SDP Memo: Data Flow Prototyping Report

Regent and Legion as an example

Document Number…… SKA-TEL-SDP-0000083
Type…… REP
Revision…… C
Author…… P. Braam
Release Date…… 2016-04-08
Document Classification…… Unrestricted
Status…… Released

Lead Author	Designation	Affiliation
Peter Braam		University of Cambridge
Signature & Date:		

Released by	Designation	Affiliation
Paul Alexander	SDP Project Lead	University of Cambridge
Signature & Date:		

Version	Date of Issue	Prepared by	Comments
C	2016-04-08	Peter Braam	dPDR submission

ORGANISATION DETAILS

Name	Science Data Processor Consortium

Table of Contents

List of Figures

List of Tables

References

Applicable Documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, **the applicable documents** shall take precedence.

Reference Number	Reference
AD-01	SKA-TEL-SDP-0000013 SDP Element Architecture Design
AD-02	SKA-TEL-SDP-0000015 SDP Execution Framework Design

Reference Documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, **this document** shall take precedence.

Reference Number	Reference
Legion	http://legion.stanford.edu/
Regent	http://regent-lang.org
Slurm	http://slurm.schedmd.com/

1. Introduction

This document refines requirements of the SDP Element Architecture Design [AD-01], [AD-02] by providing more information about data flow, to measure its expressiveness, and some of its performance characteristics.

In Section 2, we give definitions pertinent to data flow, adhering conceptually as close as possible to the SDP Element Architecture, and establish the relationship with the Regent / Legion framework.

Section 3 contains a set of requirements, summarized as quality attribute scenarios that are given in a table.

Section 4, refines and studies some 15 of these scenarios through discussions of possible implementations.

Post PDR we expect to produce a new version of this Memo that contains code (described in one or more languages) and measurements, and which adds a final section describing how the refined scenarios are applicable to anticipated modules of the SDP software leveraging data flow.

2 Architectural overview of data flow

2.1 Overview of data flow programming

A data flow description of an algorithm uses a directed graph where edges are called channels and nodes are called actors. Channels represent the movement of data objects from one actor to another and possibly multiple channels transport data to actors, which represent computations on the data objects arriving on their input channels. In the model we study here, the readiness of the data triggers a computation by the actor, but alternative scheduling strategies exist. The concurrency model is that data objects are subject to either exclusive writes or shared reads which guarantees that all actors with input data available can execute concurrently. The outputs of the actor are data objects which are sent to other actors. Data objects are sometimes called messages or events.

Some actors are initial actors and the runtime system sends them a *start* message. Similarly, some actors inform the runtime system that they completed their computations to indicate partial completion of the computation described by the graph.

Many variations on the data flow model exist. For example, in some cases actors can perform actions when data has arrived on just one of their input channels, in other cases they require data on all inputs to be ready. If the system is to offer high availability, the detection that no data has been received or that an actor has crashed, must lead to intervention by the runtime system. Some data flow models allow for parallel computations in their actors, others restrict such computations to sequential ones.

While data flow graphs conceptually express the computations well, in practice the parameter dependency graph of a sequential program invoking functions on data objects allows the programmer to leave the management of edge endpoints and data movement to a compiler. Several programming languages such as Parsec and Regent [Regent] have found it more natural to leverage this use of the call graph of a sequential program to describe data flow.

In Regent and its associated runtime library named Legion [Legion], the functions are labeled with a keyword “Task”, instead of labeling them as functions, to indicate that they may be scheduled in parallel - they are in fact the units of parallelism in the Regent program. In Regent and Legion, the data objects are called Regions, and the instantiation of a data object in physical memory is called a Mapping.

Pipeline programs, such as those found in image extraction for radio astronomy are examples of programs where the functionality maps directly to a data flow description. The fact that the algorithmic models of image processing map directly onto the abstractions of data flow, make a data flow language an attractive choice for implementing image processing pipelines.

2.2 From Functionality to Parallelism

A key concern in data flow programming is to change a graph depicting a functional description to a data flow graph where parallelization takes place. Generally, such parallelism revolves around partitioning data objects. For example, for distributed parallelism, partitioned data objects can be moved in parallel across multiple channels to other actors, and actors having access to such partitions can perform computations in parallel. For local parallelism a single actor can leverage partitions for vector instructions, multithreading or the use of accelerators. We expect multiple kinds of parallelism, to be of importance.

