PCRC Distributed Data Descriptors Specification *

Version 0.4

Peter Beckman, Dennis Gannon, Beata Winnicka
Department of Computer Science & CICA
Indiana University, Bloomington, Indiana, U.S.A.

Alan Sussman
Department of Computer Science
University of Maryland, College Park, Maryland

Tomasz Haupt
Northeast Parallel Architectures Center at Syracuse University
Syracuse, New York

James Cowie
Cooperating Systems Corporation
Chestnut Hill, Massachusetts

February 27, 1995

*This research is supported by ARPA under contract xxxxxxxxxxxxxxx from xxxxxxxxxxxxxxxxx

PCRC Distributed Data Descriptors Specification

Contents

Abstract

This document is a draft. When completed, this document will formally provide a collection of data structures and specify function interfaces to a common descriptor for distributed data structures. The data descriptor can be shared by compiler writers and language designers, and forms a compiler-oriented interface for tools supporting the construction and manipulation of parallel/distributed data structures. Currently called the PCRC Distributed Data Descriptors (DDD), the specification is designed so compilers, tools, and run-time systems can reuse code and share data. Any implementation of the DDD specification would be a vital layer for a complete run-time system designed to support High Performance Fortran, HPC++, or any other parallel programming system that combines a global name space and distributed data structures. The specification has been designed in C++. Readers should be familiar with inheritance, templates, and virtual functions if they want to read the included C++ code.

1 Introduction

Currently, high performance parallel architectures do not have standard methods to access and manipulate distributed data structures such as matrices 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 standard for creating and accessing distributed data in a parallel environment. This document, the **PCRC Distributed Data Descriptor Specification** (PCRC-DDD), is not intended to be an application program interface (API) for a user-level library. While part of this specification may be visible at the source code level, the intent is that the specification provides a standard API for low-level compiler-generated code, run-time systems, and parallel toolkits capable of interoperating and sharing implementation layers.

Interoperability is a fundamental requirement for this specification. The PCRC-DDD specification will be available to both HPC++ and HPF compilers. This will allow programs to be constructed from both HPF and HPC++ libraries and object files. For example, a C++ program creating PCRC-DDD compatible distributed data structures could pass them to a library created with HPF.

Interoperability is also important at the tool level. For example, a tool builder could provide source-language independent visualization tools to view the placement and contents of data structures by linking to a run-time layer that implements this specification. A debugging tool that allows the user to modify or learn the current owner of a data element during a *break* is another tool that could be linked to such a run-time layer. Similarly, compilers for HPF, HPC++, C, and pC++ can share that portion of a run-time system built to this specification.

1.1 Programming and Run-time model

Concentrating on the run-time system and programming model that will use this specification, this section outlines some fundamental issues and ideas. A complete run-time system for a programming language that is compatible the PCRC-DDD specification consists of three layers.

1. Programming-model and language-dependent components. This layer provides the high-level services and language-specific aspects of the run-time system. Examples include alignment and distribution operators in HPF, task ports and channels in Fortran-M, infrastructure support for pC++ collections, and sophisticated I/O structures such as those required for persistent object-oriented databases.
2. PCRC-DDD functions. The allocation, access, and modification of aggregate structures, as specified in this document, is provided in this layer.
3. The low-level control and communication library. This layer defines the way parallel threads are created, controlled and synchronized and the way communication between different address spaces is handled. One implementation of this layer may use SPMD control with MPI for communication. Another example is the PORTS interface, which defines a library for dynamically creating and manipulating threads of control with communication based on remote procedure call, message passing, or remote put/get operations.

Taken together, these three layers provide a complete run-time system for a parallel language. Layer three provides thread control and communication mechanisms to support an object address and naming model for the layers above it. Layer two, supporting the PCRC-DDD specification, defines a standard set of mechanisms for building data structures. The PCRC-DDD interface can be constructed on top of a complete layer three implementation. (We envision SPMD/MPI and PORTS will be the initial two implementations.) Layer one, the language dependent layer, is implemented in terms of the interfaces defined by the two layers below it. The original source language will be compiled to linked to all three run-time layers.

Two different programming systems that follow the PCRC-DDD interface and share the same low level parallel control and communication library should be able to share data structures and libraries. However, it should be noted that the scope of the PCRC project does not stop with the PCRC-DDD interface defined in this document. Future documents from PCRC will partition layer one into several other components that can be used by compiler and library implementors to assure complete and elegant interoperability of different programming systems.

