,1}	p1
19	2	{ 1, 0 }	{0,1,1,0}	{1,0,0,1}	9th(19)	{1,0,0,1 }	{1,0,0,1}	{1,0}	p2
21	3	{ 1, 1 }	{0,1,1,1}	{1,0,1,1}	11th(21)	{1,0,1,1 }	{1,1,0,1}	{1,1}	p3

Table 2: For receive part this table summarizes the steps to convert the local indices of a subarray resulted from a $< c, c >$ distribution on processor p1 to the global indices in the original array A. And then during global to local address translation, converts the global indices from A, to the local indices on different processors, based on the $< b, b >$ distribution.The processor number from which each element of the sub-array of A on processor p1 should receive from is computed.

[4] Charles H. Koelbel et al, The High Performance Fortran Handbook. The MIT press, 1994.

[5] William Gropp, Ewing Luk, and Anthony Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press, 1994.

[6] Vipin Kumar et al, Introduction to Parallel Computing: Design and Analysis of Algorithms. The Benjamin/Cummings Publishing Company, Inc., 1994.

[7] Thomas P. McMahon, The Mathematical Formulation of General Partitioning of Multi-dimensional Arrays to Multi-dimensional Architectures Using the Ψ Calculus. Department of CS of the University of Missouri-Rolla, Honors thesis,May 1995.

[8] Akiyoshi Wakatani,Michael Wolfe, A New Approach to Array Redistribution: Strip Mining Redistribution, Proc. Parallel Architectures and Languages Europe, July 1994

[9] Akiyoshi Wakatani,Michael Wolfe, Optimization of Array Redistribution for Distributed Memory Multicomputers.Parallel Computing,vol.21, no.9, 1995.

[10] Ashwath Thirumalai, J.Ramanujam, An efficient compile-time approach to compute address sequences in data parallel programs, In proc. 5th International Workshop on Compilers for Parallel Computer, Malaga, Spain, pp. 581-605, June 1995.

[11] Jennifer M. Anderson, Saman P. Amarasinghe and Monica S. Lam, Data and Computation Transformations for Multiprocessors, In Proceedings of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP '95), Santa Barbara, CA, July 19-21, 1995.

[12] D.W.Walker and S.W.Otto, Redistribution of Block-Cyclic Data Distributions Using MPI, Concurrency: Practice and Experience, vol8, no.9, pp.707-728, 1996.

[13] H.B.Hunt III , L.R.Mullin and D.J.Rosenkrantz, Towards an indexing calculus for efficient distributed array computation, technical report SUNYA-CS-97-04 .

[14] F. Desprez, J.Dongarra, C. Randriamaro, Y. Robert, Scheduling Block-Cyclic Array Redistribution, IEEE transactions on parallel and distributed systems, VOL 9, NO 2, February 1998.

[15] Yeh-Ching Chung, Ching-Hsien Hsu, and Sheng-Wen Bai, A Basic-Cycle Calculation Technique for Efficient Dynamic Data Redistribution, IEEE transactions on parallel and distributed systems, vol.9 NO.4,APRIL 1998.

[16] R. Thakur, A.Choudhary, and G.Fox, Runtime Array Redistribution in HPF programs, Proc. 1994 Scalable High Performance Computing Conf., pp.309-316, May 1994.

[17] R. Thakur, A.Choudhary, and G.Fox, Efficient Algorithms for Array Redistribution, IEEE Trans. Parallel and Distributed Systems, vol. 7, no. 6, pp. 587-594, June 1994.

[18] S.D. Kaushik, C.-H. Huang, J. Ramanujam, P.Sadayappan, Multi-phase Array Redistribution: Modeling and Evaluation, Proc. Int'l. Parallel Processing Symp., pp. 441-445, 1995.

[19] Shankar Ranaswamy, Prithviraj Banerjee, Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers, Proc. Frontier '95: Fifth Symp. Frontiers of Massively Parallel Computation. pp. 342-349, MacLean, Va., Feb. 1995.

[20] Haleh Najafzadeh, Lenore Mullin, A General Partitioning Algorithm for Local/Global array addressing, technical report SUNYA-CS-98-04.

[21] C. Van Loan, Computational Frameworks for the Fast Fourier Transform. SIAM, 1992.