When describing data flow with sequential programs, program transformation, a deeply explored subject, can introduce parallelism through analysis and/or through explicit hints.

In the case of SDP, at the highest level the ingested data should be brought close to computing elements responsible for processing the data - such sets of computing elements have been

labeled islands in the SDP literature. Further parallelism can occur within an island, again through subdivision of data made available to the island.

At present it is not specified up to what point the data flow paradigm for data movement and execution will be followed and at which points it will be replaced by a general purpose parallel programming language. Natural choices are (i) to make this transition at the level of a group of nodes required to execute a tightly connected message passing program, (ii) at the level of individual nodes or (iii) at the level of individual cores. A key consideration in making these decisions is to achieve portability for the programs as hardware architectures and algorithms evolve.

2.3 The structure of data objects

We will refer to data ready for use by our programs as ingested data. From a programming perspective it is valuable to regard the entire ingested data abstractly (i.e. without considering its location in a particular memory system), and following the terminology of Legion [Legion] we will call this a ***region.***

As a "type", regions are the maps from a space of indices or keys to a space of values. A ***partition*** of a region is a description of the indices of a region as a union of a set of possibly overlapping regions. A partition itself is a region, its indices labeling the subsets and its values the subsets of the region to be partitioned. By launching an actor for each element of the index space of a partition, parallelism is achieved. For use in the SDP the support of irregular data structures (such as arise in collections of visibilities) is helpful.

Access permissions of a region describe rules for concurrent access to overlapping areas. A distinction is made between read, write and access for reduction operations[1]. Further ***coherency*** hints can be given, indicating exclusive, atomic or simultaneous data access to a single region by tasks that are peers.

An important step before execution of actors is that the data must be available to the actors in a physical memory unit in the compute cluster. This involves performing a dependency analysis, and the execution of required predecessors. Further, the results of computations must be made available through so called ***mappings*** in physical memory for consumption by other computations. The availability of physical memory is a key resource consideration when creating Mappings.

2.4 Example

```
task sum(is : ispace(int1d), mults(is, float)) : float
where reduces+(mults) -- privilege specification/optimization
hint
do
      var sum = 0.0
      for i in is do
            sum += mults[i]
      end
      return sum
end
task dotp( is : ispace(int1d)
         , x : region(is, float), y : region(is,float)) : float
where reads (x,y)
do
      var mults = region(is, float)
```

[1] Although we have not found a reference for it, we understand the reduction permission to express that an associative, commutative operation such as typically used in reductions allows operating on subsets respecting this.

```
      __demand(__vectorize) -- optimization hint
      for i in is do
            mults[i] = x[i]*y[i]
      end
      return sum(is, mults)
end

local c = regentlib.c  -- for printf.
task main()
      var n = std.atoi(c.legion_runtime_get_input_args()[0])
      var is = ispace(int1d, n)
      var x = region_attach_hdf5("xdata.hdf", "data/x", is,
float)
      var y = region_attach_file("ydata.dat", 0, is, float)
      c.printf("dotp result: %f\n", dotp(is, x, y))
end
```

Example 1: A Regent data flow program to compute dot products

The extreme simplicity of the dotp task should not lead one to believe that its parallelization is trivial. Consider a commodity HPC cluster for its computation.

First of all, the ingested regions may be larger than the available memory. Then the runtime system will have to perform the dotp collective operation using partitions of the ingested data.

In a cluster the computation can be distributed over many nodes. On each node in a cluster, threading and vector instructions can be used to parallelize the (trivial) computation. If the cluster has many nodes, the aggregation of partial results may be performed to one's advantage using a tree reduction model.

2.5 The data flow environment

We see that a data flow environment may have many components. First it may compile a language expressing data flow graphs as distributed programs which use communication. A high performance communication system is typically complex by itself and must handle resource constraints in message queues. The compiler may use memory regions, partitions, deductive work and hints to transform programs into parallel programs.

The runtime of the data flow environment must be able to schedule actors and map memory regions into physical memory, managing resources carefully.

Several of the refined scenarios discussed in the next sections demonstrate the language's environment's ability to do this.

3. Data flow requirements

This section contains requirements that should be considered in the context of selecting a data flow system to implement the SDP software. Most of these reflect common best practice experiences in other programming environment, however, it might prove difficult to meet all of them, in which case awareness of what cannot be met is valuable. An alternative use of these requirements is that they provide a rich set of examples for implementation and evaluation. Such examples should prove valuable when designing the SDP system, and significantly lower the barrier to entry.