This document includes a series of definitions used throughout the remainder of the text. A critical component of the computation model is the assumption that distributed data structures are scattered across multiple virtual address spaces (such as processes within a set of machines on the network, or private address spaces associated with nodes on a distributed memory MPP system.) Consequently, we need naming mechanisms that allow us to identify individual objects within the distributed aggregate. Furthermore, mapping and translation functions must convert the global names to local ones. After a short description

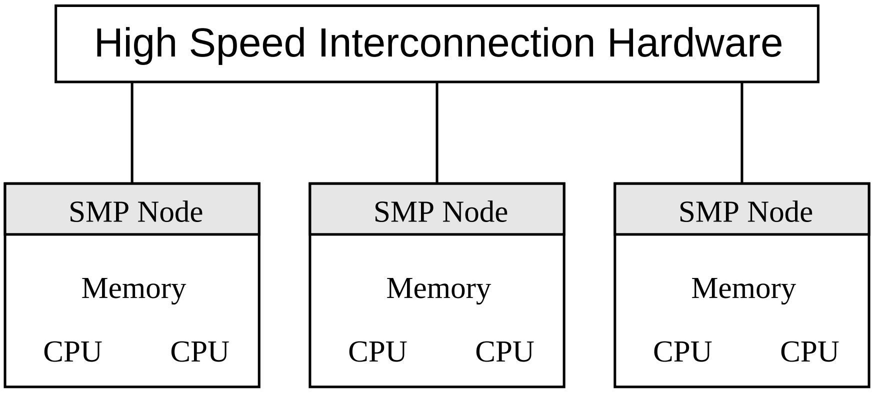

Figure 1: A Distributed-Memory Multiprocessor

of the target architectures, the remainder of this document is dedicated to the design of the PCRC-DDD specification.

1.2 Architectures

A distributed data descriptors must be designed so that efficient run-time systems may be developed for the following (MPP) machine architectures, which are targets for HPC++ and HPF:

- Shared-Memory Multiprocessors: For this class of machine, a single job may be spread across a set of processes, each of which may run several threads of computation. The processes may share data by simply sharing pointers to objects in the shared address space. For a symmetric multiprocessor (SMP), the processes might not be attached to a particular processing element. Cache coherency is expected. Most workstation vendors currently offer small SMP machines (2 - 8 nodes). SGI, Convex, and KSR offer large, high performance, shared-memory machines.
- Distributed-Memory Multiprocessors: Most message passing machines fall into this category. In this class of computer, a "node" is an SMP with memory, an operating system, and a connection to a very high speed, low latency interconnect. Vendor provided libraries often include primitives to facilitate parallel I/O and the sharing and scheduling of parallel tasks. Examples of this type of machine include the SP2 and Paragon. Although not low latency, networks of workstations running MPI or PVM are included in this class. For many machines, their "size" may be dynamic, or, the number of nodes may be altered during the computation.
- Tightly-Coupled Compute Partitions: Like the previous two machine types, these machines may have special hardware for synchronization and remote memory access, However, each node is not a general purpose SMP, but a "compute node" with an extremely simple kernel (lacking full OS support). It usually has a general-purpose front end for control and compiling, and the compute partitions are often single user systems. This class of machine includes the Cray T3D and TMC CM5. Such machines usually only support SPMD style programs, with only one process per node. The size of a partition is not mutable during the computation.

2 Definitions

The following definitions provide some vocabulary to discuss the design, basic data types, and implementation of the DDD:

Context: An address space. A SMP unix process represents a single context. Lightweight threads executing in the address space would share the single context. A machine such as the SP2 or Paragon can support several contexts per node.

Element: The basic building block for distributed aggregate data types. For example, Z[10] could be an element in a floating point array. A dynamically allocated data structure may also be an element. For example, each of the nodes in a binary tree would be called an element.

Local Elements: Those elements that share a context. This definition may be modified to mean "close" for single context shared memory machines that do not have uniform memory access. For "owner computes" execution models, the local elements represent the set of elements each processor is responsible for maintaining.

