Radio Cortex

SKA SDP DSL Milestone Architecture & Design

About Milestone Design Documentation

During each milestone, the features of DNA might be extended and previous milestones will be updated, as will be their documentation in this document. For documentation accompanying a particular delivered milestone, it is best to refer to the PDF documents that will accompany each milestone, on the Github wikis, see the SDP DSL software engineering practices document.

Runtime Environment and Programming Interfaces

Programming Environment

All programs will be written using the Declarative Numerical Analysis (DNA) package, which is the key infrastructure deliverable of the project. DNA is built on top of Cloud Haskell.

Source code organization

The RC repository will get a subdirectory for each milestone. On the machine used for testing there will be a clone of the git repository for easy dispatch of bug fixes. There will be a Makefile to compile and install the code. Make depends on the availability of the DNA libraries which must be compiled first. Scripts to fully run the demonstrations must be available (including e.g. a script to populate the file with floating point numbers).

Runtime Environment

All milestones will be run on Cambridge HPCS clusters. Of particular importance is that jobs will be started with SLURM, as the resources available to the DNA programs will be utilized dynamically.

First milestone: data driven distributed dot product

During the first milestone the goal is to create a working data driven programming environment implement a small but representative program. Equally or more important than working code is a good understanding of alternative approaches and addressing future requirements with this style of programming. We also want to understand the available tools and the possibilities to create fruitful collaborations.

This section contains design elements for *software* for the first quarterly milestone. This includes:

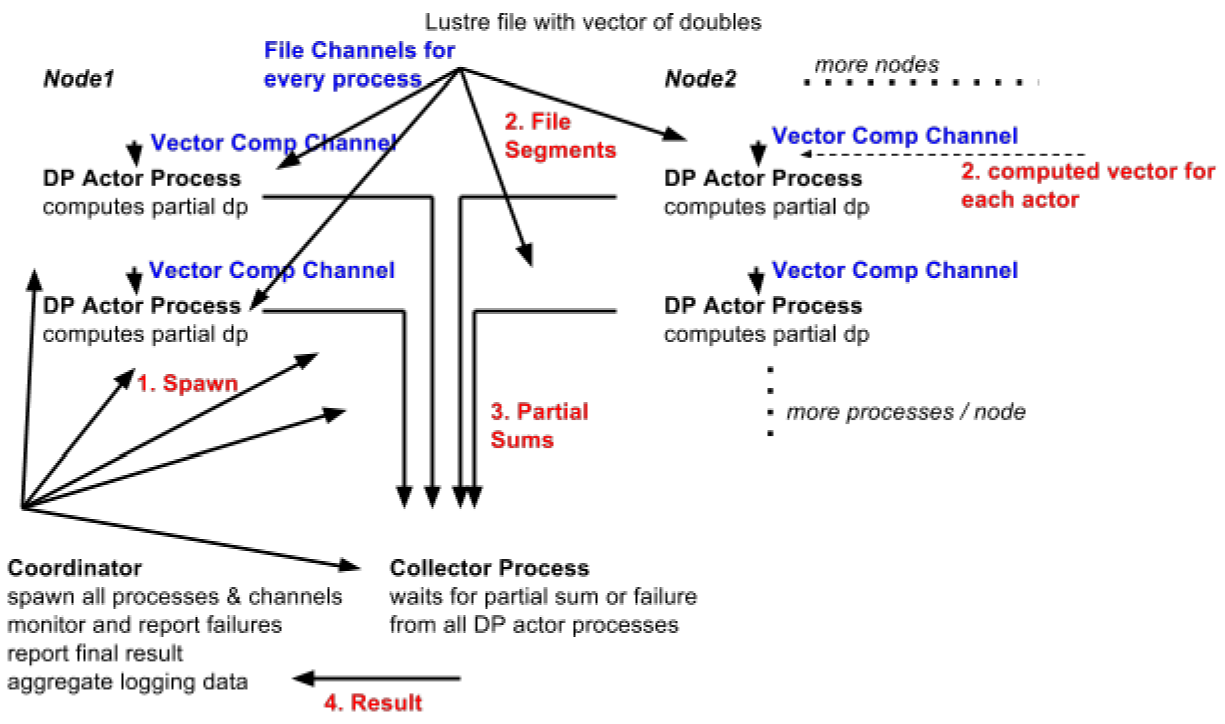
1. the distributed dot product
2. gridding and FFT infrastructure
3. measurement set high availability handling

Many components such as actors, channels, profiling and testing infrastructure are discussed in the DNA design document as they are of a generic nature.

