Evaluating Data Flow Execution Environments

Regent and Legion as an example

Document Number	SKA-TEL-SDP-0000083
Type	REP
Revision	C
Author	Peter Braam, Serguey Zefirov
Release Date	2016-04-08
Document Classification	Unrestricted
Status	Released

Lead Author	Designation	Affiliation
Peter Braam		
Signature & Date:		

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

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

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

Purpose

This document studies requirements of the SDP Element Architecture Design [AD-01], [AD-02] by defining a set of patterns and constraints that will be helpful to measure its expressiveness, and For some of its performance characteristics. It is seen as a first draft for evaluating 3rd party execution engines for the SDP.

After proposing a set of requirements and sample programs to study, we evaluate the Regent and Legion system from Stanford University following the approach.

Summary of the Conclusions

The Legion / Regent system appears to be general enough to capture key architectural drivers of the SDP architecture. The effort required to implement and profile (the majority of) the proposed test programs was far larger than expected.

1. The expressiveness of the Regent and Legion systems is sufficient to express the algorithms and parallelization strategies proposed. Regent programs are generally much shorter than Legion code, but are not concise.
2. Some of the performance results appear close to hardware limits (exploiting memory bandwidth and latency of messaging). For others no good configuration could be found.
3. The programming environment is complicated, partly because multiple languages (C++, Terra, Lua and Regent) interact with each other. Most guidelines and documentation had to be found in the source code.
4. Problems were encountered in approximately in ½ of the attempted efforts. Many were generally fixed rapidly by the Legion team. No single branch offered sufficient features.
5. A recurring theme in our exploration is the need for custom mappers (mappers instantiate virtual memory regions on physical resources). Any mention of user selected resources requires custom mappers. Mappers appear to have insufficient coordination to manage competing requests for resources automatically.

Organization of this report

In Section 2, we introduce the terminology we use pertinent to data flow, adhering conceptually as close as possible to the SDP Element Architecture. We establish a relationship with the Regent / Legion framework terminology as we will evaluate that as an example.

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

Section 4, refines some 15 of these scenarios suggesting possible implementations.

Section 5 Evaluates these for Regent and Legion, one by one. Code for this can be found on github and is cross referenced.

Section 6 Contains an overview table summarizing and comparing our experience with Haskell based data flow and Regent and Legion.

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 - parallel scalar products of vectors

The following program describes the dot product computation.

```
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
```