Logical Resource: A data entity that supports transparent access across remappings; for example, the array reference A[1:10], which retains its semantic meaning after possible realignment or redistribution. Contrast with Physical Resource

Physical Resource: A data entity that does not support transparent access across remappings; for example, a 32-bit machine pointer. More efficient than indirect addressing methods, but also more unsafe, since the compiler must "know" that runtime system services will not relocate or invalidate the storage pointed to while it is in use.

RootDDD: The abstract base class at the root of the DDD class hierarchy. All DDDs are derived from this class.

Map: Formal correspondence between two semantic domains. The rootDDD supports two maps, one from logical resource to physical resource, and one from logical resource to local (context specific) data descriptor. Sophisticated DDD clients (such as an HPF compiler) will add extra layers of mapping to implement template-alignment, block- and cyclic-distribution, runtime table-driven distribution, etc.

Remapping: Modifying the mapping functions. Usually this is done when moving elements between contexts.

ObjectID: A quantity which uniquely defines a logical resource, regardless of the processor or context in which the query is posed. Deriving such a quantity may be relatively simple (under strict SPMD models) or runtime-hard (under generalized task-parallel models, which could require general object servers).

3 Basic Data Types

OwnerContext: Identifies a single context within the computing hardware.

LocalPointer: A valid address (pointer) within a particular context. Most commonly used to point to elements. This resource contains no information about the type of a dereferenced LocalPointer. The term "Local" is used since the pointer only has meaning within the specified context; the result of dereferencing a LocalPointer in the wrong context is undefined.

GlobalPointer: A GlobalPointer uniquely describes a (untyped) memory location anywhere on the computational hardware.

Triplet: A (lowerbound, upperbound, stride) tuple that describes the elements one dimension a matrix section.

TripletIndex A vector of Triplets of length *rank*. Describes generalized array sections, including singleton points (triplets whose lowerbound equals their upperbound).

4 PCRC DDD Specification

This section describes the basic design of the DDD, its specific instances, and our implementation of it in an executable C++ specification. The design description is necessarily somewhat abstract, since the DDD must transcend linguistic, programming model, and architectural boundaries. We have tried to avoid language-specific bindings where possible; only the last section, describing our executable specification, resorts to C++ terminology to describe the DDD in terms of inheritance from templated abstract base classes.

Overview The DDD provides an abstract interface for the relationship between the global and local perspectives of a distributed datatype (DD), without specific reference to the mechanisms used to implement it, or the source language used to encode it. This allows variant implementations of the same DD to exchange information at the appropriate level of commonality. It also defines appropriate ways in which multiple-language implementations of the same abstract datatype can exchange information about the contents and the operations of that datatype at runtime.

Any distributed datatype has two inherent perspectives that must be maintained; these are the global view (describing the DD as a single, logical entity, ignoring its possible distribution) and the local view (describing one node's view of the DD after distribution has taken place).[1]

4.1 Elemental Datatypes and Logical Domains

We characterize distributable datatypes as the product of an elemental datatype with a logical domain.

An *elemental datatype* can be any undistributed datatype available in the source language. As a rule of thumb, simpler element types (machine primitive types, simple structures) will result in more portable and efficient DDD implementations.

A *logical domain* describes the structure of a collection of these elements, without reference to implementation. For a matrix, the rank-N product of strided integer intervals is a logical domain that describes strided submatrices. For a tree, the height, balance status, and branching degree might constitute a logical domain. In general, the only demand we make

[1]We describe a third perspective, the raw-physical view, in section 4.4.

of a logical domain type is that it support equality testing, although additional semantics (tests for containment or disjointness, or an inherent partial ordering of instances) will be both common and useful. We will use the terms *logical domain* and *shape* interchangably to represent the concept of the logical internal structure of an element collection.

Together, a collection's elemental type and logical domain completely specify the abstract distributed datatype. We define the concrete semantics of that datatype in stages by building two matching components: the *Global Data Descriptor* (GDD) and the *Local Data Descriptor* (LDD).

4.2 Global Data Descriptor

For a given distributed datatype, the Global Data Descriptor interface serves as the central repository for runtime information about the global "shape" of the data. It also provides three primary address translation functions that provide alternative perspectives (local and raw-physical), and two global-perspective communication functions (GET and PUT).