Dot Product Channel and Actor Program

Specification of the program

The distributed dot product (DDP) program computes a dot product on a cluster using N+2 processes. The N+2 processes in the distributed computation play one of 3 roles: there is a single coordinator, a single collector and there are N compute processes. Figure 1 summarizes the functionality of the program. The bold arrows label messages and the content is described in red. Messages expected to appear in a particular order are numbered. The bold faced text elements represent processes computing with the algorithm summarized below. Italic text provides further comments.



Each compute process spawns a file channel and reads a vector of double precision numbers from an extent - the extent depending on the process - in a single shared file (SSF) into memory. The single shared file may be assumed to be a file in Lustre created and populated by an external program. The number of double precision floating point numbers in the file can be derived from its size.

Each compute process also generates a second vector, in a background process implemented as a channel, through computation and places it into a memory buffer. This calculation may use some random number generator or other function to create floating point numbers.

When the two sub-vectors for a particular compute process are both available, their dot product on that node (which is strictly speaking the dot product of a projection of the vector to a subspace determined by which indices in the vector will be read by that process) will be computed and when the local dot product has been computed, it will be propagated to a collector. The compute process on this node may now exit.

The collector adds up the contributions from and *when all compute processes* have been accounted for either by contributing their computation, or by failing or by both, the collector will send the result to the coordinator and exit.

If a compute process fails its failure will be reported ***by the coordinator*** node to the collector node and its contribution will not be added to the accumulation of the global dot product. If coordinator also monitors the collector - if it fails the program exits. The coordinator waits for the sum (or failure) of the collector, prints it and exits.

Further simplifications:

- considerations of overflow may be avoided
- a single computing thread per process is sufficient
- all data may be assumed to fit into memory

How the program uses DNA interfaces

We want to achieve the following key goals to make sure our first deliverable works towards the Descriptive Numerical Analysis framework. To achieve this the following constraints will be placed on the design and implementation

The **CAD** will be used and is a list of pairs of node numbers 1….N+2 and node addresses. This list is obtained from SLURM upon job start. The coordinating process takes the CAD and takes the size of the input file. It then schedules a collector and computation processes on the cores of the available nodes.

The dot product is written as a simple function, that looks as much like the local Haskell function to compute the dot product.

Conceptually the two vectors are a global distributed array, but we only need extents in the array for each process.

To read an extent in a file, a ***file reading channel*** thread or process is created that is instructed what file to open, at what offset and how many floating point numbers it is to read from that file into an array. It is a standard pattern that this needs to happen on each compute node, with the extent shifting further into the file as the node number increases, hence the concept of a ***swarm of channels*** will be introduced and used to read the data.

The computed vectors similarly will use ***a swarm of compute vector channels*** that each deliver a memory buffer, which again defines a global array (by giving the elements affiliated with each process).

A specially crafted ***dot product actor*** represents the computation of the dot product on each node. When both the file and buffer channels have sent a message the dot product actor that the data is ready the actor will compute the dot product on the node using standard Haskell and when finished send the result to the collector node, upon which the processes on the compute node exit. The operation **get** will be used by the dot product operator to obtain the data from the file and memory channels when the ready messages have been received.

The collector actor is another specially crafted actor, which has a ***network channel*** (a simple Cloud Haskell connection) from each dot product actor , and one from the coordinating process. Each dot product actor uses the **put** operation on the channel to send partial sums and the HA actor running on the coordinating node uses the put method on the channel to indicate a failure. The collector actor will maintain a ***counter*** of received messages to track when its work is complete.