This is copied from a more easily maintained spreadsheet, available at https://docs.google.com/spreadsheets/d/1b9mLEPV9KZoZ0RNBFa3gmYuQeMU0tKWqn7IKS6Z4tPw/edit#gid=0

Number	Tag	Description
1	BUILD.MAKE	Software can be built with standard tools
2	BUILD.PACKAGE	Software can be packaged with standard tools
3	DEBUG.RUNPARTS	Run parts of pipeline in isolation Allow dumping intermediate results
4	DEBUG.TOOLS	Distributed debugging tools are mature: distributed and conditional breakpoints, stepping, variable inspection, back in time debugging
5	EXPR.CLIENT.SERVER	A client server pattern as in the cloud Haskell demo and with the client and server reversed
6	EXPR.DATA-DEPENDENCIES	Logical graphs allow loops and data dependencies
7	EXPR.DFGRAPH	The master node can export the data flow graph
8	EXPR.FAILOUT	A reducing actor can be notified or notice failure of nodes feeding it data and trigger actor completion upon receiving a subset
9	EXPR.FOREIGN-FUNCTIONS	Convenient foreign function interface
10	EXPR.GRAPHS.MAPREDUCE	The map-reduce data flow graph has several features worth verifying. It has a many-to-many mapper to reducer communication and it can behave very asynchronously - depending on implementation reducing actors may start working before all actors have even started. Some simple data flow languages cannot express this behavior. When load balancing is done late, the node graph requires dynamic scheduling.
11	EXPR.INGEST.ROUTE	Create a UDP packet filter that sends packets to a node based on the routing table created with OPTIMIZE.LOADB
12	EXPR.INGEST.SORT	Similar to INGEST.BIN but a sorting function may build up significant state in memory to perform sorting of the data, possibly per bin
13	EXPR.INPUT	Actors must support: precious input: must be processed, non precious: computation may proceed without it, non precious collection - a computation may proceed upon receiving a part
14	EXPR.IO.COLLECTION	I/O can read elements from a container with a collection and deliver subsets of elements repeatedly to an actor for processing
15	EXPR.KNOWN-GRAPHS	A known set of data flow graphs including map reduce, those used for calibration, hierarchical

		reductions and hierarchical spawning can be expressed
16	EXPR.LOOPS.COLLECTIONS	Loops in the data flow actors can be used to obtain elements from collections of objects when I/O throughput is not a good match for inter-actor message rates.
17	EXPR.LOOPS.SERVICE	Service requests can be handled in a service daemon, running in an actor
18	EXPR.MEMORY-SHARING	Memory regions can be partitioned with shadow regions and given to threads and processes.
19	EXPR.MESSAGES	Messages can have types
20	EXPR.NET.MULTIRPC	A node can perform parallel RPC's: all requests are dispatched and replies are handled asynchronously even while request sending is still in progress.
21	EXPR.NET.NODEID	Each node can communicate its network id, the network id's of nodes it can communicate with, and its parent
22	EXPR.NET.OUTPUTS	Actors can have multiple outputs and multiple inputs
23	EXPR.NET.PARALLEL	Messages can be sent in parallel to many child actors with waiting for multiple and blocking only to prevent overflows - if this is arranged by the runtime, its effects shall be clear to the programmer
24	EXPR.NET.SEQUENTIAL	Messages can be sent in serially to multiple child actors without overflow, with and without waiting for responses - if this is arranged by the runtime, its effects shall be clear to the programmer
25	EXPR.PIPELINES	Imaging pipelines including those with loops can be expressed nearly mechanically
26	EXPR.RESTRICT-GRAPHS	Through language mechanisms data flow graphs can be forced to have a restricted structure, such as fork join graphs, to keep programs easier to understand
27	EXPR.TREE-REDUCTION	Tree reductions can be implemented conveniently
28	EXPR.TYPEDACTORS	Actors can have types
29	OPTIMIZE.COMPOSE	When overhead of invoking separate actors exceeds the benefits the runtime system or language can combine the actors.
30	OPTIMIZE.KERNDATA	Run trial computations using a list of strategies for data partitioning and parallelization. Consider profiles and select algorithm to run on leaf nodes
31	OPTIMIZE.LOADBAL	Run computations for collections of input data, analyze profiles, create a load distribution over islands
32	PERF.BULK-NETIO	When data flow moves data 80% of raw network bandwidth can be achieved
33	PERF.IO.HDF5	ADIOS or HDF[2] files can be read at 90% of the performance of a low level benchmark
34	PERF.LOCAL	Local performance can be comparable with an MPI program
35	PERF.NET.EFFICIENCY	The network subsystem can execute standard network benchmark suites, such as those with all-to-many, each-to-neighbor, all-to-all achieving 80% of the efficiency of a low level network benchmark performing the same communication

