A Nonannotative Approach to Distributed Data-Parallel Computing

A Shafarenko

Department of Electronic and Electrical Engineering, University of Surrey, GU2 5XH, England.
e-mail: a.shafarenko@ee.surrey.ac.uk

Abstract. An approach to data-parallel computing is presented which avoids annotation by introducing a type system with symmetric subtyping. The properties that are usually specified in annotations in a machine-dependent way become deducible from type signatures of data objects. The chief advantage of the method is that it caters for portability by presenting a data description in terms of algorithmic properties (most importantly symmetry of data and of access to it) rather than any machine-specific terms.

1 Introduction

Significant effort invested into the development of HPF and similar languages shows the importance of array properties, such as distribution, alignment, etc for efficient generation of code. However, the use of a virtual machine for data parallelism with a (very specific) rectangular array processor structure and equally specific data distribution modes poses the questions of exactly how much must the user know about the construction of the machine to produce efficient code and whether this code could survive a change of the hardware platform.

The central issue here is one of describing a hierarchy of array properties in a sufficiently abstract way, but not too abstract in order to keep the bearing on real hardware. This is precisely the main difficulty too: what is "too abstract" depends on the assumptions the code designer has to make and what if they are wrong. Should we have to move on to a Fortran 2000 because nonrectangular arrays become rife and the most efficient distribution mode is finally pseudo-random?

The characteristic feature of von Neumann computing is its impressive ability to *approximate* the hardware. We have expressions in languages and the machine has commands. So expressions are turned into commands by a compiler and still the programmer can think expressions and loosely identify commands as "operations" — whence comes most of the usable part of complexity theory. The hardware instructions may change after porting the code to a different machine, but the operations will not, nor will the cost intuition based on them. Why can we not expect the same in the parallel domain?

The answer lies with the nature of approximation in the DP paradigm. It is extremely difficult to regard the translation process as one which simply brings granularity down to singular atomic actions of the hardware (by compiling down the chain "functions $\rightarrow$ statements $\rightarrow$ operations $\rightarrow$ instructions"). Because of the complex spatial structure of distributed array objects and high communication costs the operational decomposition of an algorithm is inferior to a spatial decomposition of a data structure. As we decompose a von Neumann algorithm into instructions of decreasing granularity, so we should be able to decompose a data-parallel (DP) object's type into a system of subtypes of decreasing abstraction. It should be noted, however, that while operational decomposition is implicit and is done by a compiler, the breaking-down of a type into subtypes has got to be explicit and known to the programmer. The compiler can only choose which subtypes to recognise and efficiently translate; others will not be distinguished from their supertypes.

To summarise, here is the proposed translation scheme. In the source program, data objects are typed using a type system rich enough to reflect any access feature or other particularity of a distributed object following from the algorithm and so, by definition, machine-independent. Some of these properties will be important for implementation on a particular machine, some will not. If a feature is not important for a given implementation, the compiler will ignore the subtype and use the supertype (i.e. approximate). It will then use the operators defined in the supertype to support computing with the subtype. The result of such computation should be consistent with the approximation used.

By itself, this is very similar to the agenda of OOP with its subclasses and inheritance. There is, however, an important distinction. The OOP approach makes types rigid; the use of implicit coercions is very limited indeed even when operators naturally extend to a variety of nested types. With OOP, it is believed to be a virtue of the method that the programmer has to be constantly aware of precisely the data types being used, so that no unexpected ambiguity occurs. Since the intention of our type system is approximation, we have to require that all versions of an operator are homomorphic, so an automatic coercion not anticipated by the programmer would not ruin the consistency of the program.