[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.

```
        var mults = region(is, float)
        __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, in particular to address concepts of ***compute and data islands*** found in the SDP 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 - this program mimics the estimations supporting data locality for pipeline computations.

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 shall be shown to support partitions for handling irregularly sampled collections with parallelism, demonstrated for a computation with similar side effects as those found in gridding of sampled visibilities.

Consider a calculation f(x) of an integer argument which is known to have some side-effects in such a way that it can't concurrently be applied to a pair of x_i and x_j, if that $|x_i - x_j| < D$ (some fixed constant, c.f. the width or support of the GCF). The side effect models the concurrent additions that may happen updating a single u,v grid-point when convolving nearby visibilities with the GCF.

Create a program that avoids concurrent computation through possible side effects take place forming suitable partitions of the data regions. A logical construction of a program with this data locality, 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 and serialize tasks for which resources cannot be obtained in parallel, 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

5. Evaluation of Selected Scenarios for Regent / Legion

Summary of the evaluation

Our suggested tests turned out to be remarkably difficult to implement in Regent and Legion. Nearly every program resulted challenges to very experienced programmers, and several dozen exchanges with the Legion team took place.

Requirement	**Met (y/n)**	**Performance achieved**	**Worked out of the box**	**Documentation**	**Help required**	**Comment**
build/install	yes	N/A	N	In source	No.	Dependencies and interactions with configuration parameters are highly complex Very good MPI support.
Cluster awareness	yes	N/A	N	In source	Yes	Access to the cluster descriptor variables from Regent was not well documented. Proper implementation requires tight coupling of Regent and C++ code (mapper). There are problems with accessing Regent task arguments (no such problem with Legion C++).
Messaging BW and latency	yes	Achieved a small percentage of physical BW	Y	?	Yes	Latency appears to be very good, bandwidth is ~200MB/sec on IB networks and inter-node - probably RDMA addresses eluded us.
Profiling	yes	N/A	N	In source	Yes	Profiling overhead is negligible. Profiling implementation is very good.
Dynamic Scheduling	yes	N/A	Y	In the doc below	No	
Binning	yes	N/A	Y	In the doc below	No	
HDF5	no	no		In source	No	Should work but we could not get any branch to work
Resource management	no				No	It appears that Legion runtime does not do resource management- instead of serializing requests which cannot be made in parallel program crashes.
Algorithmic Expressiveness	yes	N/A	N		Yes	No Regent language-level support for partitioning.
Scalability	no				Yes	This was attempted with matrix transposition; the program got stuck.
Failout actors	no	N/A	N			Requires changes in GASNet (lowest layer)
Data flow inside MPI	yes					This is achieved by the best known Legion program, S3D.
Foreign Function calls	yes	N/A	Y	In the doc below	No	Successfully called FFTW, OpenMP, MPI, C. Though we didn't ask for help doing this, we spent quite a bit of time digging in the Legion/Regent source code

						and test trying to understand how to access 1d dense region's underlying physical buffer. A lot of low level details are undocumented.
Support for distributed GPU architectures	no	Small percentage			Yes	Due to invocation of CUDA compiler on cluster nodes by Legion's runtime it was not possible to run code on GPU. On single nodes CUDA-demanded code works slower than baseline code without any optimization.
Support for vector instructions	yes	No	Y		No	1GFLOPS for single-precision numbers, this is ~5 times higher than the baseline, but well below hardware performance.
Support for SPMD	yes	(see comment)	N		Yes	Performance improves when SPMD optimization is demanded. Different number of requested SPMD threads gives different performance. Regent lacks the ability to extract task from inner loop so this optimization requires code change.
Support for distributed computation	yes	(see comment)	N		yes	Performance is slightly lower than for SPMD (which is expected). As with SPMD optimization requires code change.
Data Transposition	yes	Y	N	In the doc below	no	Initial C++ version (by Montse) used a task-per-point partitioning (10,000 tasks) and was very slow. Regent version was initially constructed to partition the whole task either by the number of distributed nodes used (16) or by the twice the number of SMP cores used on the single node (8). Local runs were successful from the very beginning but Wilkes runs were compromised by the bugs in our build/run system, which were fixed only quite recently.

Notes on the Regent language implementation

The most convenient way to program against the Legion runtime is to use the Regent language. Regent is written in Lua and Terra. Terry itself is written in a combination of Lua and C++, and can be meta-programmed in Lua. This combination is powerful, but may extremely easily be a source of numerous misunderstandings.

Combined with the fact that error reporting in Terra (and hence in Regent) may also be quite misleading and because Regent is quite poorly documented, we've decided to exercise restraint and follow a set of simple statements and rules which should prevent a casual user from being caught in numerous incompatibilities of the Lua/Terra/Regent trio, a few of which are explained in what follows.

Terra simultaneously extends the Lua parser, replaces the Lua host program (to be able to use this Lua parser extension) and introduces several constructs to provide a client with tools to build its own custom embedded language on top of this.

These facilities are limited at the moment:

- The Terra context is introduced by the single “terra” keyword;
- For a custom language embedded in Terra, the language context is introduced by a corresponding set of “entrypoints” -- keywords -- which the Terra parser recognizes in the Lua parsing (top-level) parsing state. Technically this is represented by the table of values for the “entrypoints”, which are keys of the table returned by the language module. For Regent these entry points are "__demand", "fspace", "rexpr", "rquote", "task";
- All embedded language parsers are completely separate from each other and steal the control right after Terra detects any of their entry-points. When it recognizes the end of custom language context it returns control to Terra parser (in Lua parsing state);
- Inside its context, Regent tries to mimic Terra as far as possible, but some Terra constructs don’t work exactly the same way as they do natively in Terra or don’t work at all (Terra globals). Two explicit deviations from Terra found in Regent are (i) usage of the `rexp ... end construct` instead of a backquote mark for expression quotation and (ii) use of `rquote` instead of the `quote` keyword for statement quotation. In Terra the `quote ... end` construct can be used for expression quotation, but in Regent the `rquote ... end` construct cannot be used for expression quotation.

Terra and Regent are staged languages but stage separation isn’t fully lexically mandated (as it is in Template Haskell for example) with numerous implicit conversions and splices (for example all Terra variables are implicit splices), which blurs the boundaries between the code executed in different stages. A novice might be well advised to explicitly use the splicing (`[...]` construct) in the tricky parts of code even if an implicit splice is possible to have cleaner understanding of what stage the code belongs to and to quickly understand if quote/splice usage is balanced. We also are under the impression that Regent does less implicit splicing than Terra, thus making metaprogramming more explicit albeit slightly more verbose.

Regent language and Legion runtime opportunities for improvement

Tasks and mappers are separated

Regent program tasks and C++ Legion mappers are weakly coupled - changes in a Regent program are not checked during compilation of the C++ mapper code. This can lead to inefficient execution and make writing efficient Regent program a challenging task. In fact, it is not practically possible to write a non-trivial effective Regent program without creating several profiles.

Mappers appear not to manage resources

The Legion and Regent literature present mappers as an efficiency feature which preserves program correctness. Our Resource management experiment shows that a mapper which assigns too many resources to a single node can lead to program crashes.

Mappers do not have view of global scheduling state

Existing mappers appear to be unable to make choices based on information that references the global scheduling graph state.Existing examples of load balancing mappers just rotate CPUs, selecting them in round-robin fashion. Because mappers hold references to “machine” runtime object and some other variables, a custom mapper doing so could be constructed.

This precludes use of mappers as an instrument to schedule resources. It also precludes mappers to make a resource-aware decisions.

Legion runtime does not employ distributed state machine mechanism

Utilization of the PAXOS algorithm (or other alternatives) would make Legion more predictable and efficient. Right now it is possible for failed program to run until SLURM terminates it upon a timeout. Signals to shutdown are not handled, nor propagated.

The utilization of PAXOS would also enable failout actors. Right now they cannot be implemented.

Regent does not provide some useful optimizations

As an example, bodies of loops attributed with __demand(...) pragma are not automatically factored into tasks.
This lack of automatic action leads to unnecessary changes in code being optimized:

- The loop bodies should be transformed into standalone tasks. Variables and regions have to be moved into parameters.
- Regions should have right privileges - putting everywhere “reads(my_region) writes(my_region)” or just omitting privileges will not suffice as performance will degrade.

So adding an optimization pragma to a loop does not necessarily lead to better performance.

Possible improvements for mappers and linking.

Improving mappers

Mappers should be part of Regent language and they should be *declarative,* expressed as rules, not as *imperative* code. For example, a mapper for the static scheduling demo program can be expressed as a simple mapping from name of task and its arguments to node (task) index.
For an example of successful use of rules for computing in stateful system one can look at Bluespec SystemVerilog.

Legion Mapper is an object that receives request to schedule resources to a computation or a memory object. This object can have (and often has) a state, over which it operates. For example, round-robin CPU scheduler should have a counter it increments during scheduling attempt so that each task will have it’s own CPU to compute; it also often has a set of CPUs and memories available that is obtained during object’s creation.

Bluespec SystemVerilog is a rewrite-rule based (hardware and software) system specification language to specify discrete logic hardware which expanded into software construction and verification area.

Bluespec SystemVerilog (BSV) module (some background is here, including an example of a module on page 14 with explanation is at page 16, lower half; more info can be found here) is a stateful system that has state and operates over input variables (of a method) and state to get output values and updated state. The output and updated state is computed using rules - named predicated blocks of code. The operation of set of rules is atomic, “first matching rule fires”. Rules are specified in a priority order. BSV compiler unrolls modules’ hierarchy to get a view of a complete state and uncover all access conflicts. If there are rules that do not conflict over resources, they can be fired in parallel, updating parts of state simultaneously.

The similarities of Mapper and Bluespec SystemVerilog module are as follows:

- Both BSV modules and Mappers have state. The state is used and gets updated during operation.
- Both BSV modules and Mappers have several entry points - interfaces’ methods for BSV module and methods for Mapper. These entry points use and update shared state.
- Operations in BSV module and Mapper methods should be atomic.

Mappers appear somewhat simpler than BSV modules - they are rarely grouped hierarchically (but they can - consider resource management as primary or secondary scheduling decision parameter). They also somewhat more complex, as they can use arbitrary expressions (including functions) for predicates and computations.

Right now Mappers work each on their own node and do not exchange information. This is OK for simple mappers that cannot exhibit diverging or bottlenecking behavior - an example is static scheduling like “put this task on that node”. We will see in a resource managment example, that mappers do not sufficiently coordinate to serialize allocation of memory when insufficient resources are available to map two regions concurrently. It is also necessary for mappers to communicate in cases like the aforementioned round-robin Mapper: they can send all tasks to single CPU, if counters are happen to be the same, which would create a bottleneck.

The expression of Mapper as a set of rules allows to detect diverging behavior and prevent it or exploit lack of divergence for faster operation. It also allows fine grained deadlock-free locking of Mapper’s resources for distributed computing of schedules.

Improving compile and runtime support

It would be better if Regent would generate C++ code, possibly annotated with source Regent code lines.

Right now Regent uses C bindings to C++ runtime, generated through LLVM. This complicates things and (i) does not allow easy examination of the code, (ii) does not allow to use examples of C++ idioms to analyze the resulting code and (iii) limits usability, for example, the Regent runtime has to execute the CUDA compiler on cluster nodes (where it is not commonly installed) to make GPU’s available.

Regent allows to export object files and this feature opens up the use of CUDA. But first two points above are still valid.

Building and running Regent programs

Reference to “cluster” in the following is the HPCS Wilkes cluster.

Building environment and prerequisites

Scripts to build and clean environment are located at <path to RC repository>/MS7/libs/scripts/ directory. There are two scripts: setup-cluster-ibv-SLURM.sh and clean.sh.

The setup-cluster-ibv-SLURM.sh script builds libraries and executables required in the build and run steps of our individual programs. The clean.sh script deletes everything built by the previous script, cleaning the environment.

Th setup-cluster-ibv-SLURM.sh script supports the following options:

- -no-ibv/--no-ibv - disables IBV support. By default, scripts will try to build two versions of the same program: <exec>-local and <exec>-ibv. Former can be used locally, even outside of cluster environment, while latter meant to be run on cluster nodes. By disabling IBV support only the local version will be built.

The setup process requires the following programs to be available in the environment:

- gcc v4.9.1 or higher or compatible (Intel MPI compilers). The cluster’s gcc 5.2 module works great, as do Intel’s MPI compilers.
- MPI implementation. OpenMPI or Intel MPI can be used. Intel MPI v5.1 is preferred on cluster. It can be loaded into cluster environment by issuing command `module load intel/impi/5.1.3.181`.
- Mention should be made that `modules` is not a true command, but alias and is unavailable in non-interactive shell mode, thus using it in a shell script might require to run the shell in interactive mode (`#/bin/bash -i`).
- Cmake, v3.2+
- Optionally, CUDA and HDF5.

Detailed user instructions

One needs to be aware of the shortcomings of Wilkes when building software:

- A user shell script (which is called from local user's shell script) loads the default set of modules which are very old; e.g. it loads modules for Intel C/C++ compiler v12 and Intel MPI v4;
- Modules for many modern packages are broken, e.g. the module for latest minor version of Intel C/C++ compiler 16 is broken;
- Thus, for example, if we manually load the module for ICC 16 it has some environment variables missed (or set to wrong values) thus leading to that wrong versions of libraries (from ICC 12) are picked giving us mysterious build errors; OTOH, if we unload ICC 12 module manually, we also receive linking errors now because the linker can't find necessary libraries at all. But this is better because we now clearly understand where the problem is;
- Thus, the typical build script for any Legion/Regent-related project on Wilkes should approximately look like:

  ```
  #!/bin/bash -i
  module purge
  Module load slurm
  ## module load cmake/3.4.3
  module load cuda/7.5
  ## module load fftw/intel/64/3.3.3
  module load python/2.7.10
  module load gcc/5.2.0
  module load binutils/2.25
  module load intel/cce/16.0.3.210
  ## Fix broken intel/cce/16.0.3.210 module, it lacks LIBRARY_PATH
  export LIBRARY_PATH=$LIBRARY_PATH:$LD_LIBRARY_PATH
  module load intel/impi/5.1.3.181
  module load hdf5/impi/1.8.16
  <do your job here>
  ```

  Things to pay attention to:

  1. We use bash in interactive mode (`#!/bin/bash -i`) to make `module` command alias work;
  2. We purge all default modules and explicitly load all needed (and most recent existing on Wilkes);
  3. We manually fix broken module to `intel/cce/16.0.3.210` make it work
  4. We use latest versions of software presented on Wilkes ATM (June 7, 2016);
  5. LLVM 3.5 or 3.6 needs to be patched to be successfully compiled with gcc 5+; we provide this patch along our build script

After successful execution of setup script it will print the following message:

