D R A F T

Document for the DARPA Data Reorganization Effort

DARPA Data Reorganization Effort

http://www.data-re.org

November 20, 1998

Table of Contents

List of Figures iii
Acknowledgements iv
Preface v
1. Introduction 1
2. Data Re-organization Examples 3
3. Data Descriptors 5
4. Data Transfer 8
5. Data Reorganization Algorithms 11
6. Interfaces 12
7. Special Topics 13
A.. References 14
B. Glossary 15
C. Data Re-organization Libraries 16
D. Example Application of the API 19
E. Objects 24
Index 28

Document contains 33 pages, I-v, and 1-28

List of Figures

Figure 2-1: Corner Turn's Effect on data distribution 3
Figure 3-1: Data Object 5
Figure 3-2: Table of possible groupings of partitioned dimensions over the Dimensionality of the global data. 6
Figure 4-1: Opacity of Transfer / Transpose Operation 8
Figure 5-1: Corner Turn Algorithm 11
Figure 5-2: Corner Turn Metric 11

Acknowledgments

To be filled in later.

Preface

The DARPA Data Reorganization Effort, funded by the Information Technology Office (ITO) under BAA9706 (Mercury Computer Systems, Inc. is the prime contractor; MPI Software Technology, Inc. is a subcontractor), is a focused effort aimed at providing specific attention on the problem of data reorganization in High Performance Computing, with some special emphasis on embedded High Performance Computing. In particular efforts to examine Application Programmer Interfaces (APIs), algorithms, and application requirements are in scope. Both "transpose" and "reshape" issues are in scope; both clique and bi-partite redistribution is in scope.

1 Introduction

1.1 Goals

In a parallel application, data locality is extremely important. The correct association of data with a particular processor at the right step in an algorithm has a direct impact on performance. Many parallel algorithms require extensive data reorganization between steps of the algorithm. One of the most classic forms is the "corner turn" reorganization of a 2-D array for the processing of SAR imagery. However, there is no well defined Application Programming Interface (API) to implement this and other complex data reorganization methods. If a standard API could be introduced to address data reorganization, the application programmer can concentrate on more important issues involved with complex numerical algorithms with the assurance the data reorganization is optimized for the application platform.

This document should provide a set of guidelines for standards committees to use in the definition of API's in forthcoming middleware developments. The guidelines specify objects that describe distributed data sets and operations on those objects that effect the data transfer. They constitute a pseudo-API to provide for optimal performance in the data reorganization effort. Some elements of the pseudo-API are best suited toward the middleware design while others apply toward the design of hardware and/or firmware.

1.2 Issues

There is much research into common but complex data reorganization methods. This research must be referenced to build a common base for data representation in a way that can enable the middleware to optimize the data reorganization for various numerical algorithms. Studies will include current data reorganization requirements, research into current solutions, and development of API design requirements to optimize the data reorganization for applications.

DISCUSSION:

Can hardware solutions be applied to help support the API (if an API is indeed developed as a result of this group) ?

1.3 Assumptions and Scope

Assume a data set D, a source process set P1, and a destination process set P2. The data set begins distributed over the source process set and ends distributed over the destination process set. By "distributed" we mean that each process is responsible for a particular piece of the data. This section details some basic assumptions about D, P1, P2, and the way that data is moved between them.

We consider cases where P1 and P2 are separate (that is, where the process graph is *bipartite*) and where they are the same.

Two processes may be responsible for the same part of D, in which case the data is said to be distributed in an *overlapping* fashion. Support for overlapping data is important for many applications.

In the initial version of this API, the user will explicitly describe the way in which D is partitioned over P1 and P2. A future version of the API may include tools which allow the user to set some guidelines and allow the API to vary the distribution according to those guidelines.

The process sets need not be homogeneous.

The API which supports data re-organization should be thread-safe in case multiple processes in the application run as threads on the same processor.

Data is assumed to be moved by some mechanism (for example, MPI) that is not explicitly described by this document. The API should support both distributed memory and shared memory systems.

For now, we assume that D contains dense data though we recognize other applications, such as ATR and sparse matrix manipulation, may also benefit from this body of work.

In the remainder of this document, we give examples of data re-organization algorithms that are in common use for various applications. Several APIs have been developed to support the data re-organization requirements of these different applications: these are listed in the appendix.

2 Data Re-organization Examples

Data re-organization occurs in a number of different applications. As a typical example consider a generic signal processing application. Assume that data arrives from four signal processing “channels” and that the data from each channel is distributed to a single processor. This might be done so that channel-specific operations such as equalization and filtering can be performed in parallel on the four processors. It is common in signal processing applications for operations that require data from different channels to be performed in the second step of processing. The data must be re-arranged to perform these operations. In signal processing, this is referred to as a “corner turn.” Figure 2-1 illustrates the distribution of a 4 by 8 data matrix among 4 processors before and after a corner turn.