[2] ADIOS may be the best performing, and HDF5 the most used data transport and layout libraries.

36	PERF.RTT-MESSAGE	Round trip message latency can be low, a sustained ping-pong sequence has latency lower than 20us
37	PERF.SCALE.ACTORS	Jobs can run at the scale of 1000,000's of actors, e.g. 1 actor / core on a currently very large system
38	PERF.SPAWN	Latency for spawning tasks is low and measured. < 100us
39	REGION.FFT	Large FFT's can be implemented efficiently as a distributed application.
40	REGION.INGEST.BIN	A daemon reading a UDP stream uses a function that may be updated frequently and appends data to specific data objects defined by the function. The function can be set at restart of the daemon.
41	REGION.NUFT	Sparse representation for a non uniform Fourier transform (NUFT)
42	REGION.PERMISSIONS	Regions support the notion of read, write and collective access
43	*Deliberately left blank*	
44	REGION.SUBDIVIDE	Memory regions can be partitioned, with support for irregular data as well as patterns, e.g. for suitable handling of locality among visibility data
45	REGION.TELESCOPE	Use of data layouts for gridding and other computations, tree like reductions, large scale data fanout, the possibility to re-implement automatic optimizations (as we did before), latency associated with task spawning and communication.
46	RUN.DETECT-EXIT	Death or exit of an actor can be detected
47	RUN.INSIDE-MPI	MPI programs can be extended to use the data flow runtime system
48	RUN.NET-OVER-MPI	The dataflow system can utilize MPI
49	RUN.NET.TOPOLOGY	The network subsystem has notions of its topology, such as neighbours, parents, and distance
50	RUN.RESOURCE	Tasks / actors can be spawned on specific cores and accelerators on any node in the cluster, with limited memory and other ulimit variables.
51	RUN.RESOURCE.AGGREGATION	Actors should implement streaming aggregation (to avoid too much state accumulating). Note that input for aggregation may arrive in any order. The correct collection concept for inputs is desirable.
52	RUN.RESOURCE.CLUSTER-ARCH	The system can export the cluster architecture
53	RUN.RESOURCE.NODE	schedulers receive information about resources from node level managers
54	RUNTIME.ARCHITECTURES	Target multiple architectures, including GPU and CPU systems
55	RUNTIME.HIERARCHICAL	Can start jobs hierarchically
56	RUNTIME.PROFILING	The system collects profiling data as specified in MS2 design
57	RUNTIME.SCHEDULER	The program shall allow a brief command line invocation including runtime resources to be interpreted by a runtime scheduler and resource manager such as Slurm
58	SCHED.INTRA-ISLAND-DYN	The system can dynamically schedule actors inside a data sub-island
59	SCHED.ISLAND-STATIC	The system makes a statically scheduled graph across islands

60	USABILITY.EXAMPLES	A collection of sample programs exist demonstrating all aspects of work required
61	USABILTY.MANUAL	A manual shall specify the properties of the data flow system precisely and in a manner usable by programmers

4. Refinement of Selected Scenarios

In order to assess and evaluate the ease of use, features and performance of potential data flow approaches for the SDP problem, it is proposed that candidate data flow programs are tested against certain criteria and problem implementations. In the sections that follow we lay out preliminary refined scenarios that represent a first step in this evaluation.

The data flow Environment should have Slurm [Slurm] Interoperability
Reference: RUNTIME.SLURM

A data flow program shall be runnable from the commandline on a single node, indicating how many cores, threads, accelerators and RAM to use. An extremely similar invocation without wrapping scripts shall start the program on a cluster using the SLURM job scheduler.