```
Please add the following definitions into your environment:
SDP_BUILD_DIR=/home/user/long/path
SDP_SCRIPT_DIR=/home/user/other/path
SDP_USE_IBV=0
```

You should add these three variables into your bash_rc or other file that sets up your environment on login.

Comments on Legion in the Wilkes cluster environment with MPI and Slurm

The user who attempts to build and run GASNet/Legion/Regent program has to note that the GASNet job-start scripts (gasnetrun_*) are needlessly complicated and have problems with Regent programs. While these scripts may be useful when SSH spawning is used, they appear to complicate MPI base spawning and their value is unclear. The problems are close to irreparable and related to interaction between job-start script, the Regent runtime in Terra and the Legion/Realm/GASNet runtime (in C++).

The GASNet builds found on the cluster either have no MPI support (gasnet/nompi/1.26.0) or OpenMPI support (gasnet/ompi/1.26.0). Legion can be build against different GASNet. The Legion build found on the cluster is legion/200416. The build of Legion in the cluster module list appears to have been built against gasnet/nompi; this is seen from "strings liblegion_terra.so | grep GASNetConfigureArgs". As there are two builds of gasnet, there should be two builds of Legion.

We have not tested whether it is possible to run OpenMPI programs using srun (from SLURM) on the cluster. If it is possible, that would be good. If it is not, then we would like to advise building GASNet against Intel MPI. Our build does that and our SLURM submit script sets up Intel MPI's environment so that MPI library uses SLURM Process Management Interface (PMI) implementation.

Building demo programs

Demo programs are located at <path to RC repository>/MS7/programs/ directory. Each demo program occupies its own directory.

Each program directory contains README file indicating executable file name and Makefile to build it. There are several make goals:

- make clean - deletes everything built for demo program to run
- make or make <executable name> - builds local (UDP) and (optionally) IBV executables and main demo executable.
- make run - (optionally) builds executable and runs it. This command has several useful environment variables that affect execution of the demo program: `NODES=n TASKS=t THREADS=th TASKMB=m GPU=gpu NET=net make run`. List below explains each variable in detail
    - NODES=N - allocate N cluster nodes for the application run. Local runs should have N=1.
    - TASKS=T - allocate T processes for application.
    - THREADS=TH - allocate TH cores per task.
    - TASKMB=MB - tell GASNet runtime to allocate MB megabytes of RAM per task.
    - GPU=gpu - just enables CUDA for building and running the application.
    - NET=(|ibv) - run application on cluster using Infiniband (NET=ibv) or locally using UDP (NET=). Preferred way is to run on cluster, because some of Legion features do not work in UDP runs.
    - PROFILING=(1|0|) - enable profiling
    - DEV_ENV=1 - removes addition of Intel optimized primitives library (libirc), useful for runs on developer’s machine (you can omit it when on the cluster login head).
- make process-profile - runs Legion profiling tool over log files specified in PROF_FILES environment variable. Example:

  PROF_FILES=”~/.dflogs/JOB_ID/*/*” make process-profile

  This command line will process log files from SLURM job JOB_ID and create files “legion_prog.html” and “legion_prof.svg”. The HTML file can then be opened and will display SVG file with profiling graphics.
  It is possible to specify flags for profiling tool with PROF_ARGS environment variable:

  PROF_ARGS=-h make process-profile

  The line above will display usage message.

The build system is build on top of Legion build system and is perfectly compatible with it. This allows to use it to run performance tests from Legion itself, with profiling information.

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 constitute 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).
7. Requested CPU clock time

Implementation

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

To share this information SLURM with Regent programs, we created a simple Terra library, which wraps reading of SLURM environment variables and parsing of the results. We considered this task as an excellent didactic opportunity to show how Terra entities are Lua values, how Terra interplays with Lua and how can Terra easily be meta-programmed from Lua.

The code can be found here:
https://github.com/SKA-ScienceDataProcessor/RC/tree/master/MS7/programs/01-Slurm-interoperability.

We first define two local Lua tables containing SLURM environment variable names for which we want to generate query functions. The second table contains the names of variables with integer values, thus we immediately convert them using get_env_int function defined in auxiliary module getenv.t. The first table contains all remaining variables when values either are strings or for those that don’t need any further processing, or should be further processed in a specific way for each variables:

```
local slurm_env_str = {
    "SLURM_CHECKPOINT_IMAGE_DIR"
  , "SLURM_CLUSTER_NAME"
  ……...
  }

local slurm_env_int = {
    "SLURM_ARRAY_TASK_ID"
  , "SLURM_ARRAY_TASK_MAX"
  ……...
  }
```

Then we define a global table which will keep all generated functions and generator function, taking SLURM env variables names list, return type (which is a Lua value), and environment variable processing function (which also is a Lua value) :

```
slurm = {}

function make_funs(env_table, rettype, fun)
  for _, e in pairs(env_table) do
        local j
        _, j = string.find(e, "_")
        local fname = string.lower(string.sub(e, j+1))
        slurm[fname] = terra() : rettype return fun(e) end
  end
end
```

For each environment variable name this function generates Terra function which process environment variable with this name. The name of terra function itself is generated as follows: from environment variable the prefix till the first '_' (in our case it can be 'SLURM_' or 'SLURMD_') is stripped and remained part is lowercased.

And finally we generate the required Terra functions:

```
make_funs(slurm_env_str, rawstring, get_env)
make_funs(slurm_env_int, int, get_env_int)
```

We see how Terra types and functions are Lua values. Terra type `int` is Lua value `int`, Terra functions `get_env` and `get_env_int` are corresponding Lua values too.

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.

Overview

There is only one way to achieve desired functionality. We should create 4*13 SLURM tasks on 13 nodes, 4 tasks per node. We give each task a low-level (Realm) runtime option to use 3 cores (3 pthreads threads). This way each actor will be bound by operating system in the number of cores to use. We then will spawn 4 top-level tasks (actors), each on the CPU on a top-level task cluster node. Each top-level task will then spawn two level 1 task on different level 1 nodes, which will spawn 5 level 2 tasks on each of level 2 nodes.

Regent program source code

Below is main program code in Regent:

```
-- Static scheduling demo.
--
-- Copyright (C) 2016 Braam Research, LLC.

import "regent"

local support
do
    support = terralib.includec("static-scheduling-support.h",{})
    terralib.linklibrary(os.getenv("SDP_SUPPORT_LIBRARY"))
end

task bottom_level(cpu, level_1_index : int, level_2_index : int)
       support.node_log("Bottom level task %d/%d/%d, SLURM node %d, SLURM task %d",
                   cpu, level_1_index, level_1_index, support.current_slurm_node(),
support.current_slurm_task());
end

task level_1_task(cpu : int, level_1_index : int)
       support.node_log("Level 1 task %d/%d, SLURM node %d, SLURM task %d",
                   cpu, level_1_index, support.current_slurm_node(),
                     support.current_slurm_task());
       __demand(__parallel)
       for level_2_index=0, 5 do
               level_2_task(cpu, level_1_index, level_1_index)
       end
end

task level_0_task(cpu : int)
       support.node_log("Root task %d. SLURM node %d, SLURM task %d",
                   cpu, support.current_slurm_node(), support.current_slurm_task())
       for level_1_index=0, 2 do
               level_1_task(cpu, level_1_index)
       end
end

task start_task()
       support.node_log("starting everything");
       -- ask runtime to not wait while we are working.
       __demand(__parallel)
       for cpu_index=0, 4 do
               level_0_task(cpu)
       end
end

-- Register mappers and setup support vars.
support.register_mappers()