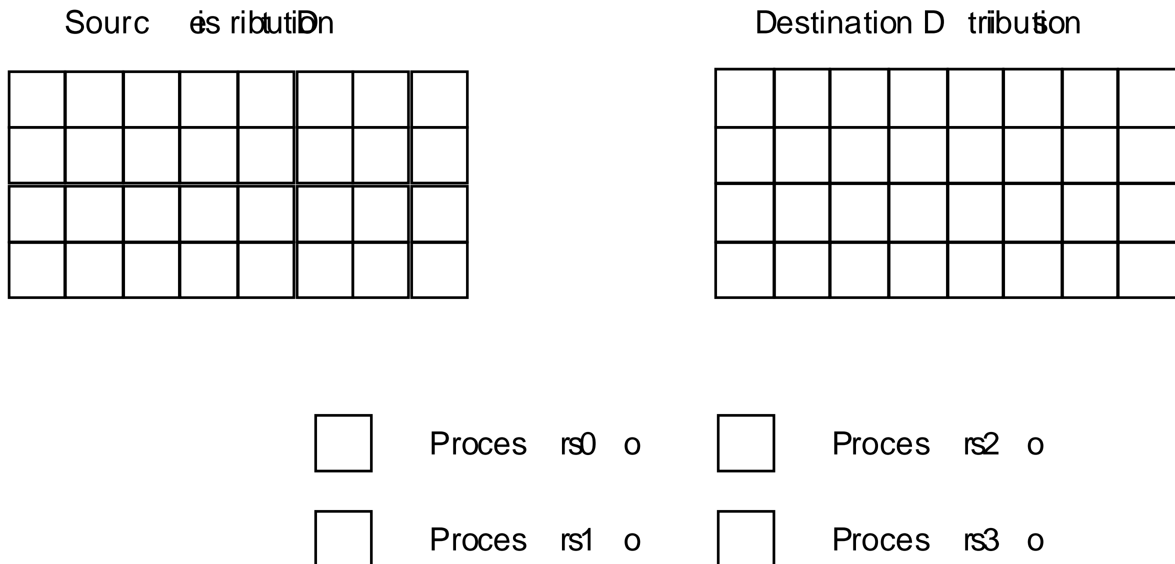


Figure 2-1: Corner Turn’s Effect on data distribution

The corner turn concept is simple, but the actual operation may put heavy requirements on system bandwidth and memory resources. This simple example illustrates many of the general issues associated with data re-organization.

1. The source and destination processors may be the same, or they may be different. There may be different numbers of processors in the two sets.
2. The source and destination distributions have different distributed dimensions.
3. Each destination processor must receive a message from several source processors, and put the data from that message into the appropriate place in memory. Obviously, the appropriate place changes depending on the size of the source and destination process sets and the distributed dimension.

4. If the destination matrix is stored in column-major order, then the data from the four source processors must be interleaved to produce the piece on each destination processor. This may require extra storage and copying, and may require efficient use of underlying hardware resources such as DMA engines.
5. Data will be streaming in continuously in many signal processing applications. The setup required for these operations should be done once, and consecutive operations should re-use resources.

Now extend the problem further to a full-blown signal processing application, in which multiple signal processing modes require different data sizes and different distributions of the data to achieve the required throughput. Clearly, one would like to have a common way for the user to set up and specify these transfer operations. In order to do this, the user must specify the overall data set and the source and destination processor sets. The layout or mapping of the data set onto the processors must also be described. The system, in turn, must take all this information and give the user a way to start and stop specific transfer operations. In the next section, we propose objects which meet these requirements.

DISCUSSION:

This section should include other examples which motivate overlap and show non-signal-processing applications if we know of any.

3 Data Descriptors

Assume a data set D, a source process set P1, and a destination process set P2. The data set begins distributed over the source process set and ends distributed over the destination process set. By "distributed" we mean that each process is explicitly assigned responsibility for a particular piece of the data. This document specifies data structures that describe D, P1, and P2, and the distribution of D onto each process set. As defined here, these data structures do not directly interact with the user's data set or control process objects: they are simply descriptors that the user creates based on the application's uses for the data. Once the descriptions of D, P1, P2, and the mappings have been made by the user, they are used to create the transfer object. This object stores internal data that the machine uses to perform the data transfer. The object-based approach allows memory allocation and control information to be hidden from the user.

In this section, we propose descriptor objects for the data set D, the process set(s) P1 and P2, and the mapping between them. In the next section, we propose a transfer object and associated operations.

As we saw in the previous chapter, the system needs a description of the data and its mapping to the parallel processor to optimize the data reorganization process. The key pieces of information are the size and shape of the global data object, the size and shape of the process set, a description of the partitioning, and a description of the local data organization. In this chapter, we propose objects to store this information. Each object is assumed to include defined operations to access each of its members, a constructor, and a destructor.