The command line arguments shall a subset of those supported by SLURM, and include:

1. number of nodes
2. number of processes per node
3. number of threads per process
4. memory per process
5. if a GPU accelerator is used
6. what network conduit is used through a `-net=` argument. If the `-net` argument is missing, the number of nodes must be 1 which will be assumed if the –nodes parameters is missing (the program shall then run on a single node).

To achieve such interoperability with the SLURM scheduler, the runtime system of the data flow environment must gather information available to SLURM - such as the resources allocated and their addresses - and make it available to the communication system and architecture management of the data flow environment.

4.1 Cluster awareness, Static Scheduling

The data flow system shall make the programs aware of the cluster architecture. A sample data flow program must:

1. Show it has found the nodes in the cluster
2. It will define a tree structure and elect and name the top of the tree the coordinator node where it will run a process at level 0
3. Start processes on all level 1 nodes
4. The level 1 processes start island groups of other processes at level 2
5. Pass through resource information to higher levels, e.g. the processes at level 2 can have access to 1...*N* cores per process, and *K* GB of RAM per process.
6. The cluster architecture and process tree with its resources is printed out by the coordinator

For example, use 13 nodes, a top node, 2 level 1 nodes and 5 leaf nodes for each level 1 node. Each node has 16GB of RAM, and 12 cores. Create 4 actors per node, each using 3 cores.

4.2 Messaging - bandwidth and latency

The data flow system shall use a high performance interconnect between processes. A latency and bandwidth study will demonstrate the utilization of the link. Reasonable performance might be indicated by a one way latency for small messages of 5 microseconds and utilization of 90% of raw bandwidth of the transport for messages of sizes bigger than 1MB.

1. The data flow system will spawn two processes that communicate in a client server fashion.
2. The client will send buffers of *K* bytes each at least 3 times, where $K = 1...2^{32}$.
3. When the buffer has been received, the server process will send a response to the client, upon which the client will send the next buffer.
4. The program will print out bandwidth and latency of the communication process. Average and standard deviation of the 3 measurements will be tracked.

4.3 Profiling information

Event and debug logs shall be handled as follows. Under /.df-logs the data flow system will create multiple directories. Filenames surrounded by square brackets will be substituted by values:

1. slurm/[jobid]/ - global parameters and output of the job here - like the nodes on which it is running
2. slurm/[jobid]/[network nodeid]/ - log files here
3. slurm/[jobid]/[task rank] - a symbolic link to the corresponding network-nodeid directory
4. local/pids/ - global parameters and output of the job here
5. local/pids/nodeids/logs
6. names/symbolic-links contain a string with [time-job-name-job params] and point to slurm jobid or process id directories
7. In this scenario we can rely on a shared file system to unify this into a global directory of log / debug / profiling information for the job.
8. the log files will leverage CPU performance counters, and a similar set of data for GPU.
9. The debug target and level can be amended dynamically.

Note: This requirement is overly specific and does not abstract the required qualities for a logging sufficiently.

4.4 Dynamic Scheduling

A data flow program shall be able to make runtime decisions concerning scheduling of actors.

Consider a program which is executing a computation repeatedly using a set of (identical) actors on M nodes, triggering the computation by sending data objects O_i , i = 1...N, to one of the nodes in order to start the computation and doing this K_i times for object i. The actors have a varying runtime T_i known to the program, depending on the data object sent to them to start a computation. The optimal outcome would be that the program completes after $\frac{1}{M}\sum_{i=1}^{N} T_i K_i$ An optimization calculation (brute force is acceptable) is performed to find a suitable distribution so that the actual runtime is close to the optimal one. The data flow program then schedules the transmission of data objects accordingly.

4.5 Binning irregular visibility data for parallelism

The data objects used by the data flow system support partitions for handling irregularly sampled collections.

Consider a one (or if easily done two) dimensional regular grid U containing N points. A function V on the grid is zero except at locations $V(i_j)$, j = 1...K, with K << N, i.e. on a sparse subset of the grid. Compute V'(i) = sum_j K(i,j) V(j), where K(i,j) = 0, unless |i-j| < r(j), with r(j) a simple, compactly supported function of j.