The nature of approximation "in space" is nothing new either. Consider what we do with the shapes of three-dimensional objects, say pieces of wood, in real life. If a craftsman needs to process such an object, he may have a tool for the simplest particular case, for example, a cubic shape. Such a shape is characterised by the fewest parameters and can be processed very efficiently. What if the same processing needs to be done on a more complex shape, e.g. a parallelepiped? There may be a tool for that, too, albeit less efficient. Such a tool would work on a cube (since a cube is an instance of parallelepiped). Finally there may be a tool that does the job on an arbitrary prism, perhaps even less efficiently: it would still have to be applicable to the previous two cases of the object as they are instances of prism, too. This example shows that the classification principle in 3d can be identified with the spatial symmetry of objects. The less symmetry an object has the greater generality and the more senior its type in the subtyping hierarchy.

Symmetry is indeed the key to any spatial approximation. We can easily iden-

```
DIMENSION Q(IL,JL,KL), V(KL), W(JL,IL)
...
DO I=1,IL
    DO J=1,JL
        DO K=1,KL
           Q(I,J,K)=V(K)*W(J,I)
        END DO
    END DO
END DO
```

Fig. 1. Example of a vectorisable loop nest

tify certain symmetries that are important for data parallelism, not all of them being purely geometric. The following sections will introduce translational symmetry (which is essentially one of replication), affine symmetry (which comes from analysis of constant-strided integer objects) and the symmetry of distributed access. We shall define and exemplify the type lattices corresponding to our classification and shall introduce appropriate coercion rules.

Finally, the general model limitation, which we shall assume hereinafter, is that nested DP computing (in the sense of [Ble93]) is not supported, although it appears nesting would not change the nature of fundamental DP symmetries too much. It nevertheless requires a separate study.

2 Translational symmetry

Fig 1 shows an example of a vectorisable loop nest as it appears in Fortran. This example uses 3 arrays of different ranks in a treble loop nest. Consequently, some of the indexed variables will not depend on some of the loop indices, for example `V(K)` is not affected by `I`- or `J`-iterations and `W(J,I)` does not change with iterations in `K`. According to f-code[MSS93] this symmetry should be interpreted as *orientation* of operands to a DP operation and should be defined statically using constant Boolean masks.

Definition 1. *An m-*orientation *of a rank-R array A is an array object that, if indexed with $[i_0, i_1, \ldots, i_r]$, where r equals the length of the mask $|m|$, selects the element $[j_0, j_1, \ldots, j_R]$ of the array A, with the indices $[j_0, j_1, \ldots, j_R]$ drawn from $[i_0, i_1, \ldots, i_r]$ according to the Boolean mask m in order. The number of ones in a Boolean mask is called the* character *of the mask and is denoted $\Box m$. For any valid orientation of A, $\Box m = R$.*

Note that orientation introduces translational symmetry in each result dimension corresponding to a zero in the mask.

Using orientations instead of the original arrays one can bring the example in fig 1 to a common dimensionality and then drop the explicit iteration space altogether:

```
Q=[001]V * [110]W'
```

where the prime denotes matrix transposition. The notation here is syntactically similar to the "numbers in brackets" of APL 2[BPP88].

In a complete DP world, a function can be applied to a nonscalar argument. Although we can limit our analysis to a function of a single argument (and use currying), it has to have a certain rank: since the rank is a component of the multi-type, a function should have to have a static type signature in the rank component. If a function is applied to an object of a rank higher than the one the function assumes for the argument, this can only be interpreted as a DP application of the function in the co-space of the argument.

Definition 2. *An m-orientation of a function of a rank-r argument is a function that accepts an array argument of a higher rank $R = |m| > r$. It uses a subset of the argument indices, according to the mask m, with the rest of the indices appended to the index list of the function result.*

The type inclusion relation for types of translational symmetry follows from the fact that the lack of symmetry along an axis is a more general case than its presence, taking into account that symmetries associated with different axes are independent.