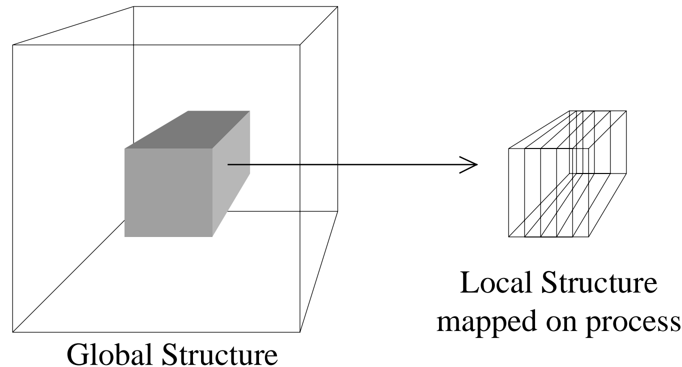


Figure 3-1: Data Object

3.1 Global Data Set and Process Set Descriptors

The global data set can have up to four (five?) dimensions. The descriptor must store the number of dimensions, the size of each dimension, and the size of the data elements

process set object is almost exactly the same, except that no data element size is required: it consists of just a number of dimensions and a size for each dimension.

The process set over which a data set is distributed may have any number of dimensions: for example, a large two-dimension matrix may be distributed over a three-dimensional process set, or a three-dimensional data set may be distributed over a two-dimensional process set.

DISCUSSION:

Should we address this as five dimensions (see notes)?

Should we limit our description of distributions to n-dimensional structures (n<=5) distributed over p-dimensional process sets for p<=n?

What is the relationship of the process set object to the underlying communication object, say, the MPI communicator or PAS pset?

3.2 Partitioning Descriptor

The global data set and the process set are tied together with the partitioning description object. Given the size of the global data and the number of partitioned dimensions, there is a maximum number of partitions possible. These partitions are named in the table below. In each dimension, an arbitrary block-cyclic partitioning should be possible: such a partitioning can be described with a single blocking factor. When the partitioning object is constructed, the user must specify the dimension of the process set over which each dimension of the data set is distributed.

For SAR and similar applications, it is desirable to allow processors in the same dimension to store some redundant data. The data on different processors is said to overlap. (It is suggested that data overlap should only be allowed for a block partition of the data cube.) Overlapping data has to be described with two parameters, a left and a right overlap, to allow for the possibility of an asymmetric overlap in a particular dimension. This could occur, for example, in support of fast convolution by the overlap-and-save method, where each processor needs a copy of some data belonging to the previous processor.EDITORIAL NOTE: Add pictures of data distribution methods, cyclic, block overlapped, tiled,

Dimension of Global Data # of Partitioned Dimensions	1	2	3	4	5
0	1	1	1	1	1
1	1	2	3	4	5
2		1	3	6	1

					0
3			1	4	5
4				1	5
5					1

Figure 3-2: Table of possible groupings of partitioned dimensions over the Dimensionality of the global data.

These partitions are named as shown in the following list.
(data, part)

(1, 0): WHOLE
(1, 1): PART_X

(2, 0): WHOLE
(2, 1): PART_X, PART_Y
(2, 2): PART_XY (TILE_XY or, simply, TILE)

(3, 0): WHOLE
(3, 1): PART_X, PART_Y, PART_Z
(3, 2): PART_XY, PART_XZ, PART_YZ
(3, 3): PART_XYZ (TILE_XYZ or, simply, TILE)

(4, 0): WHOLE
(4, 1): PART_X, PART_Y, PART_Z, PART_T
(4, 2): PART_XY, PART_XZ, PART_XT, PART_YZ, PART_YT, PART_ZT
(4, 3): PART_XYZ, PART_XYT, PART_XZT, PART_YZT
(4, 4): PART_XYZT (TILE_XYZT or, simply, TILE)

3.3 Memory Layout Object

The memory layout object describes local memory organization. It consists of a stride for each dimension of the global data object. We allow implicit stride specifications. For example, consider 3D:

PACKED_XYZ => x_stride = 1, y_stride = x_length,
z_stride = x_length * y_length

PACKED_ZXY => x_stride = z_length,
y_stride = z_length * x_length,
z_stride = 1

x_stride = PACKED_Y (= K * y_length)
y_stride = K
z_stride = PACKED_X (= K * y_length * x_length)

This permits fully packed and sub-tensor instantiations without needing to know the lengths of each dimension, the number of processors in the partitioning or the actual partitioning algorithm across the processors. When accessing an instantiation of a distributed data object (e.g., after a transfer), all fields are returned with the correct stride values.

4 Data Transfer

The data descriptors mentioned in the previous section enable the system to create a transfer object. Operations on this object start and stop the transfer and indicate the status. This chapter describes the transfer object and issues related to it.