The aim of the demonstration is to partition the grid into intervals, called bins, B_n of varying sizes, each containing an approximately equal number of points where V is nonzero. Now let the computation of V' be done by one thread per bin B consuming points where V is non-zero in the bin B, doing this in parallel across a subset of bins (each element of the subset being allocated to a thread) and then sequentially for a collection of such subsets. The concept of partitioning the set of bins should prove applicable to organize the algorithm. The key requirement is to arrange the partition such that for each of the subsets of bins the threads contribute non zero values for V' to disjoint areas. This is achieved by suitable spacing of the bins in the subset, and as a result there is no need to perform atomic additions when computing the sum accumulating the results in the above formula in parallel.

A logical construction of a program with this parallelization and concurrency avoiding strategy using the data flow and its data objects is the key feature to demonstrate.

4.6 HDF5 inter-operability

A data flow program can start an actor upon completion of reading an HDF5 indexed set. The HDF5 fields will become the field values of a region. The program shall achieve I/O rates equal to 95% of a simple C program reading the values.

4.7 Resource Management

The data flow system shall handle resource management in its scheduling, for example:

1. A program will start 4 actors, two senders running on separate nodes, and two receivers running on a single node.
2. Each of the processes will be allowed to use *K* bytes of memory.
3. Each sender will send *K* bytes to each receiver, which acknowledges receipt and exits.
4. The program will demonstrate that if *K* is more than ½ of available RAM, the two receiving actors will be scheduled sequentially (to avoid deadlock).
5. When *K* is less than ½ of available RAM on the receiver, the two receiving actors will be scheduled concurrently.
6. The program will use the SLURM interfaces to control allowable memory consumption.

4.8 Algorithmic Expressiveness

A data flow program will be able to perform a data dependent decision, invoke different actors in the branches of the decision and exit. For example, a data flow program will approximate a square root using Newton's algorithm and exit when the square of the result is sufficiently small.

4.9 Scalability and bi-sectional bandwidth

The program shall start an actor on each core on a subset of the cluster for a total of at least 15,000 cores. A communication scheme among the actors will transfer data, demonstrating data transfer throughput approximating the bi-sectional bandwidth of the network connecting the actors.

4.10 Failout Actors

An actor named C collecting data from *k* other actors to compute C's output can be made aware of the liveness of its *k*-inputs and upon failure of up to *m* of the *k* inputs compute a partial answer without an error condition. (This scenario may depend on adding further features to Regent and Legion, and if so, this shall be noted and the scenario shall be implemented in pseudo code.)

4.11 Data Flow inside MPI

An MPI C or C++ program uses Legion Tasks and Regions to create data flow. (This is not a scenario that targets the Regent language).

4.12 Foreign function interfaces

This scenario refinement describes that foreign function call interfaces exist to execute inside actors:

1. The FFTW library
2. C++ with OpenMP
3. Functions defined with MPI programs
4. C++ with OpenACC
5. A Halide function

The demonstration should make it clear how mapped regions can be used by the libraries and languages indicated. Sample programs are indicated in the following subsections.

4.12.1 Calling FFTW

A Regent program reads HDF5 data and calls FFTW routine, then finds and reports peak frequency.

4.12.2 OpenMP kernel in Region program

Perform a dot product using OpenMP-parallelized loops and HDF5 data from HDF5 support.

4.12.3 MPI from data flow

A Regent program reads HDF5 data and calls an MPI code for distributed dot product.

4.12.4 OpenACC

Demonstrates a foreign function call in an actor, where the foreign function leverages OpenACC.

4.12.5 Halide

A regent program can call a Halide function effectively.

4.13 Support for distributed SMP CPU and GPU architectures

A data flow program can be created which optionally runs an actor on the GPU and optionally runs another actor on an SMP CPU. The sample program from the Example section is a good candidate. The emphasis is to demonstrate some automatic parallelization strategies for the computation leveraging different hardware architectures.

The program will demonstrate parallel computation on the architectures:

1. SMP vector instructions
2. SMP threads
3. GPU parallelism
4. Distributed parallelism

4.14 Data Transposition

A region with a two dimensional index space is used, which is mapped for use by a first actor, with a layout that is ordered row-wise (i.e. subsequent elements rows are contiguous in memory, different rows become contiguous segments in a linear address space).

A second mapping will be used by a second set of actors. The second set of *k* actors performs computations on *k* sets of columns of the Region. By mapping, or using a second region, the data is transferred such that a column oriented ordering is achieved in the second mapping or second region.

The program will demonstrate that the data flow language supports the creation of the two layouts with parallelism on architectures with:

1. SMP nodes
2. Distributed parallel computation