-- start main work.
regentlib.start(start_task)
```

The text is straightforward - spawning level 0 tasks, then spawning level 1 tasks and, finally, spawning level 2 tasks. CPU and indices within level 1 and level 2 are kept as parameters for logging purposes and are very important for demo to work.

The actual work of scheduling is in the mapper code and it is tightly tied to the Regent source code.

Mapper source code and discussion

The code takes Task pointers (task to schedule) and should find a Legion CPU to schedule it to. The interested reader can read the corresponding part of mapper's logic here.

The Task structure contains a pointer to buffer with arguments passed at the time of Task invocation. We decode arguments and use the fact that the cpu is always first argument, level_1_index is second when present and level_2_index is fourth.

If we decide to change ordering or number of arguments we have to change our mapper code.

Then we search through CPUs available, selecting first that satisfy the criteria: the Legion node number must equal the SLURM task number obtained for specific task in the specific tree position. We then return that CPU as a CPU to schedule to.

Please note that this code relies on the fact that Legion node index is equal to SLURM node index (which is the case for MPI spawner and SLURM PMI library) **and** that SLURM schedules tasks in order (tasks 0..3 are put on node 0, tasks 4..7 are put on node 1, etc). If either assumption fails we have no easy way to correct that.

Issues

It should be noted that Regent program with the mapper does not work. It looks like Regent does not pass arguments as Legion does[3]. The Legion program with similar functionality should work.

It should be noted that there is no easy way to communicate scheduling structure between mapper and Regent program (or Legion program for that matter).

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.

Messaging latency

There is a message latency performance measurement program in Legion tests: event_latency.cc. It measures merging of events inside tasks - a task is spawned, its goal being to merge two events and spawn an event and another task when a decrementing count is not zero.

I should note that it is more than "just spawn an event and wait for it to fire". It spawns tasks, rotating CPUs - each new task will be spawned on different CPU, including CPU on different nodes. If no special care is taken, for two tasks on two nodes all events will cross nodes.

The following latencies were found:

1. Intra-node (single task - single CPU) events delay is 1.005+-2 microseconds.
2. Inter-node events delay is 5.50+-0.5 microseconds.

Messaging bandwidth

The program is short and can be seen here. It's main task on first node creates a region (message) with ever increasing size, fills it with zeroes and spawns a modifier task on second node with the region as parameter (effectively sending a message to another node). The region is checked that it has been modified (effectively first node receives message back). The roundtrip time then printed in the log.

Round-trip times change with the size of message. Up to 1MiB sized message times are close to random - they may decrease with increased size. After that round-trip times display behaviour consistent with message size. Fig.1. Below displays complete picture of bandwidth and size relations (notice very low bandwidth for size 1024 - at the very start).

[3] Mailing list discussions did not resolve this.

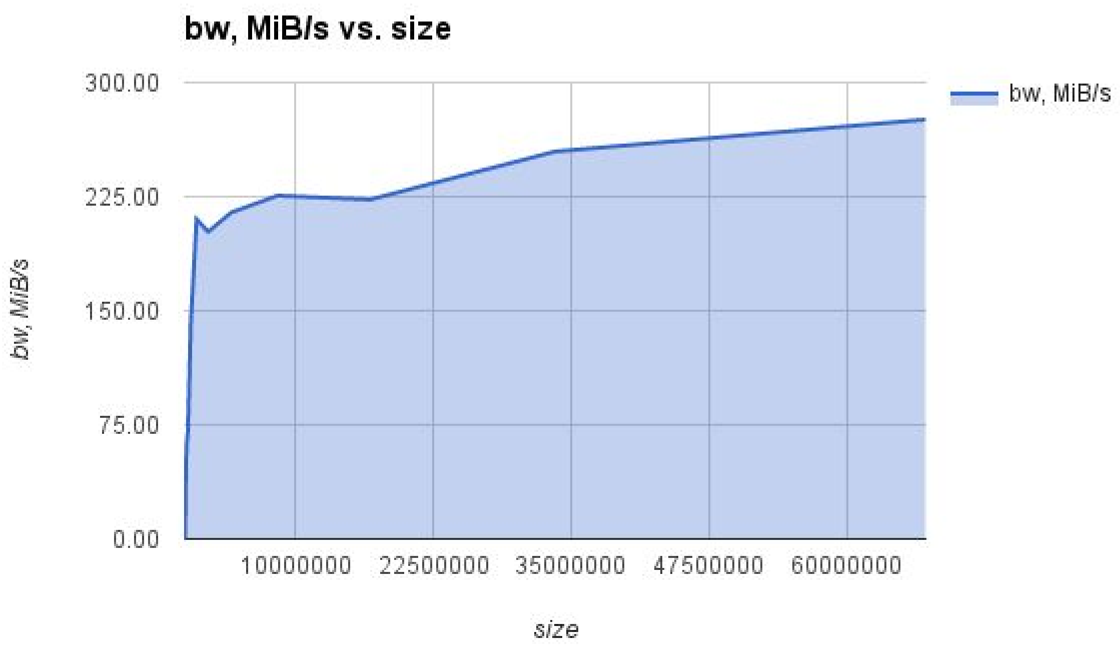


Fig.1. Bandwidth vs size for all message sizes.

Figure 2 below demonstrates reduced overhead for big message sizes.

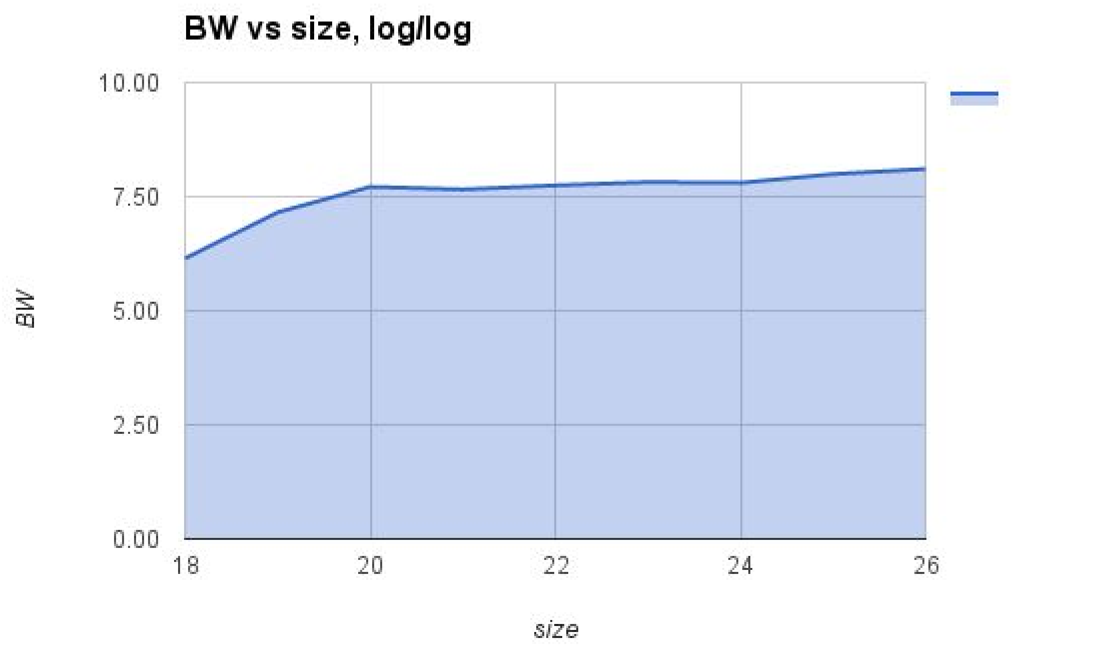


Fig.2. Bandwidth vs size in log/log scale.

For 64MiB messages, **bandwidth for one sided communication is 276 MiB/s**.

Evaluation

The achieved latencies are excellent, but the bandwidth is not. Almost certainly we have been unable to find memory in an RDMA address range and email with the Legion team has not resolved this issue.

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. However, this organization of logging information has proven to be useful.

Technical details

The logging infrastructure of Legion/Regent program uses Realm runtime system logging and I/O redirection provided by SLURM. Realm logging is set up so that all logs will be printed into stdout and SLURM I/O redirection puts everything into separate files for each node and task.

This has a positive side that all events are timed, including user logs, the negative side is that there can be too much logged information while user needed only few lines. It is possible to provide separate files for different logging streams, but that will detach SLURM node/task numbering and log files naming. This is so because SLURM cluster nodes are different from nodes of Regent (Regent nodes are tied to SLURM tasks) and it is not possible to properly name files with logging information without complicating system even further.

User can use grep and less utilities to get only user logs:

```
grep log_logging ~/.dflogs/JOB_ID/logs/node-N-task-M | less
```

Legion and Regent compile files into same directory as source files. Profiling options change compilation settings (for example, lowest log level is set to LEVEL_DEBUG instead of regular LEVEL_PRINT[4]) and all files should be recompiled with new settings. This can be achieved with “clean” make target and should be performed every time profiling setting is changed.

Running application with profiling information collection

To build and run your application with profiling information you should use the following command line:

```
PROFILING=1 <other args> make clean run
```

That line will build your application with profiling enabled and run it. The last line will be output of sbatch command with the SLURM job ID.

Displaying profiling information

After submitted SLURM job completes, you can run a handy make target to obtain profiling display:

```
PROF_JOB_ID=<SLURM job id> make process-profile
```

This command will create files prof_<SLURM job id>.html and prof_<SLURM job id>.svg in current directory. It will process files from ~/.dflogs/logs/node-*-task-*

You can then open HTML file with web browser and use hints for various parts of the profile.

The Legion profile is hierarchical - if operation needs some other operation to complete, the inner operation will be in profile just above the outer one.

[4] We believe the Legion team is addressing this, making it possible to dynamically change the logging levels.

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. The actor has a 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 $\frac{1}{M}\sum_{i=1}^{N} T_i$. An optimization calculation (brute force is acceptable) is performed to find a suitable distribution of the computations so that the actual runtime is close to the optimal one. The data flow program then schedules the transmission of data objects accordingly.

We implemented a simple program (the program code can be found here: https://github.com/SKA-ScienceDataProcessor/RC/blob/master/MS7/programs/05-Dynamic-scheduling/lpart.rg) which:

1. reads integer values T_i from 1d region of size `N`.
2. partitions this region into M disjoint subsets and schedule them in parallel to `M` actors which perform the task whose run time is T_i

Thus the optimization problem we have to solve is to partition the whole input region in such a way that a total computation time would be minimal. This is a well-known problem which is called "linear partitioning" problem in the dynamic programming parlance. We implemented S. Skiena's linear partitioning algorithm directly in Regent:

```
task solve(size : int, k : int, input : region(ispace(int1d), int))
where
  reads(input)
do
  var box = ispace(int2d, { x = size, y = k })
  var lin = ispace(int1d, size)

  var possible_cost_table = region(box, int)
  var partition_table       = region(box, int)
  var p = region(lin, int)

  p[0] = input[0];

  for i = 1, size do
    p[i] = p[i - 1] + input[i];
  end
  for i = 0, size do
    possible_cost_table[{x = i, y = 0}] = p[i]
    partition_table[{x = i, y = 0}] = 0
  end
  for j = 1, k do
    possible_cost_table[{x = 0, y = j}] = input[0]
    partition_table[{x = 0, y = j}] = 0
  end
  for i = 1, size do
    for j = 1, k do
      possible_cost_table[{x = i, y = j}] = 1000000000
      for r = 0, i do
       var cost = max(possible_cost_table[{x = r, y = j - 1}], p[i] - p[r])
       if (possible_cost_table[{x = i, y = j}] > cost) then
              possible_cost_table[{x = i, y = j}] = cost
              partition_table[{x = i, y = j}] = r
       end
      end
    end
  end
  ... ... ...
end
```

Now using partition table built we actually color input region and partition it:

```
task solve(size : int, k : int, input : region(ispace(int1d), int))
  ... ... ...
  -- Now color each partition
  var coloring = c.legion_domain_coloring_create()
  var lo : int64[1], hi : int64[1]
  var n = size - 1
  var ki = k - 1
  while ki > 0 do
    lo[0] = partition_table[{x = n, y = ki}] + 1
    hi[0] = n
    c.legion_domain_coloring_color_domain(
      coloring,
      ki,
      c.legion_domain_from_rect_1d {
       lo = c.legion_point_1d_t { x = lo },
       hi = c.legion_point_1d_t { x = hi }
      }
    )
    n = partition_table[{x = n, y = ki}]
    ki = ki - 1
  end
  lo[0] = 0
  hi[0] = n
  c.legion_domain_coloring_color_domain(
    coloring,
    0, -- ki == 0
    c.legion_domain_from_rect_1d {
      lo = c.legion_point_1d_t { x = lo },
      hi = c.legion_point_1d_t { x = hi }
    }
  )
  var parts = partition(disjoint, input, coloring)
  c.legion_domain_coloring_destroy(coloring)
  return parts
end
```

And finally main task uses the machinery developed:

```
task main()
  var size = 9
  var k = 3

  var isi = ispace(int1d, 9)
  var i = region(isi, int)
  i[0], i[1], i[2], i[3], i[4], i[5], i[6], i[7], i[8], i[9] = 4, 2, 1, 4, 5, 6, 7, 8, 9

  var parts = solve(9, 3, i)

  for part = 0, k do
    print_partition(parts[part])
  end
end
regentlib.start(main)
```

It should be mentioned that in this code returns input region partitions back to the main task which, in turn, launches a parallel set of `print_partition` tasks[5].

[5] When we tried to launch the parallel set of `print_partition` tasks directly from `solve` task we've seen intermittent segfaults at least on udp Legion backend even if we serialized them introducing fake (`writes (input)`) dependencies.

Binning irregular visibility data for parallelism

The data objects used by the data flow system shall be shown to support partitions for handling irregularly sampled collections with parallelism, demonstrated for a computation with similar side effects as those found in gridding of sampled visibilities.

Consider a calculation f(x) of an integer argument which is known to have some side-effects in such a way that it can't concurrently be applied to a pair of x_i and x_j, if that $|x_i - x_j| < D$ (some fixed constant, c.f. the width or support of the GCF). The side effect models the concurrent additions that may happen updating a single u,v grid-point when convolving nearby visibilities with the GCF.

Create a program that avoids concurrent computation when the side effect may take place forming suitable partitions of the data regions.

Approach

Let x_i belong to interval [0, N*D), N also some fixed integral constant significantly bigger than D. How to partition this set to allow f be executed in parallel on some subsets of points of these partitions? The idea is as follows:

1. First partition the whole set on two “chequered” subsets: we simply partition the domain [0,ND] into N parts [(j-1)D, jD], j=1...N and declare point being “black” if j is odd, “white” if j is even.
2. Now if we compute f for at most 1 point from each black tile - that is a subset [(m-1)D, mD] with odd m of the black subset - then we can compute f on all these points in parallel, and can repeat such steps until all black points are processed; then we can repeat the whole procedure for white tiles;
3. To accomplish this we perform further partitioning of black and white partitions. Each of these partitions is a union of tiles, and we have N/2 of such tiles for each (black and white) colour, some perhaps being empty. Now let’s take all black tiles and colour them additionally in N/2 different colours, such that all point within any given tile are colored with this tile colour. Then perform the same manipulation with white tiles.
4. Now map this to Regent/Legions Task concepts. Declare our f task as reading from and writing into the region it works on. If we launch f task now on all subpartitions (tiles) of black partition we see that Legion’s semantics say f will not encounter side effects. But f can be computed in parallel on the points of different tiles because they have different colours. After processing of all black tiles we repeat this procedure with white tiles. This way we accomplish precisely the behaviour described in the paragraph 2.

We note that an approach of dynamic load balancing as pursued above may be effective to get load balanced parallel execution only if (i) the number of threads available is small compared to the number of tiles and (ii) the distribution of work across the tiles is sufficiently varied to avoid outliers due to uneven distributions. If such outlier exist, a slower but safe parallel computation f’ - for example using atomic additions, or a private result grid which is later reduced - may be used only on a minimal amount of visibilities. We also note that the precise layout of the data can be optimized for cache locality.

Implementation

The implementation presented here works on unstructured index space. The code is here: https://github.com/SKA-ScienceDataProcessor/RC/blob/master/MS7/programs/06-Binning-Visibility-Data/binning.rg
Notable excerpts from it are:

```
local opaque = terralib.global(int[dim])

struct sparse {
    x : int
  , v : int
}

terra setdata(s : sparse)
  opaque[s.x] = s.v
end
```

We use some opaque function `setdata` working on `sparse` (tuple of coordinate x and value v) data structure which simply puts the value into the corresponding coordinate. It is a Terra function thus Regent doesn't reason about its side-effects. All relevant side effect information is conveyed to the corresponding Regent tasks via privileges annotations.

This is a utility task which does almost nothing except the very important thing: it is annotated as reading the region `r` which thus should be ready at the point in time when this task is executed:

```
task bw_barrier(r : region(ispace(ptr), sparse)
              , w : region(ispace(ptr), sparse)
              , s : rawstring)
where
    reads(r), writes(w) -- artificial dependency to wait for data be ready
do
  c.printf("Done with %s!\n", s)
end
```

This is the main task executing our opaque function on the region `r`. An important thing is that it is declared as both reading from and writing to region `r`, thus being forbidden to run on the points of this region in parallel:

```
-- perform the desired computation on a region
-- later called for partitions
task go(r : region(ispace(ptr), sparse))
where
    reads(r)
  , writes(r) -- artificial dependency
do
  for i in r.ispace do
       setdata(r[i])
  end
end
```

To lower the noise we use some metaprogramming. We generate 2 variants of partitioning code from the single template using quoting of fragments of Regent code. `make_coloring_task` is a metafunction which receives metafunction `fun`, applies it to quoted `r[i].x` expression, and splices into the returned terra function (see `[fun (rexpr r[i].x end)]` below):

```
function make_coloring_task(fun)
  local task gago(r : region(ispace(ptr), sparse))
          where
            reads(r)
          do
            var coloring = c.legion_coloring_create()
            for i in r.ispace do
              var color = [fun (rexpr r[i].x end)]
              c.legion_coloring_add_point(coloring, color, __raw(i))
            end
            var p = partition(disjoint, r, coloring)
            c.legion_coloring_destroy(coloring)
            return p
          end
  return gago
end