If the elemental type of an abstract distributed datatype is E, and the logical domain is D, then we define global data descriptor $GDD_{E,D}$ with mapping functions

$$GDD_{E,D}.RestrictToLocal : D \rightarrow D\ list$$
$$GDD_{E,D}.TranslateToLocal : D \rightarrow LDD_{E,D}\ list$$
$$GDD_{E,D}.toGlobalPointer : D \rightarrow GlobalPointer\ list$$

RestrictToLocal takes all or part of the logical-global data domain, and returns only those portions of the logical-global data domain that are local to this context. For example, for the upper-left processor in a 4x4 grid, and a 100x100 block-distributed matrix, calling *RestrictToLocal* with argument $(20 : 30, 1 : 100)$ might reasonably return $[(20 : 25, 1 : 25)]$.

TranslateToLocal takes all or part of the logical-global data domain, and returns one or more local data descriptors that describe the local perspective on that subdomain (see section 4.3 for details).

toGlobalPointer takes all or part of the logical-global data domain, and returns one or more global pointers (processor ID / memory offset pairs) to that data. This mapping function inherits the invalidation risks of the raw-physical pointer model (section 4.4).

Get and Put These global-perspective communication functions carry out single-threaded, many-to-many, globally-addressed data movement between global data structures.

$$GDD_{E,D}.Get : GDD_{E,D} \rightarrow DDD_{S}tatus$$
$$GDD_{E,D}.Put : GDD_{E,D} \rightarrow DDD_{S}tatus$$

Unlike the GDD address translation mechanisms, which may be safely used in a multithreaded context under the assumption that the distributions they query are static, the

GDD communication mechanisms should only be invoked from single-threaded contexts, unless appropriate exclusion mechanisms are implemented [2]

4.3 Local Data Descriptor

Similarly, for a given distributed datatype, the Local Data Descriptor interface serves as the central repository for runtime information about the local (in-memory) "shape" of the data. Where the GDD stores the logical-global shape, the LDD caches a reference to the GDD it derives from, and to the local collection of in-memory elements that it represents.

Also, like the GDD, the LDD provides two local-perspective communication functions (`Get` and `Put`):

$$LDD_{E,D}.Get : GDD_{E,D} \rightarrow DDD_Status$$
$$LDD_{E,D}.Put : GDD_{E,D} \rightarrow DDD_Status$$

These define, respectively, fetching of globally indexed source data into a target in local memory, and writing from a source in local memory into a globally indexed target region. These communication functions require nearly the same exclusion assertions as GDD `Get` and `Put`, to ensure that reads and writes to and from multiple contexts do not interfere. DDD implementations may provide this security, or trusted users (compilers, for example) may opt to work without a net.

4.4 Physical Pointer Model

Local data descriptors cache "real" (machine-level) pointers to elements in memory; coupled with a context identifier to show what address space they are native to, these cached values are useful as global pointers. Once a smart user (the compiler, for example) has derived pointer relationships between distributed data objects, these global pointers could be reused by a "Network DMA" type machine such as the Cray T3D or Meiko CS2 to copy elements from remote contexts to the local context without interrupting or "asking" the remote context for the element:

```
RemotePutBytes(global_pointer *dst, local_pointer *src, int count);

RemoteGetBytes(local_pointer *dst, global_pointer *src, int count);
```

This form of addressing is inherently more dangerous in the presence of dynamic redistribution, because of the threat of pointer invalidation. But the reduction in communication protocol overhead and the elimination of redundant calls to the GDD's address translation functions may make the physical pointer shortcuts attractive to careful DDD users.

[2]Not all DDD-compliant runtime systems will implement global communication routines, but symmetry and the existence of single-threaded runtime interfaces demand their inclusion in the interface.

4.5 Extending the DDD

The most abstract DDD basis defines only these minimal interfaces for GDDs, LDDs, and the physical pointer model, leaving distributed data's elemental type, logical domain type, and exact mechanisms of address translation and communication open to further specification. New DDDs will be *derived from* this (uselessly abstract) basis in three stages, starting with specific distributed datatypes, then moving on to their language-specific requirements, and finally capturing the details that differentiate specific runtime system implementations.

As the first step, the DDD specification provides the most common distributed data structure, the generic DDD for Matrices [3] `MatrixGDD` and `MatrixLDD` are the abstract global data descriptors for Matrices, where a Matrix is defined as a distributable datatype whose "elements" are contiguous blocks of words in memory, and whose "logical domain" is a tuple of integer triplets (ala Fortran 90). This abstracts the specific array models from both HPF and HPC++.