Definition 3. *Let two objects x and y have different rank masks $\rho(x) \neq \rho(y)$, with the actual ranks being the same: $|\rho(x)| = |\rho(y)| = r$. Then the type inclusion relation $\rho(x) \subset \rho(y)$ is defined by the partial order $(\forall i : 1..r)\rho(x)_i \leq \rho(y)_i$, according to the standard subsumption.*

The orientation symbol used above for array orientation (a mask in square brackets) has the following "rank-mask signature":

$$(\forall a : |a| = \Box m)[m_1..m_n] : \{a_i\} \to \{b_j\}, \text{ where } b_j = \begin{cases} a_{\omega(m,j)}, & \text{if } m_j = 1 \\ 0, & \text{otherwise} \end{cases} .$$

Here $\omega(m,j) = \sum_{k=1}^{j} m_k$. Finally, let us join all rank lattices together at the bottom, by making every scalar type a member of all ranks since this only introduces unambiguous upgrading coercions.

3 Individual access symmetry

Abstract parallelism of data can be described as the lack of interference between different elements of a nonscalar assignment so that the hardware *may* perform all elemental assignments at once. In practice, however, a distributed implementation would perform DP assignment in a certain fuzzy order to minimise the communication and scheduling costs. In the simplest case of a rectangular processor array, data objects participating in the same DP operation will be co-mapped onto the array with a certain block size. Although scheduling of different blocks may be totally independent, within a block computing would have to be strictly serial. Such an arrangement is less symmetric than the source DP model, but the user does not see much *operational* manifestation of that (i.e., in terms of what can and what can not be done with a given data object). To separate out objects with different symmetries we shall introduce an *a priori* access cost, which is an asymptotic ($N \gg 1$, with N being the object size) measure that guides the user in the choice of the correct access type.

Definition 4. *The a priori access cost is a triple* $(c_\tau, c_\alpha, c_\rho)$*, where* c_τ *is the cost of* **total access**, *i.e. retrieval of all items of the arrangement, but not necessarily in order;* c_α *is the maximum cost of* **affine access**, *i.e. an arrangement of array elements with the indices forming an arithmetic progression, and* c_ρ *is the maximum cost of* **random access**.

Now we are in a position to introduce access subtypes, initially for a single dimension of an array, by giving upper bounds to the corresponding access costs.

Subtype	c_τ	c_α	c_ρ
locator	$O(N)$	$O(N)$	$O(N)$
collector	$O(1)$	$O(\log N)$	$O(\log N)$
sequencer	$O(1)$	$O(1)$	$O(\log N)$
director	$O(1)$	$O(1)$	$O(1)$
replicator	0	0	0

The above definitions also define the chain of type inclusions

$$\textit{replicator} \subset \textit{director} \subset \textit{sequencer} \subset \textit{collector} \subset \textit{locator}\,,$$

which is a linear order on types.

Definition 5. *The access type of a multidimensional object is the Cartesian product of per-axis types.*

The type inclusion relation between multidimensional access types is one of partial order: a subtype has to be junior in *all* dimensions of a supertype. Objects of different ranks have incommensurable access types. All access types with a common rank form a lattice.

4 Data-parallel skeletons

In the framework of the skeleton approach[Col89], the DP operators can be regarded as instances of a few high-order functions that depend on functional parameters or introduce appropriate data structures. We shall consider some of them below. Since we need to use product types as well as array types in type signatures, it is important that we use some unambiguous notation. We shall denote as ${}^r x$ the type of an array which has rank r and el-type x. When a superscript follows a type variable, as in x^n, this should be interpreted as a product type, i.e. the type of all n-tuples of objects of type x. When we use both preceding and succeeding superscripts, an ambiguity may result as that can be interpreted either as an array of tuples or a tuple of arrays. In all such cases we shall bracket the type expression explicitly. Finally, wherever the access component of type has to be specified, we shall use a preceding subscript, so ${}^{2}_{\mathrm{sl}}t$ denotes the type of any 2d array with el-type t whose access types in the first and second dimensions are s and $\mathbf{l}$, respectively. Note that s$\mathbf{l}$ in this example is, in fact, the Cartesian product of per-axis types (see def. 5), which makes it legal to use power as well, e.g. $\mathrm{c}^3 = \mathrm{ccc}$.