4.1 Transfer Mechanics

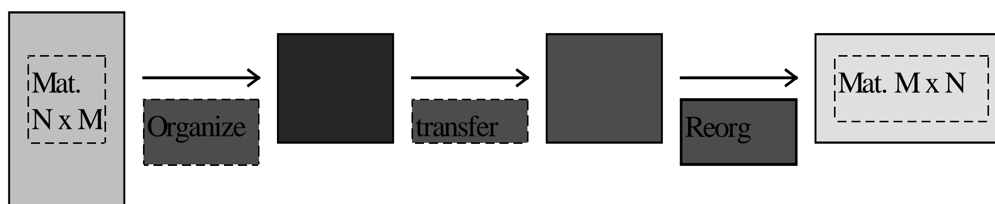


Figure 4-1: Opacity of Transfer / Transpose Operation

The transfer can be viewed as a three phase operation as shown above in Figure 2-2.

In the first phase, "organize", data is readied locally for transfer. This may involve copying to some data buffers which may have been pre-allocated. The second phase is where the interprocessor communication takes place. In the third phase, the results may be locally "re-organized" at the destination. The objects are opaque to the library in that contents of the matrix are not examined. The intermediate forms of the matrix should be opaque to the user.

The "organize" and "reorg" steps are under the API, but they are invisible to the user. The rationale for this is that on different architectures the best implementation or ordering of these operations may be different.

DISCUSSION:

(Should the original matrix still be accessible after the setup phase?)Doesn't the paragraph above directly contradict the paragraph below?

For performance, these three phases may be three separate calls. This allows the organize phase for one data transfer operation to be overlapped with the communication phase for another operation. Many times, a user will want to have a single call which either sets up the communication and sends the data or receives the data and rearranges it. In any case, it should be possible for the user to choose blocking or non-blocking versions of the transfer operation.

Two major classes of algorithms for transposition exist: direct send and store and forward. In the direct send algorithm, each source processor sends the data packet directly to each destination processor. In store-and-forward algorithm, messages are sent in several steps: data passes through an intermediate destination at each stage before arriving at its final destination. The best choice of algorithm for a given problem depends on the relative magnitude of communication overhead, communication bandwidth, and processor speed for the given machine. (We should provide a more detailed example, showing models of computation, communication, and copying, and a crossover point between the algorithms.) DMA engines can have a particularly beneficial effect on this process, allowing the main processor to proceed with other computation while the first and last copies for a direct-send algorithm (or the intermediate copies in a store-and-forward algorithm) take place.

4.2 Early Binding and Memory Allocation

If the communication patterns of a system are known ahead of time, then the implementation can make resource decisions before entering its operational mode. This is referred to as *early binding*.In order to transfer data, the system must know the data movement algorithm and the amount of buffer space required. To make these decisions, the implementation must be able to turn the transfer descriptions given by the user into a model of the time required for computation, communication, and copying operations, and the amount of memory required. A direct send algorithm can be chosen if copying is relatively expensive: a store-and-forward algorithm can be chosen if message overhead is high. Buffer space may also be the limiting factor: the sends may be done in a number of stages to preserve buffer space. There may be situations where it would be advantageous to lay out the buffer memory in a particular order for alignment reasons, or place the memory in a particular location (such as the SRAM in a DSP chip). All of these decisions may be made at early binding time, after the user has described the transfers that must occur but before they have started.

An advantage of early binding is the potential to re-use resources. If the system has a limited number of operating modes, one can potentially allocate buffers to fit the largest mode requirements and re-use those buffers for all the other modes. Routing information for the different modes would be stored separately. A purely late binding API forces the user to perform this setup manually.

DISCUSSION:

Of course, the downside of a purely early binding API is that in systems with hundreds of modes, storing a description of each transfer required in each mode may require too much memory.

DISCUSSION:

Do we need to concern ourselves with buffer allocations for handling data organizations?

Or is it better to have this done at higher layers? Should we develop this feature into the API? Another aspect is resource reuse, which might save time in overall facilitation of the data transfers. Another consideration is support for chaining or glueing of buffers during prolonged, large transfers.

4.3 Transfer Object

The transfer object must be created from a knowledge of the global data set, the source and destination processor sets, the source and destination partitionings, and the source and destination layouts. This knowledge enables the transfer object to be created with adequate buffer space and a plan for moving the data efficiently. The buffer space and the plan should be created at an early binding time.

DISCUSSION:

Can the user get at local information (e.g. how many elerments are on my node) from the transfer object after it has been constructed? It's a pretty common operation, and all the required information has been used in order to construct the transfer object.

