PCRC Distributed Data Descriptor Abstract Specification

DRAFT

Bryan Carpenter, Don Leskiw and Xiaoming Li
Northeast Parallel Architectures Centre,
Syracuse University,
Syracuse, New York

James Cowie
Cooperating Systems Corporation,
Chestnut Hill, Massachusetts

With contributions from:

Denis Gannon et al
Department of Computer Science & CICA,
Indiana University,
Bloomington, Indiana

November, 1996. Revised June, 1997

1 Introduction

Currently, high performance parallel architectures do not have standard methods to access and manipulate distributed data structures such as arrays and pointer-linked trees. Research groups and commercial vendors often reinvent the tools and functions required for distributed data structures for each source-language, machine architecture, and run-time system.

The purpose of this document is to propose a simple and compact model for creating and accessing regular distributed data in a parallel environment. This document is *not* intended to be an application program interface (API) for a user-level library. While part of the specification may be visible at the source code level, the intent is that the specification provides a standard model for low-level compiler-generated code, run-time systems, and parallel toolkits capable of interoperating and sharing implementation layers.

2 Definitions

Context: An address space. A SMP UNIX process represents a single context. Lightweight threads executing ith the address space would share the single context. A machine such as the SP2 or Paragon can support several contexts per node. The terms "context" and "process" will be used interchangeably in this document.

Local segment: The set of local elements for a given context.

Distribution Target: The set of processes or contexts across which an aggregate is distributed.

Process grid: A set of processes or contexts labelled by a tuple of *process coordinates*.

Global subscript: A subscript into a distributed array, viewed as a global entity.

Local subscript: A subscript into a local segment of a distributed array. In the specific array model presented in the following section, a local subscript is a subscript into the local segment of an abstract index space or *template*. This may or may not require some translation before being used as a subscript into the physical local array segment—see section 3.7.

3 PCRC DAD Specification

This section describes that basic design of the Distributed Array Descriptor (DAD). The design description is necessarily somewhat abstract, since the DAD must transcend linguistic, programming model, and architectural boundaries. We have avoided language-specific bindings.

The model presented here is restricted to distributed multidimensional *arrays*: aggregates of elements of the same type, indexed over a fixed Cartesian product of integer intervals. The ultimate scope of an extended descriptor may be broader, covering, for example, distributed recursive data structures such as lists and trees.

3.1 Overview of the Data Representation

There are three kinds of information in the distributed array descriptor:

- "Type" information. This is the kind of information a compiler would use in static type-checking. In general this information must also be passed to the run-time system. It includes some description of the type of the elements, and the rank of the array.

- Information on the global shape and distribution of the array. This includes a description of the group of processes over which the data is distributed and of the mapping of elements of the array to those processes. This information is needed, in particular, in translation of a set of global array subscripts to a set of process grid coordinates and a set of local subscript.

- Information on the layout of the local elements of the array. This information is needed in translation from a set of local subscripts to an address in local memory.

These domains are further divided into pairs of components: a scalar component, containing information pertinent to the whole array, and a vectorial component, containing additional information on a per-dimension basis. As illustrated in figure 1, this yields 6 logical components:

1. Element type information.

2. Rank or dimensionality of the array—an integer `R`.

3. Distribution target—the *group* structure.

4. Information on the bounds and distribution of index ranges—a vector of `R` *range* structures.

5. A base address in local memory—a pointer.

6. Information on layout of elements in local memory—a vector of `R` *stride* structures.

This organization has been developed to maximize the potential for sharing of components by similar arrays. Two arrays which share the extent and alignment of a particular dimension will have identical range structures for the dimension

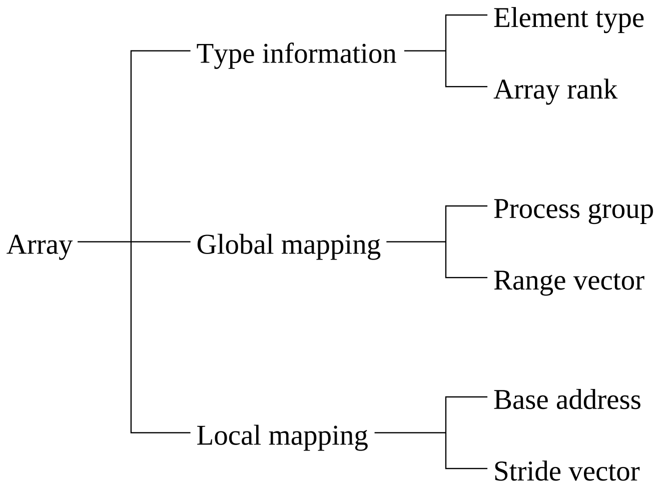

Figure 1: Structure of the array descriptor.

concerned. Two sections of a common parent array will share strides inheritted from the parent.