We have fully defined 4 skeletons for DP computing: *Map*, *Juxtapose*, *Select* and *Concatenate*. In this brief overview we shall not, for lack of space, describe all of them, but only the most interesting ones: *Map* and *Select*.

5 Map

This is the fundamental skeleton of DP computing. It applies a pure function to an array element-wise and has the following type signature:

$$(\forall r > 0, a, b)\,({}^0a \to {}^0b) \to {}^ra \to {}^rb\,,$$

which introduces overloading in rank. For any function f, *Map* f is indifferent to the access type of the argument: the access part of the signature is therefore fully decoupled from the rest and is given by

$$(\forall r > 0, x \subseteq \mathbf{l}^r)\,() \to x \to x\,,$$

where $\mathbf{l}$ is the locator access type and () is the access type of a scalar.

A generic *Map* skeleton must also allow the function argument to accept arrays of any rank not exceeding the rank of the second argument of the *Map*. Therefore there has to be a family of skeletons $\{Map_m\}$ parametrised with an orientation mask m, with the following signature:

$$(\forall k = |m|;\ \forall a, b)\,({}^{\Box m}a \to {}^0b) \to {}^ka \to {}^{k-\Box m}b\,.$$

Now let us define the (still disjoint) *access* type signature of Map_m:

$$(\forall y : (\widehat{m}y) \subseteq x)\,(x \to ()) \to y \to (\widehat{\overline{m}}y)\,,$$

where the bar above m is the standard denotation of bit compliment, and the hat over the mask denotes the projection operator defined earlier. Note that the first argument, a function returning a scalar, is antimonotonic in the access type of its argument.

The functional argument (call it functional parameter to avoid confusion) can be any function taking an object of rank $\Box m$ into an object of rank 0 (the latter guarantees non-nesting). However, three important cases below structure the functional parameter further, down to the level of scalar user-defined functions, which can be regarded as parameter-operators, and hence be treated algebraically.

Computation. This is a case of applying the functional parameter to the nonscalar argument to compute a new array. If the rank of the functional parameter argument is 0 then it defines an ordinary unary operator, such as $(-)$; if the rank is 1 or higher, the meaning of the Map is one of a reduction. Define three subskeletons:

$$\begin{aligned}
\Gamma^l &: (\forall a, b)^0(a \to b \to a) \to {}^0a \to {}^1_{\mathbf{l}}b \to {}^0a\,,\\
\Gamma^s &: (\forall a)^0(a \to a \to a) \to {}^0a \to {}^1_{\mathbf{s}}a \to {}^0a\,,\\
\Gamma^c &: (\forall a, r)^0(a \to a \to a) \to {}^0a \to {}^r_{\mathbf{c}^r}a \to {}^0a\,.
\end{aligned}$$

The reader familiar with high-order functions will easily recognise the **`foldr`** type signature of Γ^s, which has the meaning of a reduction with any associative (but not necessarily commutative) operator typed $a \to b \to a$ and its identity value typed a. Due to noncommutativity, the access signature requires type sequencer for the last argument. If the reduction operator is commutative as well, Γ^c should be used instead, generally with an increase in parallelism. Γ^c is polymorphic in the rank of the last argument as its semantics is not sensitive to the array structure (it uses the array argument as a bag).

Selection. This is a case of using the nonscalar argument of Map to provide some location information that the functional parameter can use to select a specific element from another array: such a function can always be represented as $\lambda x.(\Xi\, S\, f(x))$, with some numerical function f, some array S and the constant Ξ being the element selection function which returns the element of its first argument selected using the second argument as index tuple. Unfortunately such a selection primitive turns out to be insensitive to the access type of its argument and so a separate primitive is required which is not based on an instance of Map.