5 Data Reorganization Algorithms

5.1 Data Transformations

The corner turn in concept is simple, but the actual operation puts heavy requirements of system resources. Therefore, the corner turn operation is a good consideration for data reorganization. The corner turn requires that each element in a sub-matrix is transposed and each sub-matrix is passed on to a different place in memory.

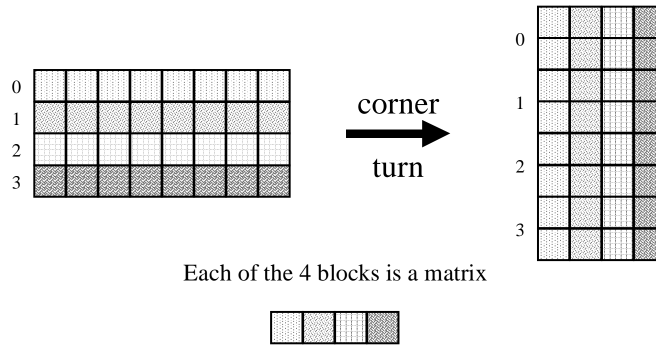


Figure 5-1: Corner Turn Algorithm

One particular example of a metric for this algorithm was developed to move and redistribute elements from one contiguous segment of memory into completely different contiguous segments of memory.

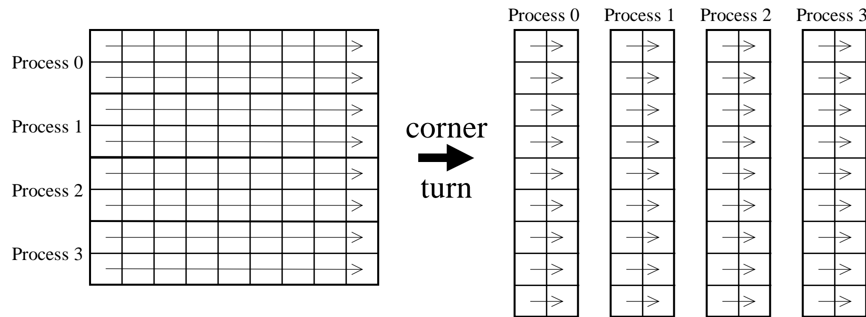


Figure 5-2: Corner Turn Metric

There are also other cases, where a single “slice” or several “slices” of a cube are extracted. However, in all cases, sub-partitioning of a larger dimensioned instance of data into lesser dimensioned instances will be done in regular Cartesian based partitions.

6 Interfaces

6.1 Summary

6.2 Targets

6.2.1 MPI/RT

6.2.2 MPI 1.X

6.2.3 MPI 2.X

6.2.4 CORBA

6.2.5 SPE (SPAWAR)

6.2.6 Parallel VSIP

6.2.7 PAS (Mercury)

6.2.8 SCL (Sky)

6.2.9 VIA

6.3 Languages

6.3.1 C

6.3.2 C++

6.3.3 FORTRAN 77

6.3.4 FORTRAN 9X

7_ Special Topics

Input strongly encouraged.

A: References

[1] Partow, P., D. Cottel, "Scalable Programming Environment," NCCOSC RDT&E DIV Technical Report 1672, Rev. 1, September 1995.

B: Glossary

Corner Turn
Early Binding
Late Binding
MPI/RT
Overlap
Stride
Tiling
Transformation
VSIP

C: Data Re-organization Libraries

System Identification

DFS -- Data Flow Shell, Honeywell Space Systems, Clearwater, Florida, John Samson, samson_john_r@space.honeywell.com.

KRI -- Parallel/Advanced Khoros, Khoral Research, Inc., Albuquerque, New Mexico, Joe Fogler, fogler@arc.unm.edu.

MIT/LL -- (name??) for ATR Image Processing, MIT Lincoln Laboratory, Paul Harmon, harmon@ll.mit.edu.

MPI -- Message Passing Interface Standard. http://www.erc.msstate.edu/mpi/

PAS -- Parallel Application System, Mercury Computer Systems, Chelmsford, Massachusetts, Jon Greene, greene@mc.com.

SPE -- Scalable Programming Environment, SPAWAR System Center, San Diego, Dennis Cottel, dennis@spawar.navy.mil.

STAPL -- Space-Time Adaptive Processing Library (moving vectors?), MIT Lincoln Laboratory, James Lebak, jlebak@ll.mit.edu.