Other milestone functionality

As an initial profiling tool, the channels and actors each report all times when they start and stop execution. All actors and channels, except for the aggregation actor and high availability channel, appear to run only one time. We will extract this logged information and visualize it using a spreadsheet.

An interface is made to the coordinator to kill a node, to simulate high availability events.   A failing node is reported in a log (possibly not in the profiling log).

An architecture for testing DNA channel actor based programs will be designed and initial elements will be included in the program.

Syntactic issues

A key challenge is to formulate the dot product computation in a mathematically attractive manner using a ***DNA function*** that basically indicates not more than

1. what formula to use on the two distributed arrays,
2. how the segments of the distributed arrays are obtained with get operations from the file and memory channels
3. that a separate accumulation actor collects partial sums
4. that the accumulation actor writes its result to a file
5. how the two types channels connect to either a file or a compute actor thread to obtain their data
6. that the high availability model is, in this case, is a fail-out model

We think the following structure of the program text clearly expresses intent and is remarkably similar to the Dague language syntax.

```
computeProcess n len file collector = do
       fileChannelPid <- DNA.Channel.File (Offset n len) (Count n len) file
       computeChannelPid <- ComputeChannel (Offset n count)
       v1 <- expect fileChannelPid ptr
       v2 <- expect computChannelPid ptr
       let dotp = sum zipwith ‘.’ v1 v2
       send collector PartialSum dotp
       return getSelfPid
```

```
collector_process computePids = do   XXX this is more like: do …. until empty
list
       receiveWait
              match: (Failure pid) -> strip node from list
              match: (Partialsum pid s) -> strip node and add s to sum
       where
              stripnode pid list = filter (/= pid) list
       send coordinator $ Result sum

coordinator file cad_file = do
       cad <- interpretCad cad_file
       (compPids, collectorPid) <- scheduleProcs cad_file file
       monitorFailures [collectorPid] Fatal
       monitorFailures compPids Notify collectorPid
       (Result dotp) <- expect
```

Components of the program

At a high level the program decomposes into what runs on each of the nodes (or within each of the processes).

Basic functionality proceeds through the following components executing the following actions:

- **N compute processes**. Upon instruction from the coordinator, these nodes run
    - a thread implementing a channel to do I/O reading double precision floats from a file into an array
    - a thread implementing an actor connected to a memory buffer channel to compute an array of floating point numbers
    - the file reader and memory buffer signal the dot product actor when data is ready upon which a per node dot product computation takes place
    - the last action of the dot product actor before exiting is to send its result to the accumulator node and to let the coordinator know it is exiting for reason of completion (not for reason of failure). Note that this tears down two distributed channels: one between the compute node and the accumulator, and the "built in channel" between the compute node and coordinator that monitors high availablity.
- **collector**
    - this node waits for messages from compute nodes or from the coordinator.
    - If a message arrives from a compute node, it contains a partial dot product which is added to the total, and the node from which it arrives is added to the list of nodes that completed the work
    - If a message arrives from the coordinator node it contains the node number of a failed node and this is added to the list of nodes that completed the work
    - when all compute nodes have completed (through failure or finishing the computation), the accumulator prints the result and the list of failed nodes and lets the coordinator node know that it is exiting. It tears down its channel with the coordinator as it exits

- **coordinator**
    - reads the cluster configuration
    - computes reasonable buffer sizes to hold the arrays from the file size and the number of compute nodes
    - it starts the accumulator node with its actor
    - this coordinator forms a swarm of actors and channels and distributing the swarm means the threads on each compute node are set up
    - it sets up the connections (channels) between the compute nodes and the accumulator, upon which the read channel and compute actor can start
    - it can propagate kill messages
    - it waits for all nodes to exit
    - it exits itself