6 Select

There are two reasons for treating selections separately from the Map skeleton. Firstly, as was mentioned in section 5, they should be sensitive to the access type of the array source. Secondly, a more complex subtyping structure is required for the nonscalar index argument, which combines the already encountered translational with yet another, affine, symmetry, which occurs in integer objects.

The type signature of the Select skeleton is as follows:

$$Sel :: (\forall r, d, x)^r x \to ({}^dI)^r \to {}^dx\,,$$

where ${}^dI$ is some rank-d index type defined below, which we shall assume to be a subtype of ${}^d int$. (Remember the notation t^n is used for the nth power of type t in the Cartesian product sense, i.e. the type of n-tuples of type-x components.)

7 Affine integer type

In this section we shall use the translational symmetry notation introduced in the end of section 2.

Definition 6. *The purely affine type ${}^dA$ is the type of all d-dimensional, integer arrays v whose elements satisfy the following formula*

$$v_{i_1 i_2 \ldots i_d} = \sum_{k=1}^{d} a^{[k]} i_k + b$$

with some integer $a^{[k]}$ and b. (We enclose the superscript in square brackets to avoid any confusion with Cartesian powers of types)

An array may not have a purely affine type, with some of the dimensions still being purely affine. The most general case is described by the following expression:

$$v_{\mathbf{j}} = \sum_{k=1}^{n} a^{[k]}_{\mathbf{p}} i_k + b_{\mathbf{p}} \,,$$

where $\mathbf{p} = \widehat{m}\mathbf{j}$, $i_k = \{\widehat{\overline{m}}\mathbf{j}\}_k$, for some mask m, and all the coefficients are of the same rank $\Box m = |m| - n$.

Definition 7. *The index type ${}^dI$ is a type of a d-dimensional, integer array all elements of which satisfy the above formula with some mask m, $|m| = d$ and rank-l coefficients (where $l = \Box m$) $a^{[k]}$ and b. The general index type is fully defined by two Boolean masks:*

$$\tau = \overline{[m] \left(\bigvee_{k=1}^{\Box m} \rho(a^{[k]}) \right) \vee \rho(b)} \,,$$

which indicates by 1's which dimensions have translational symmetry, and $\alpha = \overline{m}$, showing which dimensions have affine symmetry.

The data constructor Υ for the general index type is parametrised with the mask α and accepts as the argument an $(l+1)$-tuple (where $l = \Box\alpha$) of affine form coefficients of equal rank: $\Upsilon\,\alpha[a^{[1]} : e_1, a^{[2]} : e_2, \ldots, a^{[l]} : e_l, b]$, where the integer scalars $e_1..e_l$ define the dimensions of the result along the affine axes.

Now we are well-equipped to define *affine subtyping* on type ${}^n int$. For a single dimension the type inclusion relation is as follows: ts $\subset$ as $\subset$ ns, where "ts" stands for translational symmetry, "as" for affine symmetry and "ns" for no symmetry. As before, a multidimensional subtype must be junior or equal to a supertype in all dimensions. We exemplify the type lattice in fig 2, where the case $d = 2$ is displayed.

The access type of an axis of affine symmetry is replicator. The implementation may choose to introduce "smart" upgrading coercions from the replicator type, which modify the way the coercee is produced rather than moving it about when the production is completed.

$$\begin{tikzcd}
 & & c_{i,j} & & \\
 & b_i j + c_i \arrow[ur] & & a_j i + c_j \arrow[ul] & \\
c_i \arrow[ur] & & ai + bj + c \arrow[ul] \arrow[ur] & & c_j \arrow[ul] \\
 & ai + c \arrow[ul] \arrow[ur] & & bj + c \arrow[ul] \arrow[ur] & \\
 & & c \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

Fig. 2. 2d affine type classification.