Gedae -- system from Lockheed Martin, Bernie Schaming ,

Parti system University of Maryland, College Park, contact, e-mail@umd.edu

Feature Description

----

dimensions -- The number of dimensions of the overall data "cube" understood by the system.

module independence -- Has features that allow modules to be written so that their code is independent of the other modules to which they are connected. For example, DFS uses a function call interface, SPE and PAS have named ports.

reconfiguration -- The communication connections between modules can be reconfigured at (compile, load, or run) time.

fixed cube size -- The size of the data cube for any connection can be reconfigured at (compile, load, or run) time.

streaming -- Supports streaming data, that is, downstream modules can receive part of the communication.

overlap -- A distributed data part can include extra data from adjacent parts.

distribution -- Data can be distributed across processors using different methods: striped, replicated, or tiled. Replicated means that all processors have all the data.

redistribution -- Data can be rearranged from any one of the allowed distribution methods to any another, e.g., striped with overlap to replicated.

uneven distribution -- Distributed data is not required to divide evenly among a set of processors.

multiple input ports -- A module can receive distributed data from more than one source program.

multiple output ports -- A module can send distributed data to more than one receiving program.

processor topology -- Allows applications to think of the processors as interconnected in some topology, e.g., a two-dimensional mesh with neighboring processors to the north, south, east, and west.

pipeline support - A module can send one block of data to the next module in the sequence, while performing computaiton its next block.

SPMD support - The Single Program, Multiple Data model entails the same executable algorithm on each node, data is partitioned to each process, which computes its piece, forwards the results to a controlling process, and receives its next piece of the computation.

COMPARISON CHART

SYSTEM / Feature	DFS	KRI	MIT/LL	MPI	PAS	SPE (1)	STAPL	GEDAE	PARTI (?)
dimensions	3	5	?	?	2	2	2	?	?
module independence	Y	Y	?	N	Y	Y	Y	?	?
reconfiguration	R	R	?	R	?	L/R	L	?	?
fixed cube size	R	R	?	R	?	L/R	L/R	?	?
streaming	?	?	?	N	?	Y/N	?	?	?
overlap	?	Y	?	?	?	Y	Y	?	?
distribution striped	Y	Y	?	Y	Y	Y	Y	?	?
distribution replicated	?	Y	?	?	?	Y	Y	?	?
distribution tiled	N	N	?	?	?	Y/N	Y	?	?
redistribution	Y	Y	?	?	?	Y	Y	?	?
uneven distribution	?	Y	?	N	?	Y	Y	?	?
multiple input port	Y	Y	?	Y	?	Y	Y	?	?
multiple output port	Y	Y	?	Y	?	Y	Y	?	?
processor topology	Y	N	?	Y	?	N	Y	?	?
pipeline support	?	?	?	?	?	?	?	?	?
SPMD support	?	?	?	?	?	?	?	?	?

(1) For the SPE, the first answer is for port-to-port communication, the second is for intra-program data reorganization.

D: Example Application of the API

To help grasp the API more easily this example, though not readily applicable to signal processing, serves to illustrate the concepts and components of the Data Reorganization library. The second part of this message is an attempt to understand how a Data Reorg API could be used to implement this example a couple different ways.

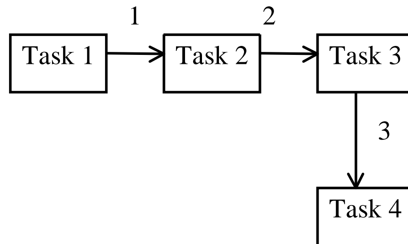
Example Description:

The first task reads blocks of data from either a real-time sensor or a disk file. The data are composed of a number of 16-bit samples of a time sequence, 800 samples per sequence. Each block contains 5000 of these sequences. Altogether, there are 200 blocks of data to be processed.

The second task takes a block of data from the first task and for each time sequence, filters, re-samples, and otherwise processes the data to produce sequences which now have 1024 32-bit complex values per sequence; there are still 5000 sequences per block. Each sequence can be processed independently.

The third task takes transposed blocks from task 2 and, for each 20 blocks, interpolates them into an image, producing a 1024x1024 image, one byte per element. Each row (i.e., a given sample across sequences) can be processed independently. The resulting image is transposed from the "standard" image orientation used by other downstream application programs and displays. Just to make this interesting, a fourth task takes the output of task 2 and writes it directly, block for block, to disk.

Functional Block Diagram



Data Shape Objects:

#1: x= 800, y=5000, eltsize=2 bytes
#2: x=1024, y=5000, eltsize=8 bytes
#3: x=1024, y=1024, eltsize=1 byte

Coding Examples

There are two distinct ways of writing this application: one, called SIMD here, in which a single program does each task one after the other; and the second, referred to as pipeline, in which there are four separately compiled programs running in parallel.

The 'reshape now" function call has been separated into two parts to unify the API for the SIMD and pipeline methods. Also, whether the user or the library provides the buffer address (who does the malloc) has been omitted. Arguments have been left out of the API calls (e.g., overlap) and other issues have been ignored to simplify the examples.

In the example code, functions in the Data Reorg API begin with 'DR_'.