The remainder of this section reviews the content of the six components in turn

3.2 Element type information

An array element can have any type representable as a contiguous group of bytes. All elements of the array have the same type.

The type information in the descriptor consists of the element length in bytes and an integer tag field optionally used to provide more detailed information about the type. The tag field could be used by arithmetic array operations accepting arguments whose elements have any one of specified set of primitive arithmetic types.

3.3 Rank

The rank is a simple integer—the number of dimensions of an array. The rank may be zero. This allows the descriptor to describe a scalar variable which is localized to a particular process or group of processes.

3.4 Distribution target

An array is distributed over a particular group of processes, or contexts. This may be the whole set of processes actively executing the program, or some subset

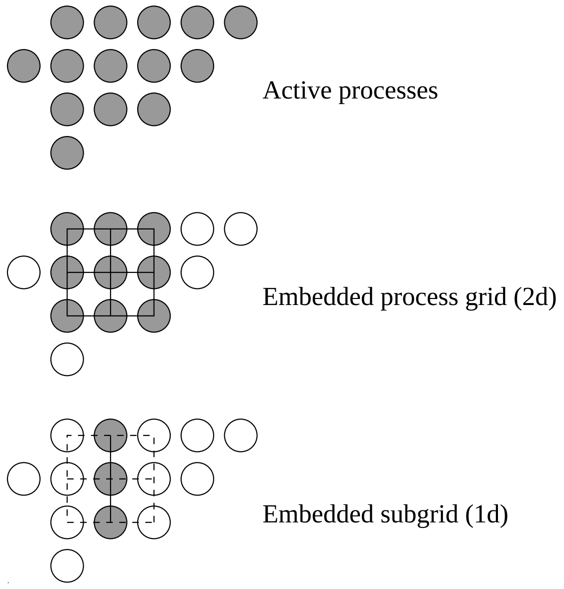

Figure 2: Examples of process groups over which an array could be distributed (shaded processes).

of those processes[1].

The model supports groups which are multidimensional grids of proceses, mapped into the active process set. It also supports groups which are regular subsets of other groups, obtained by restricting some of the grid dimensions to particular coordinates. These options are illustrated in figure 2.

Operations supported by the group structure include

- Creation of a group structure representing a process grid.
- Creation of a group structure representing a subgrid by restriction a dimension of an existing group.
- Provision of an inquiry function which determines whether the local process is a member of the group concerned, and (assuming that to be the case),
- Inquiry functions which return coordinates of the local process relative to the grid-dimensions of the group.

[1] The DAD does *not* directly provide mapping of multiple *abstract processors* (in the HPF sense) into a single address space. The higher levels of language specific run-time support could provide such mapping, if it was needed (in some cases that may depend on extensions to the supported distribution formats, described in the next section).

Additional lower-level group operations may include

- Conversion of grid coordinates to physical process identifiers
- Enumeration of all processes in a group
- Construction of spanning tree covering a group.

Such functions will certainly be needed within the run-time, but may not be required directly by an application or compiler generated code.

3.5 Index ranges

In the DAD, the extent of an array dimension and the mapping of its global subscripts into a process dimension is parametrized by a *range* structure. In languages that provide two-level mapping, with alignment of arrays to abstract index spaces (templates), the extent and mapping of a *template* dimension can also be described by a DAD range structure.

An essential röle of the range structure is to provide a map from global subscripts to local subscripts. This point of view is illustrated in figure 3. The map is visualized as a function from global subscripts to pairs consisting of a process coordinate and a local subscript.

The global subscript associated with an array dimension takes values in the range `0, ..., n - 1`, where `n` is the *extent* of the array in the dimension concerned. Where the source language allows lower bounds other than zero for array subscripts the compiler (or the language-specific layer of the run-time) is responsible for normalizing the subscript so that it "starts at" zero before passing it to the kernel run-time.

A *primitive range* is one that is mapped directly to a process grid dimension. Template dimensions are represented by primitive ranges. A primitive range is effectively parametrized by its extent, its target processor dimension, and a distribution format. A primitive range may have one of a number of supported distribution formats, including "block" or "cyclic" distributions. Different source languages often require different distribution formats, therefore this part of the interface and its associated implementation code are organized to make addition of new formats as straightforward as practical[2].

Besides primitive ranges the DAD supports *aligned ranges* (also called *subranges*). An aligned range is effectively parametrized by its extent, a parent primitive range, an alignment offset, and an alignment stride.

Operations supported by the range structure include

- Mapping from global to local subscripts, as discussed above

[2] In the NPAC/Adlib implementation each distribution format is represented by a different subclass of a common "range" base class, and, within certain constraints, new distribution formats can be added by extending the inheritance hierarchy with new subclasses.