7.1 *Sel* skeleton

This function takes as many other arguments as the rank of the first one, the source. The reason they are not juxtaposed in the sense of the general DP paradigm, see [MS96] is because we do not wish to coerce the nonscalar index tuple to a single affine type, which would cause a loss of type information and therefore an excessive generalisation. Nevertheless, as far as the result contents are concerned, these can be defined element-wise as follows:

$$(Sel\, Z\, X_0\, X_1\, \ldots X_n)_{\mathbf{k}} = (Map\,(\Xi Z)\,[[X_0, X_1, \ldots, X_n]])_{\mathbf{k}}$$

for any valid multi-index $\mathbf{k}$.[1]

However, function *Sel*, unlike *Map*, can use the information about affine symmetries of the indices as well as the source argument access type to choose the most efficient *particular* selection. This is achieved by overloading *Sel* for any combination of τ and α of each index argument.

The access type requirements for the source of the *Sel* function are very easy to establish. Indeed, if the index corresponding to an axis of the source has an affine dimension, the axis type can be as high as sequencer. Otherwise the source axis is required to be a director. *Sel* is obviously polymorphic in the access type of all indices. How is the access type of the result defined? Denote as $\{w_k\}$ the access type tuple of $[[X_0, X_1, \ldots, X_n]]$. For any k, consider the following cases:

1. w_k is senior to type replicator. The respective result axis has the same type and alignment.
2. w_k is of type replicator. If the kth axis of each of the $X_0, X_1, \ldots, X_n$ is translationally symmetric, so is the result axis, and it has the same type and alignment. Else if all but one axis are such, with the remaining axis of an X_m being affine, then the result axis is aligned with the mth axis of the source. Otherwise, same as case 1.

[1] This is *not* how Sel should be implemented, see section 5; we only use Ξ to define the value of the elements of the result

How does *Sel* act on an affine integer object as the source? If τ_i^s and α_i^s are the parameters of the affine symmetry of the source, τ_i^r and α_i^r the respective parameters of the result, and $\tau_i^{[k]}$ and $\alpha_i^{[k]}$ of the kth index object,

$$\tau_i^r = \bigwedge_{k=1}^{d} \tau_k^s \vee \tau_i^{[k]}$$

$$\alpha_i^r = \bigwedge_{k=1}^{d} \tau_i^{[k]} \vee \left(\alpha_i^{[k]} \wedge (\tau_k^s \vee \alpha_k^s)\right) .$$

It should be noted that the power of *Sel* surpasses all known non-nested DP selections so that they can be expressed via it straight away. For example, a SLICE of a vector V is given by $Sel\, V\,(\Upsilon 1[k : l, m])$, where m is the start, k is the increment, and l is the new horizontal dimension and the transposition of a matrix R can be encoded as

$$Sel\, R\,(\Upsilon 10[[1]1 : \mathrm{dim}_2(R), [1]0])\,(\Upsilon 01[[1]1 : \mathrm{dim}_1(R), [1]0])\,,$$

which clearly shows the 1d-affine, 1d-translational symmetry of the operation. (dim_k stands for the kth component of the object shape).

8 Conclusions

A type system based on analysis of symmetries inherent in distributed DP computing has been introduced and the fundamental DP skeletons have been typed accordingly. Programming with those could be free from HPF style annotation while conveying similar information to the compiler.

The support from EPSRC under grant GR/H78993, as well as funding from the Nuffield Foundation under their research grant scheme is acknowledged.

References

[Ble93] Guy E Blelloch. Nesl: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Karnegie Mellon University, 1993.

[BPP88] J A Brown, S Pakin, and R P Polivka. *APL2 at a glance.* Prentice Hall, Englewood Cliffs, N.J. 07632, 1988.

[Col89] M I Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, 1989.

[MS96] V B Muchnick and A V Shafarenko. *Data-Parallel Computing: the Language Dimension.* Thompson Publishers, 1996.

[MSS93] V B Muchnick, A V Shafarenko, and C D Sutton. F-code and its implementation: a portable software platform for data parallelism. *The Computer Journal,* 36(8):712–721, 1993.