SIMD Code

```
// Setup -- declaring all the data objects
// In all the object creation calls below, the first argument
// is the new object returned to caller.

// Declarations of Data Shapes the application will use.
// There is one object per connection source, or interconnected
// net as we used to say in the digital hardware business.

DR_create_data_shape_obj(shape1, X= 800, Y=5000, size=2)
DR_create_data_shape_obj(shape2, X=1024, Y=5000, size=8)
DR_create_data_shape_obj(shape3, X=1024, Y=1024, size=1)

// Declarations of the Partitionings needed by the various
// algorithms. There is one object per connection point for
// the various nets.

DR_create_partition_obj(part_task1out, PART_X, shape1)
DR_create_partition_obj(part_task2in, PART_X, shape1)
DR_create_partition_obj(part_task2out, PART_X, shape2)
DR_create_partition_obj(part_task3in, PART_Y, shape2)
DR_create_partition_obj(part_task3out, PART_Y, shape3)
DR_create_partition_obj(part_task4in, PART_X, shape2)

// Declarations of process sets.
// (There is only one process group/set in SIMD.)

DR_create_proc_group (process_group, ALL)



// Now that we know the process set, we can create the actual
// memory layout objects. Let's assume for now that the memory
// buffers that will be used by the application are malloced
// in here and provided as part of the layout object.
// Note there is one per connection point for the various nets.

// The layout objects can trace their lineage all the way back to
// the Data Shape objects, and therefore each layout object
// also contains all information about the connections needed
// to move data to the other end points of the same Data Shape
// object. So it doesn't seem that we need to have any further
// object type to describe each end point.

DR_create_layout_obj(layout_task1out, part_task1out, process_group)
DR_create_layout_obj(layout_task2in, part_task2in, process_group)
DR_create_layout_obj(layout_task2out, part_task2out, process_group)
DR_create_layout_obj(layout_task3in, part_task3in, process_group)
DR_create_layout_obj(layout_task3out, part_task3out, process_group)
DR_create_layout_obj(layout_task4in, part_task4in, process_group)

// Processing -- with reshape steps intermixed.
```

```
for each of 200 input blocks

// Task 1 is disk I/O using layout object layout_task1out
read_data_from_disk(layout_task1out)

// tell the Data Reorg library that data is now available in
// the form needed for output from task 1
DR_data_available(layout_task1out)

// we need data in the form for input to task 2
DR_need_data(layout_task2in)

// Task 2 -- do processing based on the task 2 input and
// output layout objects
"<compute -- use accessor functions to get object information such as number of rows to be processed>".


// data is now available in the form for output from task 2
DR_data_available(layout_task2out)

// we need data in the form for input to task 3
DR_need_data(layout_task3in)

// Task 3 -- incorporate the data block into the current image
// based on the task 3 input and output layout objects
< compute, compute, ...

if 20th buffer, an image is done
DR_data_available(layout_task3out)
// subsequent unspecified image post-processing goes here

// we need data in the form for input to task 4
DR_need_data(layout_task4in)

// Task 4 -- output to disk based on the task 4 layout object
write_data_to_disk(layout_task4in)

end for each input block
```

Pipeline Code

```
// Task 1 ------------------------------------

// Setup -- declaring the data objects

DR_create_data_shape_obj(shape1, X= 800, Y=5000, size=2)
DR_create_partition_obj(part_task1out, PART_X, shape1)
process_group = get_proc_group(MPI_COMM_TASK1)
DR_create_layout_obj(layout_task1out, part_task1out, process_group)
```

```
// Processing loop: read blocks from disk and send downstream

for each of 200 input blocks
< read_block from disk
DR_data_available(layout_task1out)

// Task 2 ------------------------------------

// Setup --
"//Setup": "How do other processes get the same initializing information?  Avoid all-to-all communications here."



DR_create_data_shape_obj(shape1, X= 800, Y=5000, size=2)
DR_create_data_shape_obj(shape2, X=1024, Y=5000, size=8)
DR_create_partition_obj(part_task2in, PART_X, shape1)
DR_create_partition_obj(part_task2out, PART_X, shape2)
process_group = get_proc_group(MPI_COMM_TASK2)
DR_create_layout_obj(layout_task2in, part_task2in, process_group)
DR_create_layout_obj(layout_task2out, part_task2out, process_group)

// Processing loop: process blocks
 while true
DR_need_data(layout_task2in)
< compute, process, etc. ...
DR_data_available(layout_task2out)


// Task 3 ------------------------------------

// Setup -- declaring the data objects

DR_create_data_shape_obj(shape2, X=1024, Y=5000, size=8)
DR_create_data_shape_obj(shape3, X=1024, Y=1024, size=1)
DR_create_partition_obj(part_task3in, PART_Y, shape2)
DR_create_partition_obj(part_task3out, PART_Y, shape3)
process_group = get_proc_group(MPI_COMM_TASK3)
DR_create_layout_obj(layout_task3in, part_task3in, process_group)
DR_create_layout_obj(layout_task3out, part_task3out, process_group)

// Processing loop: process blocks

while true
DR_need_data(layout_task3in)
< compute, process, etc. ...
if 20th buffer, an image is done
DR_data_available(layout_task3out)


// Task 4 ------------------------------------

// Setup -- declaring the data objects
```

```
DR_create_data_shape_obj(shape2, X=1024, Y=5000, size=8)
DR_create_partition_obj(part_task4in, PART_X, shape2)
process_group = get_proc_group(MPI_COMM_TASK4)
DR_create_layout_obj(layout_task4in, part_task4in, process_group)

// Processing loop: read incoming blocks and write to disk
while true
DR_need_data(layout_task4in)
< write block to disk
```