Annexes to the DDD specification will then derive Matrix DDDs that are specific to each of these languages (for example, defining the characteristic two-level mapping of HPF). Finally, for a given runtime system to be "DDD-compliant," it must present its specific treatment of matrices as a most specific ("leaf-level") DDD instantiation.

Appendix A. Class Interfaces

```
// Modified by Jim Cowie (2/20/95)
// Modified by Pete Beckman (2/20/95)
// Modified by Alan Sussman (2/22/95)

// Draft PCRC DDD (Distributed Data Descriptor) Version 0.42

/* This is a very rough draft of the classes for the DDD.  It includes
an abstract Matrix data descriptor.  It does not worry about
public/private yet, and has some contructor/destructor functionality
left out.  -Pete */

// ****************************************************************
// Some simple typedefs for type aliases
// ****************************************************************

// A basic machine address, no type information.
typedef void* LocalPointer;

// A boolean return type
typedef int boolean;

// Return status for some DDD functions.  if (DDD_Status < 0) ERROR!
```

[3] Others, for selected linked data structures, will follow.

```
typedef int DDD_Status;

// ****************************************************************
// Physical Objects
// ****************************************************************

// An OwnerContext uniquely describes an address space
class OwnerContext {
 public:
  int nodenumber;
  int contextnumber;

  // Comparing two OwnerContexts is done element-wise, and lets you
  // know if the two contexts refer to the same address space
  boolean operator==(OwnerContext C) {
    return ((C.nodenumber == nodenumber) &&
            (C.contextnumber == contextnumber));
  }
};

// Pointer to any physical memory location, regardless of location...
// Does NOT contain any type information (such as GlobalPointer to int)
// Should GlobalPointer be a template based on ElementType?
class GlobalPointer {
 public:
  LocalPointer toLocalAddress() { return P; };
  LocalPointer P;
  OwnerContext O;
  int IsValid;  // Does this GlobalPointer represent a REAL physical
       // resource I can "GET" or "PUT" to?
                // Maybe we should define a NULL?
};

// ****************************************************************
// ****************************************************************
// LOCAL Distributed Data Descriptor
// ****************************************************************
// ****************************************************************

// Need a description of what we mean by a DOMAIN - I'd do it, but I'm not
// sure that I know exactly how to define it

template <class ELEMENTTYPE, class DOMAIN> class GlobalDataDescriptor;

// Base class for all LOCAL Descriptors
```

```
template <class ELEMENTTYPE, class DOMAIN>
class LocalDataDescriptor {
 public:

  // Local storage associated with this local descriptor?
  // **************************************************************
  ELEMENTTYPE *LocalStorage; // Collection of ELEMENTTYPES (array, list, etc.),
                             // each one pointing at the beginning of a
                             // local component of the dist. data structure
  int CollectionLength; // Number of ELEMENTTYPES currently in collection

  // Global (parent) descriptor associated with this local descriptor?
  // **************************************************************
  GlobalDataDescriptor<ELEMENTTYPE,DOMAIN> *RestrictionOf;  // cache!

  // Basic constructors for LocalDataDescriptors
  // **************************************************************

  // Simplest Constructor.  Create new storage.
  LocalDataDescriptor();

  // Do we need copy constructor, aliased storage constructor, and
  // assignment operator (in derived class) here, as for GDDs?

  // Simple data movement routines
  // **************************************************************
  // Get copy of a portion of the domain
  virtual DDD_Status
     Get(GlobalDataDescriptor<ELEMENTTYPE,DOMAIN> *src) =0;

  // Put a portion of the domain into destination
  virtual DDD_Status
     Put(GlobalDataDescriptor<ELEMENTTYPE,DOMAIN> *dst) =0;

  // Proposed more complex data movement routines, to allow Getting
  // into an LDD that already has some valid data or putting some of
  // the data from a pre-existing LDD

  // Get copy of a portion of the domain into local element pointed
  // to by dst - dst must be one of the elements in the local collection
  virtual DDD_Status
     Get(GlobalDataDescriptor<ELEMENTTYPE,DOMAIN> *src, ELEMENTTYPE *dst) =0;