Use case perspective

Next we discuss two refined use cases: failures and profiling

- **correctness**
    - the program is run at various scales with known input functions and the correctness is inspected.
- **failure handling**
    - through a slurm command, shell command, or other control command a *compute process, its file channel or its compute channel* is killed. (Note: this version of the program does not handle “hangs” where one of the threads does not make progress.
    - the monitoring process in the coordinator observes the failure and removes the node from the processes that it is watching
    - the coordinator informs the accumulator of the node number that has failed. The accumulator checks if this completes the total distributed task by working with the list of completed nodes, failed nodes and the total number of compute nodes
- **profiling**
    - The compute node actors and channels in this program appear to run only one time, but the start of some of them may not be the same as the start time of the program (e.g. the local dot product calculation starts after the memory buffers are ready). They record in nano second precision the identities of the channels and actors, their start times and finish times and write these to a log file upon exit. The number of cycles executed, bytes moved and other useful information should be added to the log (don’t make it too complicated initially).
    - The accumulator node and coordinator node will be waiting for most of the time, but also record names of actors, start and finish times.
    - Upon completion the data is gathered into a spreadsheet for visualization

Binaries and data file organization

We implemented

Gridding and FFT High Level Design

Abstract

The second milestone of the DSL project will build a gridding program using a data flow manager. The DSL compiler will target multiple architectures, but one programming language which could be Cloud Haskell, Legion or another one, to be determined before 9/1/2014. An initial model of scaling and performance as a function of cluster, data set and channel parameters will be included.

Introduction

The gridding and FFT milestone in the DSL project - labeled MS2 - is a program that reads visibility data, applies a default (not gain adjusted) gridding convolution function and performs a Fourier transform, to get a dirty image which shall be stored in a file.

Key aspects of the program that go beyond the previous milestone is that we will target multiple architectures (GPU and SMP), multiple measurement data sets will be processed from each node, including initial infrastructure to track an equal load distribution, and a scheduler arranging colocation of compatible GCF and uv data.

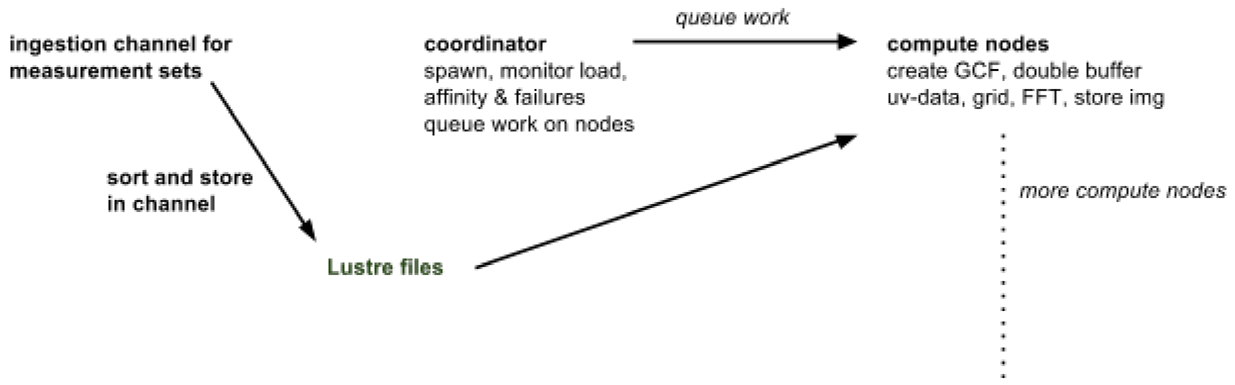

ingestion channel for measurement sets
sort and store in channel
Lustre files
coordinator
spawn, monitor load, affinity & failures
queue work on nodes
queue work
compute nodes
create GCF, double buffer uv-data, grid, FFT, store img
more compute nodes

Figure 2: Gridding work flow

Moreover, we will introduce a performance model using (i) cluster / system architecture, (ii) input data across a set of nodes and (iii) kernel performance as a function of buffer sizes (obtained from profiles) and (iv) the possibility to measure performance differences arising from layout reorganization. The purpose of the model in this milestone is to see if adjustments to its parameters lead to clearly observable results, with consistency beteen model and profile based data.

We will also extend our dot product software to run on a GPU and use input data vectors that are not sequentially stored.

This document summarizes the architecture and high level design of the gridding and FFT implementation.