E: Objects

The data access object is defined with the following members:

Data Description

- Dimensions
- Sizes of each dimension
- Local storage
- Data types or element sizes

Data Distribution

- Index order
- Global permutation
- Number of processors
- Dimensionality
- Density

In the following description of a distributed data object,
I conveniently leave out all process[or] and buffer
(i.e. location) information.

Process Set Object

For describing a distributed data object, we minimally
require the number of processes (nprocs) in the set.
Nprocs can instead be made part of the partitioning
object (see below) but I think it's cleaner to leave
it separate.

We may decide its useful to define a cartesian process shape
(1D, 2D, ...) but this gets complicated so I will for now
assume a linear mapping of the data set onto a process set.

Process Set
nprocs
...

Data Shape Object

A data shape object simply describes a cartesian data set of

a particular dimension (1, 2, ...). Note that strides are deliberately excluded as this effectively determines memory layout (i.e. see below) which may be bound to a data shape object to describe the local instantiation of a distributed data set. The data dimension object does not require that the data set actually exist in any memory(ies).

Also note that I chose to include element size (in bytes) as this may be viewed as another dimension (I admit this is debatable).

DR_shape
dimensions (e.g. DATA_1D, DATA_2D, ...)
(can actually determine this by the first 0 length value)
element_size (e.g. 4, 8, ...)
x_length
y_length
z_length
t_length

Partitioning Object

The partitioning object defines how the data set is partitioned. It consists of a partitioning type combined with two overlap values (left and right) for each partitioned dimension (may be 0). We may want to insist that the left and right overlap values be the same (symmetric) when specifying the object. However, when accessing the object for a specific distribution (local instantiation), the two values neatly handle the edge cases (e.g. corner of an image) where they may in fact be different.

DR_part
[nprocs] (the partitioning object can exist without this)
partitioning type (e.g. WHOLE, PART_Y, PART_XY, ...)
x.left_overlap
x.right_overlap
y.left_overlap
y.right_overlap
z.left_overlap
z.right_overlap
t.left_overlap
t.right_overlap

Memory Layout Object

The memory layout object determines memory layout. It is defined by D element strides where D is the dimensionality of the bound data shape object.

DR_memory_layout
x_stride
y_stride
z_stride
t_stride

When specifying the object, we should allow the following sorts of implicit specifications. Consider 3D:

PACKED_XYZ => x_stride = 1, y_stride = x_length, z_stride = x_length * y_length

PACKED_ZXY => x_stride = z_length,
y_stride = z_length * x_length,
z_stride = 1

x_stride = PACKED_Y (= K * y_length)
y_stride = K
z_stride = PACKED_X (= K * y_length * x_length)

This permits fully packed and sub-tensor instantiations without needing to know the lengths of each dimension, the number of processors in the partitioning or the actual partitioning algorithm across the processors.

When accessing an instantiation of a distributed data object (e.g., after a transfer), all fields should be returned with the correct stride values.

So, continuing to ignore location (good thing I'm not a realtor), a distributed data object consists of a process set object (really, just nprocs), a data shape object for the "virtual" data set, a partitioning object and, finally, a memory layout object for the local piece (source) or piece(s) (destination) of the distribution.

A transfer object should include both a source and destination distributed data object and be constructed

prior to the transfer (early binding). It may allocate
local scratch memory for better performance
(e.g. packing and unpacking).

For performance, I think the API should have an "unpack"
call for "readying" the transfer (e.g. local transpose),
which may be a nop, a transfer call to be applied to the
unpacked local data (these two can certainly be combined
into one function for typical usage) and a "repack" call
by the destination process(es) (after the transfer) for
laying out the data as desired (may be a nop).

The repack (or separate access) call should give back to
the caller all the necessary information for accessing the
local data set. This should include the local address of
the buffer, the local shape, partitioning and memory layout
and, possibly, the global shape and global coordinates of
the start (min_x, min_y, min_z, min_t) of the local shape
within the global shape.

The memory layout is provided in both the source and
destination distributed data specifications to allow
the implementation to deliver the data as desired, in
which case the subsequent repack call does no data
reordering

Index

corner turn, 4