local function chequer(body)
  return rexpr (([body] / tile_size) % 2) end
end
```

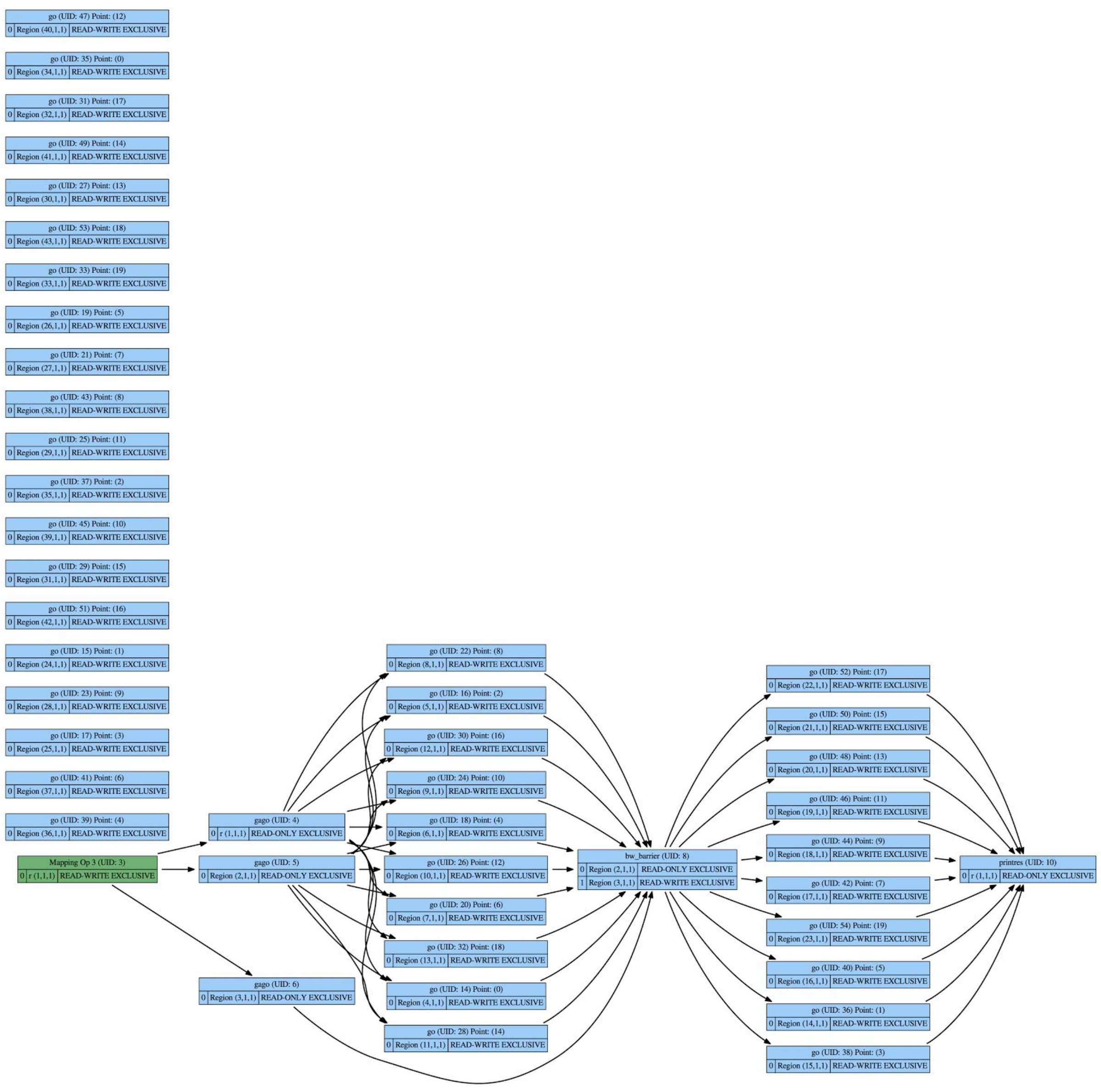
```
local function tile(body)
  return rexpr ([body] / tile_size) end
end

local chequer_task = make_coloring_task(chequer)
local tile_task = make_coloring_task(tile)
```

Finally we collect all the pieces together:

```
task top_level()
  var is = ispace(ptr, num_of_points)
  var r = region(is, sparse)
  ... ... ...

  var chequered = chequer_task(r)

  var p0 = tile_task(chequered[0])
  var p1 = tile_task(chequered[1])

  for n = 0, num_of_pieces do
    go(p0[n])
  end

  bw_barrier(chequered[0], chequered[1], "whites")

  for n = 0, num_of_pieces do
    go(p1[n])
  end

  ... ... ...
end
```

We first partition all the input on 2 parts using chequering partitioning, then partition each of these partitions further, then run our main task first on all partitions of the first of chequered partitions, wait for it’s completion and repeat this for the second of chequered partitions.

The event graph associated with this program can be generated by Legion’s “spy” utility. We include it below:



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.

Resolution

Sample programs for HDF5 integration with Legion are shown in the Legion repository, for example see: https://github.com/StanfordLegion/legion/test/hdf_attach/hdf_attach.cc . No working combination of branches could be found to reproduce this.

Resource Management

The data flow system shall handle resource management in its scheduling and serialize tasks for which resources cannot be obtained in parallel, 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.

C++ program main points

The whole program can be read here. Interesting parts are mapper's selection of task node and main entry point task.

The Main entry point spawns two tasks: SENDER1 and SENDER2. Each SENDER task is given an integer argument indicating amount of memory to allocate for receiver to process. Each SENDER$_i$ task creates region of specified size then spawns FILL$_i$ tasks (writes to region) and RECEIVER$_i$ tasks (reads region). FILL$_i$ and RECEIVER$_i$ tasks are scheduled sequentially due to their privileges. SENDER$_i$ and FILL$_i$ execute on i-th node (i=0,1), RECEIVER$_i$ execute on node 2. There are two receiver tasks for runtime to schedule on the same node, so they will not compete for anything given CPU and other resources available.

Results

Program, as run under stable branch of Legion, does not run to completion when amount of data to process in receiver's node exceeds allocated memory on the node. E.g., when each Legion node is given 512MB and command-line argument for main entry point indicates to each sender to send 257MB messages, the runtime crashes instead of sequence scheduling of processing requests. We expect this is due to unhandled signals.

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.

This was resolved without difficulties, see the code in github. The code does show the aforementioned problems with lack of Regent language-level support for partitions and colorings:https://github.com/SKA-ScienceDataProcessor/RC/blob/master/MS7/programs/09-Algorithmic-Expressiveness/choice/choice.rg#L28

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.

Our implementation can be found at https://github.com/SKA-ScienceDataProcessor/RC/tree/master/MS7/programs/10-Scalability-bi-sectional-bandwidth.

The idea of the program is as follows:

1. We query Legion runtime about all existing CPUs and randomly partition them into two equal groups, i.e., given the number of all CPUs available is N, we have 2 groups of CPUs with N/2 CPUs in each;
2. We allocate N/2 data buffers and make the first group of CPUs write to these buffers and the second group of CPUs read from these buffers -- the idea is that this will enforce the cluster to send the data between local memories with affinity to each CPU.

The fragment of code relevant to point 1 looks as follows:

```
local struct rep {
    datap : &c.legion_processor_t
  , procs : uint64
  , cpup  : &c.legion_processor_t
  , cpus  : uint64
}
```

```
terra cmp_procs(pa : &opaque, pb : &opaque)
  var pat : int = c.legion_processor_kind([&c.legion_processor_t](pa)[0])
  var pbt : int = c.legion_processor_kind([&c.legion_processor_t](pb)[0])
  return pat - pbt
end

terra report()
  var machine = c.legion_machine_create();

  var n = c.legion_machine_get_all_processors_size(machine)
  var procp = [&c.legion_processor_t](std.malloc(sizeof(c.legion_processor_t) * n))
  c.legion_machine_get_all_processors(machine, procp, n)
  c.qsort(procp, n, sizeof(c.legion_processor_t), cmp_procs);

  var off : uint64 = n
  for i = 0, n do
       if c.legion_processor_kind(procp[i]) == c.LOC_PROC then
       off = i
       break
       end
  end

  var ncpus : uint64 = 0
  for i = off, n do
       if c.legion_processor_kind(procp[i]) == c.LOC_PROC then ncpus = ncpus + 1
       else break
       end
  end

  c.legion_machine_destroy

  return rep{procp, n, procp + off, ncpus}
end
```

We query Legion runtime about all existing processors, allocate the vector of processors, read processor information into it, sort it regarding processor kind, traverse it until the first processor of `LOC_PROC` (latency optimized core, in other words “CPU”) kind is found, traverse the vector further till the last CPU is registered and return all this information info the corresponding structure `rep`, containing the pointer to the sorted processors vector, the number of processors, the pointer to the place where CPUs starts and the number of CPUs. Mention should be made that we never use anything other than the number of CPUs in our Regent code.

The main task looks as follows:

```
…
local chunk_size = 8*1024*1024;
…
local struct data {
  val : int8[chunk_size]
}
…
task main()
  var rep = report()
  std.printf("We have %lld processors in total, %lld cpus:\n", rep.procs, rep.cpus)
  for i = 0, rep.cpus do
       std.printf("\t%llx : %x\n", rep.cpup[i].id, c.legion_processor_kind(rep.cpup[i]))
  end
  c.free(rep.datap)

  var num_points = rep.cpus / 2
  var ps = ispace(int1d, num_points)
  var r = region(ps, data)

  var p_disjoint = partition(equal, r, ps)
  for i=0, number_of_tests do
       run_test(num_points, ps, r, p_disjoint)
  end

  std.puts("SUCCESS!")
end
```

We call previously described report task, then allocate the region, containing `num_points=rep.cpus/2` of `data` structures (`data` is a simple 8MB buffer) and partition it on `num_points` parts such that each part contains precisely 1 `data` structure (alternatively we tried to make each `write` task allocate its own region and return it to the calling task to be later consumed by `read` task, but it turned out Regent is unable to properly typecheck such a code). Then we perform `run_test` task several (`number_of_tests`) times. Each time the custom mapper (written in C++ and described further below) selects different randomly generated mapping of physical CPUs handling data transfers.