Requirements

The program shares the requirements applicable to the entire DSL project and some additional ones which we summarize here.

1. An I/O channel will load visibility or uv-data from a storage system into memory. Its implementation will be separate from the compute environment and it can take certain parameters such as buffer sizes and handle double buffering (for overlapping computation and I/O)
2. A compute channel will compute an analytical gridding convolution function (GCF). For the purpose of this milestone, the GCF will use a w-projection and prolate spheroidal AAF.
3. A data driven approach will be introduced between the CPU and GPU resident computations and the data channels.
4. A multithreaded convolution kernel running on a CPU or GPU will be started when the channels have produced their data.
5. The gridded data will be subjected to a Fourier transform, utilizing a second kernel.
6. We will combine the execution of gridding and Fourier transform with a full execution model that schedules ongoing execution of the algorithm on multiple data sets across a cluster of nodes.
7. A performance model is introduced and studied based on (i) cluster / system architecture, (ii) input data and (iii) kernel performance as obtained from profiles.
8. Profile data will be obtained at the level of the data driven framework and at the level of the kernels.

Specification

Channels for uv-data

Ingest of uv-data is done through an artificial uv-data set generator, which we will borrow from ASKAPsoft. (*We are not aware where to find one.*) If required, the DNA channel will sort the data (e.g. to keep data belonging to different w-planes together) or deliver a subset of uv-data before writing it into one or more Lustre files.

For the computation we probably only need a few tables specifying the field/pointing direction and a table of antenna positions and u,v,w with complex visibility data. In future milestones we will explore what data layout leads to appropriate I/O. The I/O demands on a single node are very high and it is not impossible that common formats such as UVFITS, Casacore are too complex to achieve the required rates.

We expect to create many measurement sets to observe differences in computation and I/O times and prepare for adapting the scheduling of the gridding tasks.

DNA FileChannels will be used to read and use double buffering for the uv-data in the compute nodes, so that I/O can progress while gridding is in process.

We will specify exactly what the channel needs to produce in conjunction with the detailed design of the algorithm.

GCF

We will use an analytical GCF computed using w-projection as in Cornwell Voronkov Humphrey's paper. The w-term G and a prolate spheroidal function as AAF will be combined to obtain a GCF, as explained following formula (5) of the above paper.

We understand that for a simple function such as a prolate spheroid we need support of 7 pixels, but if there is a significant w-term (e.g. not a simple east-west array) we would need to determine an appropriate support size of the GCF which depends on a number of parameters. In practice this ranges from 7 (w==0) up to many hundreds of pixels. We generate the convolution function on a grid of say 1024x1024 pixels then run a “support searcher” and then cut out the appropriate area. *We assume this is done by knowing the size of the GCF and cutting off at a certain %.*

When the GCF is used, we will send the subset representing the correct w-value to the convolution task.

Gridding operation

For the implementation on CPU and GPU’s we will follow the algorithms indicated in Romein’s ISC 2012 paper, sections 2 and 4.

Section 6 indicates indicates that the right strategy for GPU and CPU implementations and data access is different. This offers the opportunity to adapt the data channels and/or sort the file data differently.

Storing the result.

We will write a FITS format image, using a C foreign function interface to cfitsio.

Alternatives are to write a CASA Image.

Implementation

Consistent with the goals of the project we will focus on:

1. A clear separation between architecture dependent code for the kernels and the application level code.
2. An approach that enables assessing performance based on (profile driven) models and adapting the data flow approach based on the results of the models.
3. Code for the application that is as simply as can be reasonably expected.

High availability and scheduling proposal for processing measurement sets

A simple High Availability model for the management of measurement sets is to provide two capabilities:

- If a node fails, data that is stored and not available over a distributed storage system is not processed. This is similar to the failout model we used for the dot product.
- If a node fails, the ingest process avoids the failed node and delivers new data to other nodes, taking into account reasonable locality requirements wrt to the required GCF's.

The second requirement is more complex to implement as a cluster membership module is needed to track the failure and arrival of nodes.