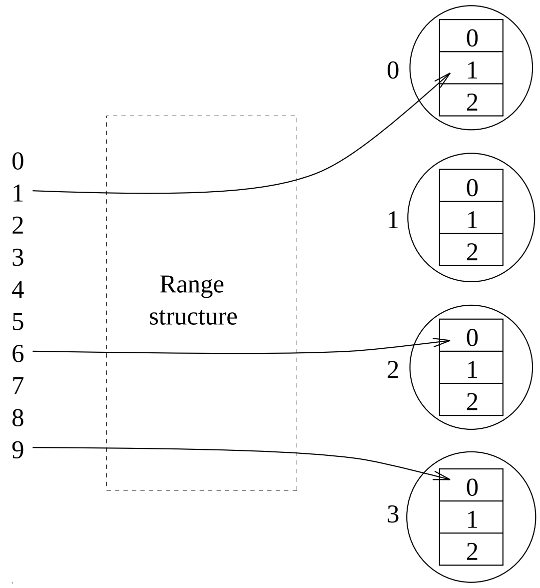

Figure 3: The range structure viewed as a map from global to local subscripts. The range has extent 10 and is distributed blockwise over a grid-dimension of extent 4. The arrows represent the mappings $1 \to (0,1)$, $6 \to (2,0)$, $3 \to (3,0)$.

- The reverse mapping, from local to global subscripts
- Inquiry functions returning extent and mapping information
- Construction of aligned subranges
- Operations to support enumeration of the locally held block(s) of subscript values.
- Operations to support enumeration of the global set of subscript blocks defined my the mapping of the range.

3.6 Base address

The base address is a pointer to base of a local array which holds the elements, or a pointer into heap memory.

3.7 Strides

> **stride (n.)** a term derived from the concept of walking or striding through data from one location to the next. [...]

A local subscript defines a local element of a primitive range or template. A local subscript does not *directly* encode information about the spread of array elements in physical memory. The mapping from local subscript to memory offset is embodied in the *stride*.

In the simplest case a memory stride is a single integer: if the local subscripts are

```
l_0, l_1, ...
```

and the memory strides are

```
str_0, str_1, ...
```

then the offset of the element from the base of the local array segment is

```
(str_0 * l_0) + (str_1 * l_1) + ...
```

Different *packing strategies* may be supported, requiring more general mappings from local subscript to offset. These may require extensions of the stride structure beyond a simple integer. For example, if an array is declared with an alignment stride[3] of s then it is wasteful of memory to create an array segment as large as the template because only one in s elements are ever used. This can be avoided if the local subscript is divided by s before subscripting the array. The value of the divisor s could be stored in the generalized stride structure.

[3] Of course the alignment stride and memory stride are two quite different concepts.

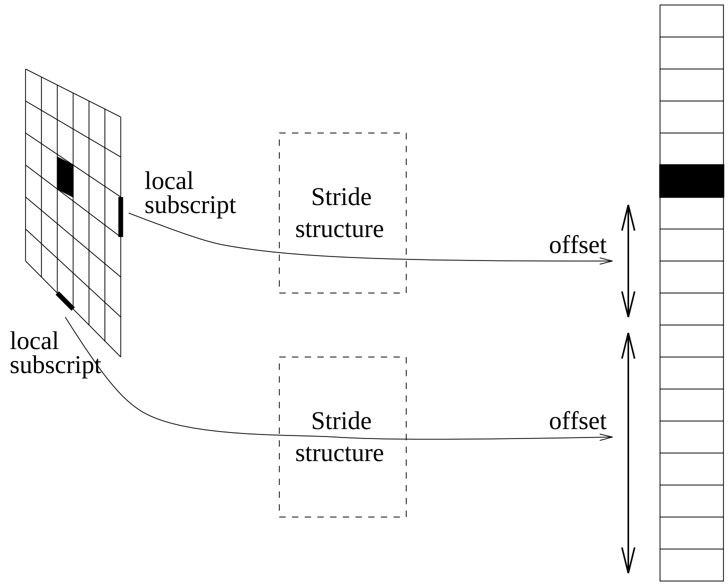

Figure 4: A pair of stride structures map a two-dimensional array segment into a section of local memory.

3.8 Further notes on the DAD

The DAD represents replication of data over any or all grid dimensions simply by distributing the array over a group containing the dimensions concerned, without specifying any index ranges distributed over those dimensions.

4 Language Bindings

The exposition given here is intentionally language-independent. Several bindings of the interface to specific programming languages (Fortran 77, C, C++, Java) will be presented in accompanying documents.