```
task run_test( num_points : uint64
              , ps : ispace(int1d, num_points)
              , r : region(ps, data)
              , p_disjoint : partition(disjoint, r, ps)
              )
where
  writes reads(r)
do
  __demand(__parallel)
  for i = 0, num_points do
       write(p_disjoint[i])
  end

  __demand(__parallel)
  for i = 0, num_points do
       read(p_disjoint[i])
  end
end
```

`run_test` task looks performs indexed launch of `write` and `read` tasks (they are straightforward and one can refer to the code in the repository if interested). To make sure Regent performed an indexed launch (the single launch routine is used for many parallel subtasks) we use `__demand(__parallel)` annotation, which makes Regent to fail at compile time if it can't perform indexed launch of annotated task.

Now we need to ensure that each and every `write` and `read` tasks were physically mapped to 2 separate groups of CPUs in such a way that the first group only does `writes`, the second only does `reads` and all are mapped to different CPUs -- one cpu for one operation. To accomplish for this we've written custom C++ mapper:

```
struct ReportMapper : public DefaultMapper {

  ReportMapper(Machine machine,
       Runtime *rt, Processor local) :
          DefaultMapper(rt->get_mapper_runtime(), machine, local)
       , local_mapped(false)
       , mapper_proc(Processor::NO_PROC) {}

  // We assert both tasks are mapped on the same proc
  bool local_mapped;
  Processor mapper_proc;
  std::vector<Processor> permutation;

  virtual void slice_task(const MapperContext ctx,
                          const Task& task,
                          const SliceTaskInput& input,
                          SliceTaskOutput& output)
  {
	assert(task.is_index_space);
	size_t middle;
	Machine::ProcessorQuery cpus(machine);
#define tname_is(tn) (strcmp(task.get_task_name(),tn)==0)
	if (!tname_is("write") && !tname_is("read")) goto def;
	cpus.only_kind(Processor::LOC_PROC);
	if(task.current_proc != cpus.first() || local_mapped){
		output.slices.push_back(TaskSlice(input.domain, task.current_proc, false,
	false));
		return;
	}
	middle = cpus.count()/2;
	local_mapped = true;

	if(tname_is("write")) {
		assert(mapper_proc == Processor::NO_PROC);
		mapper_proc = task.current_proc;
		printf("From Mapper! We have the following CPUs:\n");
		for (auto p : cpus) printf("\t%llx\n", p.id);
		permutation.resize(cpus.count());
		std::copy(cpus.begin(), cpus.end(), permutation.begin());
		std::random_device rd;
		std::shuffle(permutation.begin(), permutation.end(), std::mt19937(rd()));
		printf("Permuted to:\n");
		for (auto p : permutation) printf("\t%llx\n", p.id);

#define __MK_SLICES(__plus_off)				\
	assert(input.domain.get_dim() == 1); \
	auto rect = input.domain.get_rect<1>(); \
	assert(rect.dim_size(0) == middle);  \
	output.slices.resize(middle);        \
	size_t idx = 0;                      \
	for (LegionRuntime::Arrays::GenericPointInRectIterator<1> pir(rect); \
		pir; pir++, idx++)              \
	{                                    \
	Rect<1> slice(pir.p, pir.p);         \
	output.slices[idx] = TaskSlice(      \
		Domain::from_rect<1>(slice)  \
		, permutation[idx __plus_off]       \
		, false                         \
		, false);                       \
	}
	__MK_SLICES()
	}
	else if(tname_is("read")) {
		assert(mapper_proc == task.current_proc);
		__MK_SLICES(+middle)
#undef __MK_SLICES
		// Back to nothing
		mapper_proc = Processor::NO_PROC;
	}
	return;

	def:
	DefaultMapper::slice_task(ctx, task, input, output);
  } // slice_task

};
```

Our mapper inherits from default Legion mapper and overrides its `slice_task` method, which is called by Legion runtime when indexed task launch is occurred. According to our current understanding after indexed task is first mapped by parent task it is **always** remapped on the target processor even if `recurse` field of returned `TaskSlice` is set to `false`. In our early attempts we didn't respect this fact and had our programs hung (presumably) in a mapper.

Thus our mapper in the first stage does all the things described early above -- queries Legion runtime about all CPUs, shuffles them randomly, partition on two parts, slices the domain making a slice per each point in the domain, and sends all `write` tasks to different CPUs in the first partition, and `read` tasks in the second partition. In the second stage, on the target processor the mapper simply leave things unchanged.

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.)

Resolution

This is presently impossible. If one process fails which uses gasnet the application fails entirely.

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).

We refrained from implementing this because the whole actively developed project addresses this completely. This is Los Alamos FleCSI project. It covers much more, but one of essential part of it is to provide a client a protocol and API which allows to call Legion runtime from inside the MPI routine and vice versa.

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.

Calling FFTW

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

An implementation resides in the following file:
https://github.com/SKA-ScienceDataProcessor/RC/blob/master/MS7/programs/13-Foreign-Function-Interfaces/fftw_ffi.rg

This implementations omits HDF5 mapping, see above.

C foreign function calls in Regent

Some further details should be mentioned. C language invocation in Terra (and thus of Regent) is basic and can't handle any #define's except the simplest ones, when literal constants are defined, hence more complex macros should be converted into C-functions. The example from the code above looks thus:

```
local c = terralib.includecstring [[
……...
#include <fftw3.h>

/* Terra can't deal with nontrivial macros */
int __FWD(){return FFTW_FORWARD;}
int __EST(){return FFTW_ESTIMATE;}
]]
```

Another possibly undocumented fact is the existence of Regent built-in facilities to access high-level Regent and low-level Legion code. There exists a set of pure C-wrappers to the main Legion C++ functionality and a set of Regent constructs, among the accessible functions are regent intrinsic functions `__runtime(), __context(), __physical(r)` and `__raw(o)`. The first two return current runtime and context handles which are are used by Legion ubiquitously, the third returns a pointer to physical region handle corresponding to a given region and the last is applicable to several different types of Regent objects -- `ispace` (returns a value of `legion_index_space_t type` applied to an unstructured index space and a `legion_ptr_t` value, for elements of structured index space -- underlying integral value), `region` (delivers value of `legion_logical_region_t` type), `partition` (delivers value of `legion_logical_partition_t` type), `cross_product` (delivers value of `legion_terra_index_cross_product_t`), and so called “bounded types”.

We can see how this machinery can be used to get a raw data pointer on 1-dimensional region data.

The corresponding Terra function is:

```
terra get1dptr(
    regs : &c.legion_physical_region_t
  , ctx : c.legion_context_t
  , runtime : c.legion_runtime_t
  )
  var r = c.legion_physical_region_get_logical_region(regs[0])
  var is = r.index_space
  var d = c.legion_index_space_get_domain(runtime, ctx, is)
  var rect = c.legion_domain_get_rect_1d(d)
  var acc = c.legion_physical_region_get_accessor_generic(regs[0])

  var subrect : c.legion_rect_1d_t
  var offsets : c.legion_byte_offset_t[1]
  return [&c.fftw_complex](c.legion_accessor_generic_raw_rect_ptr_1d(
                    acc, rect, &subrect, &(offsets[0])))
end
```

and client Regent code looks like this:

```
task gen_data(
   N : int
 , d: region(ispace(int1d), double)
 )
where
  writes(d)
do
  var data = get1dptr(__physical(d), __context(), __runtime())
  var Nd : double = N
  var theta : double
  for n = 0,N do
       var nd : double = n
       theta = nd / Nd * c.M_PI
       data[n][c.REAL] = c.cos(10.0 * theta) + 0.5 * c.cos(15.0 * theta)
       data[n][c.IMAG] = c.sin(10.0 * theta) + 0.5 * c.sin(15.0 * theta)
  end
end
```

Also mention should be made that in Terra code above we used constructions which are valid both in Terra and Regent (once more: while Regent tries to mimic Terra as far as possible there exist some differences), e.g. we de-reference `regs` pointer with `[0]` index which is valid in both Terra and Regent while dereferencing with `@` sign is valid in Terra only.

OpenMP kernel in Region program

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

This case is not different from calling C++ code. However, embedding such code in Terra may lead to difficulties compiling, but linking with a compiled C++ library should not pose problems.

MPI from data flow

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

An implementation resides in the following file:
https://github.com/SKA-ScienceDataProcessor/RC/blob/master/MS7/programs/13-Foreign-Function-Interfaces/dot_mpi.rg

This implementations as of now lacks HDF5 part since it is not implemented yet, nor do we try to use a real region for the data (when HDF5 is done we'll use `get1dptr` function already devised in FFTW program). The implementation uses `MPI_Gather` to accomplish the task.

OpenACC

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

The same considerations as for OpenMP are valid here. OpenACC is implemented by gcc 5.2.0. Provided the runtime's saveobj function is fixed, it might be possible to create GPU binaries with the Regent program during compilation.[6]

Halide

A regent program can call a Halide function effectively.

As for OpenMP.

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

The program as a whole can be seen here. It consists of five time measurements, one for establishing baseline timing and others for timings of optimized loops.

Results

The following table contains results of executing program on a four cluster machines, each machine is given 11 pthread threads:

Optimization	Runtime, ms
Baseline	579.4

[6] This function has been improved, but is not currently working.

SIMD	116.3
SPMD	78.6
Parallel	97.2
GPU	120.5

The GPU result is out of line because Regent program compiles its code in the runtime on every cluster node and produces CUDA kernels to be compiled with CUDA compiler. This compiler cannot be executed on the cluster node.

Baseline

By default, Regent tries to optimize loops to the maximum, applying vectorization if possible. So to test baseline performance we have to explicitly forbid vectorization optimization.
In other cases below demanding optimizations basically just reports whether optimization is not possible - due to implicit data dependencies, for example.

SMP vector instructions

The loops inside Regent code can be marked as demanding vectorization:

```
__demand(__vectorize)
for i in is
      ...operation on region[i]
```

SMP threads

It might appear that Regent can allow to use SPMD (OpenMP, for example) optimization for loops, but it is not the case. The SPMD demand is for distributed parallelism and will be explained in the next section.

Legion code can use OpenMP pragmas for loops with a caveat. Code must ensure regions are ready before the loops. Otherwise there will be MPI messaging inside OpenMP parallel code which will slow program down (best case) or cause unpredictable behaviour.

Distributed parallelism

To utilize distributed parallelism, we should partition our data and demand parallelization. The actual code is here. It requires a bit of extra code to partition data and we provide all the code in full below.
We need a task that performs our baseline loop on parts of regions:

```
__demand(__cuda(__unroll(10))) -- hint for GPU execution, explained below.
task subtask(ra : region(ispace(int1d), float),
        rb : region(ispace(int1d), float),
        rc : region(ispace(int1d), float))
where writes(rc), reads(ra, rb) do
    __demand(__vectorize)
    for i in ra do
        rc[i] = ra[i]*rb[i]
    end
    return 0
end
```

In our main task we partition our regions into equally-sized partitions:

```
var np = 4 -- number of subregions for data parallelism
var colors = ispace(int1d, 4)
var pa = partition(equal, ra, colors)
var pb = partition(equal, rb, colors)
```

```
    var pc = partition(equal, rc, colors)
```

The partitions provide view into same data as original regions, just split into four (np) parts for parallelization convenience.

The loop is below:

```
    do
        var _ = 0
        __demand(__parallel)
        for i = 0, np do
            _ += subtask(pa[i], pb[i], pc[i])
        end
        wait_for(_)
    end
```

There is an extended version of distributed parallelism, called SPMD. It operates over parallel loops by creating barriers inside them and generally leads to faster code (yet experimental).

GPU parallelism

Regent code can ask for execution on GPUs using `__demand(__cuda)` request for specific task. The current runtime makes this difficult to exploit on the Wilkes cluster, as it will compile Cuda code on the fly and Cuda compilers are not installed on compute nodes[7].

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 in 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 in the 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

The code resides in https://github.com/SKA-ScienceDataProcessor/RC/blob/master/MS7/programs/15-Distributed-Partial-Data-Transposition/trans.rg and is a straightforward implementation of matrix transposition by the set of parallel tasks. This is achieved by using the most basic and fundamental Legion/Regent construct: partitioning.

The transpose task itself is absolutely straightforward:

```
task trans( i: region(ispace(int2d), double)
      , o: region(ispace(int2d), double)
      )
where
  reads(i),
  writes(o)
do
  for rc in i.ispace do
```

[7] Numerous branches of Regent and Legion were tried without success.

```
        var co = int2d{x = rc.y, y = rc.x}
        o[co] = i[rc]
  end
end
```

Now we want to partition input region into k “stripes”. Built-in Regent partitioning facilities for structured index spaced regions are unable to accomplish this. Even if we try to equally partition the region with the stripe-shaped hint Region tries to tile this region. Thus we need to use low-level Legion binding:

```
-- true - "vertical" partitioning, false - "horizontal" partitioning
task structured_partition(r : region(ispace(int2d), double), dir : bool, rx : int64, ry :
int64, pieces : int64)
  var qr : c.lldiv_t = c.lldiv(rx, pieces) -- need rsize >= pieces
  var small_size = qr.quot
  var big_size = small_size + 1
  var big_pieces = qr.rem

  var d0 : int, d1 : int
  if dir
  then d0, d1 = 0, 1
  else d0, d1 = 1, 0
  end

  var coloring = c.legion_domain_coloring_create()

  var curr_lo : int64[2], curr_hi : int64[2]
  curr_lo[d1] = 0
  curr_hi[d1] = ry - 1

  curr_lo[d0] = 0
  for cb = 0, big_pieces do
        curr_hi[d0] = curr_lo[d0] + big_size - 1
        c.legion_domain_coloring_color_domain(
               coloring,
               cb,
               c.legion_domain_from_rect_2d {
                      lo = c.legion_point_2d_t { x = curr_lo },
                      hi = c.legion_point_2d_t { x = curr_hi }
               }
        )
        curr_lo[d0] = curr_lo[d0] + big_size
  end
  for cs = big_pieces, pieces do
        curr_hi[d0] = curr_lo[d0] + small_size - 1
        c.legion_domain_coloring_color_domain(
               coloring,
               cs,
               c.legion_domain_from_rect_2d {
                      lo = c.legion_point_2d_t { x = curr_lo },
                      hi = c.legion_point_2d_t { x = curr_hi }
               }
        )
        curr_lo[d0] = curr_lo[d0] + small_size
  end

  var p = partition(disjoint, r, coloring)

  c.legion_domain_coloring_destroy(coloring)

  return p
end
```

This function is generic and takes one extra `dir` argument, which tells how to “stripe” an input region -- along the first or second dimension. Important thing to notice: we don’t swap input dimensions depending on this `dir` argument, thus the client code should swap instead (more on this below).

The client task is parametrized by numbers of rows, columns and numbers of subtasks ("actors") required to work in parallel. Regarding our terminology this number of "actors" is a number of "stripes" we partition our regions to:

```
task test(r: int, c : int, k : int)
  var isi = ispace(int2d, { x = r, y = c })
  var iso = ispace(int2d, { x = c, y = r })

  var i = region(isi, double)
  var o = region(iso, double)

  var pi = structured_partition(i, true,  r, c, k)

  __demand(__parallel)
  for n =0, k do
       init(pi[n])
  end

  var po = structured_partition(o, false, r, c, k)

  __demand(__parallel)
  for n =0, k do
       trans(pi[n], po[n])
  end
end
```

`structured_partition` function is applied to input and output regions in different directions: the first is striped vertically, the second --- horizontally. Please, pay attention to how row and column numbers are the same in both cases, this is because for output region they are swapped two times -- since the output region has the shape of input region transposed, the rows and columns are swapped the first time, since our `structured_partition` function doesn't perform required swapping for horizontal striping we perform swapping another time thus returning to the original order.

We profiled this program both in SMP and distributed configuration on Wilkes cluster. We transposed 8192x8192 matrix of double precision floating numbers with the whole `trans` task partitioned on 16 parts:

```
task main()
  test(8192, 8192, 16)
end
```

The SMP configuration uses 8 CPU cores (and 8 CPU configured) on a single Wilkes node and the task was automatically parallelized by Regent. The distributed configuration used 16 separate Wilkes nodes with 1 CPU configured on each node. All tasks also are automatically parallelized by Regent.

Profiles, generated by legion_profile utility, with a legend obtained by hovering over the boxes, can be seen here: http://awson.github.io.

Interesting observations are as follows:

1. In distributed configuration all work distribution is completely uniform, each CPU makes a single call to `trans`, duration is also very stable. The whole 512MB of data movement are evenly distributed between 16 nodes. Each node allocates both it's stripe of the original matrix and the transposed stripe of transposed matrix and transposes it's own portion of data. Each call thus transfers 32MB of data and this takes ~46 ms delivering ~5.6 gigabit/s bandwidth per CPU, for all nodes combined we have approximate bandwidth ~39 gigabit/s, see for example job 3055218 (all data are transferred for 105

ms -- between 827 and 932 millisecond). I count the total duration from the beginning of the most early started `trans` task to the end of the most late finished `trans` task. Surprisingly, Legion optimizes the layout of the transpose and avoids distributed data transfers. A further study should probably explore this by adjusting parameters.

2. In SMP configuration all configured cores are used but in different runs they can be loaded in different ways from uniform load -- see job 3055624, where each CPU calls `trans` twice, to a pretty non-uniform load -- see job 3055626 where the number of calls to `trans` varies for different CPUs from 1 to 4. And thus total bandwidth (for all 8 CPUs) varies from ~24.8 gigabit/s (all data are transferred for 165 ms -- between 324 and 489 millisecond for job 3055624) to only ~16.9 gigabit/second (all data are transferred for 242 ms -- between 238 and 480 millisecond for job 3055626).
3. Thus we see that distributed configuration could be even slightly better than SMP one. For SMP we can see sometimes tasks scheduled slightly erratically and SMP can even lose to a distributed configuration (16.9 on 8 cores is worse than 39 on 16 cores).