  // Put a portion of the domain into destination from local element
  // pointed to by src - src must be one of the elements in the
```

```
  // local collection
  virtual DDD_Status
     Put(GlobalDataDescriptor<ELEMENTTYPE,DOMAIN> *dst, ELEMENTTYPE *src) =0;

};

// ****************************************************************
// ****************************************************************
// GLOBAL Distributed Data Descriptor
// ****************************************************************
// ****************************************************************

// Base class for all GLOBAL Descriptors
template <class ELEMENTTYPE, class DOMAIN>
class GlobalDataDescriptor {
 public:

  // **************************************************************
  // Initial DOMAIN for the original construction of this object.
  // Can be used for comparing two Descriptors, depending on the
  // properties supported by DOMAIN.
  DOMAIN BaseDomain;

  // Basic constructors for GlobalDataDescriptors
  // **************************************************************

  // Simplest Constructor.  Create new storage.  Call LDD Constructors
  GlobalDataDescriptor(DOMAIN);

  // Copy Constructor - from another DDD
  GlobalDataDescriptor(GlobalDataDescriptor<ELEMENTTYPE,DOMAIN> &);

  // Aliased Storage Constructor.
  GlobalDataDescriptor(DOMAIN, GlobalDataDescriptor<ELEMENTTYPE,DOMAIN> &);

  // Assignment operator (operator=) must be defined in subclass because
  // of template limitations

  // Mapping and translation function interfaces.
  // **************************************************************

  // Restrict LOGICAL domain to local LOGICAL elements
  // RESTRICTION: Domain=>list of Domain
  virtual DOMAIN* RestrictToLocal(DOMAIN) =0;
```

```
  // TRANSLATION: Domain=>LDD
  virtual LocalDataDescriptor<ELEMENTTYPE,DOMAIN>
 *TranslateToLocal(DOMAIN)=0;

  // Special case, where DOMAIN = this->BaseDomain;
  virtual DOMAIN* RestrictToLocal() =0;
  virtual LocalDataDescriptor<ELEMENTTYPE,DOMAIN>
 *TranslateToLocal() =0;

  // Translate LOGICAL domain to PHYSICAL
  virtual GlobalPointer toGlobalPointer(DOMAIN) =0;

};

// *****************************************************************
// *****************************************************************
// EXAMPLE: Matrix
// *****************************************************************
// *****************************************************************

// Array triplet for indexing into array
class Triplet {
 public:
  int lowerbound;
  int upperbound;
  int stride;
};

// A list of Triplets:
typedef Triplet* TripletIndex;

// *****************************************************************
// Defines a _PHYSICAL_ section (chunk) of an array
template <class BASETYPE>
class ArraySection {
 public:
  int rank;

/* Provides information about the logical "spacing" of elements in the
   address space, and how they mapped from logical to physical.  for
   example, a 2D "normal" matrix has an NextElementStride of: ( (1,0)
   (0,1) ), meaning you add (1,0) to an index to get to the next row.
   However, if the logical mapping was suppoed to be even rows, but
   regular columns, than the NextElementStride would be: ( (2,0) (0,1 ). */
```

```
  int *NextElementStride; // array of length <rank> * <rank>

  // A list of bounds in terms of Global Index names
  TripletIndex globalindex; // array of length <rank>

  // "layout" determines the physical layout of the array by dimension
  // This permits the compiler to be aware of column-major and row-major
  // layout patterns.
  int *layout;          // array of length <rank>

  BASETYPE baseaddress;
};

// ****************************************************************
// Distributed Matrix Data Descriptor
// ****************************************************************

// This is the representative data descriptor that must reside
// on each OwnerContext for a typical SPMD owner-computes computation

// The first template argument to both descriptors is an ArraySection,
// not a base type, because the local elements are more compactly
// defined as a collection of ArraySections, not individual elements
class LocalDoubleMtxDD
    : public LocalDataDescriptor<ArraySection<double>,TripletIndex> {
  public:

  LocalDoubleMtxDD();  // Basic Constructor

  // Add special fields here
};

// Global Distributed Matrix class.  (Still an abstract class)
class GlobalDoubleMtxDD
    : public GlobalDataDescriptor<ArraySection<double>,TripletIndex> {
  public:
  int rank;

  GlobalDoubleMtxDD();  // Basic Constructor;

  GlobalDoubleMtxDD& operator=(GlobalDoubleMtxDD&); // assignment operator

};
```