Declarative Numerical Analysis

Peter J. Braam

Motivation

This document describes elements of data flow computation, in which actors are triggered by the arrival of data on input channels and the availability of buffer space on output channels.

A dataflow description originates from a data flow directed graph. Often the data flow has an abstract form involving families of graphs, and can be turned into a physical data flow graph by scheduling computations using variable numbers of compute elements and varying buffer sizes. A central abstraction in this scheduling is a cluster architecture descriptor (CAD).

We will describe how cloud Haskell can implement DNA programs using a few simple type definitions and functions and how these can be run and scheduled using the SLURM job scheduler.

By formulating DNA as a DSL we gain the opportunity to create the code for the program dynamically, and insert optimization steps and target languages. A DSL compiler has already been created but no optimization steps are currently performed.

As the SDP DSL project matures, we expect to describe other aspects of data driven programming as it benefits the imgaging algorithm.

Actor and channel computation

Concepts

A channel is a unit of executing software that exists primarily to move data from sources to sinks and enforce a data flow driven execution of programs. The data flows:

- from one or more **sources**
- to one or more **sinks**

A simple example of this data movement is to read different parts of a file and deposit data that is read into different memory regions, which could be RAM or accelerator (e.g. GPU) memory.

Data in the sink of a channel is to be consumed by **actors**: in this case the channel is called an **input channel** of the actor. Actors can also produce data and place it into the source of a channel in which case the channel is called an **output channel** of the actor.

Channels signal the actors to which they connect when their sinks or sources are ready to provide or absorb data. The actor can consume these ready messages from the channels asynchronously or use blocking operations to **get** data from input channels and **put** data on output channels.

A channel may transform data using

- **layout invariants for its source and sink** that need to be respected by the channel
- a **filter function** attached to the channel

As an example, a channel may be told to read an array of floating point numbers from a file where they are stored in big endian format, while placing the results in memory in little endian format: this is a layout descriptor. The filters enforce the source and sink layout invariants, through transformation of the data. Filters may act on data where the layout descriptor is a "best effort", for example, a filter may try to find clusters of nearby data points in its source and place those close together in its sink, performing some kind of best effort local sort.

Layouts and filters can be given statically at the setup time of the channel or be assigned dynamically after finding an optimized layout or filter for the channel using through modeling, profiling and/or mathematical optimization.

When channels are interacting with actors they are expected to provide **buffers of data** for use during data consumption or data production by the actor. Buffers can be re-used, e.g. when the channel detects that last element of the buffer has been consumed, and standard constructions like double buffering, or ring buffering can be used to enable e.g. overlapping I/O and

computation, stream processing etc. The buffer sizes are parameters to the actor which can be given statically or adjusted dynamically (e.g. through profile driven benchmarks).

Anatomy of running a data flow program

Cluster Architecture Descriptor

Through requesting resources from a job scheduler a list of nodes and their attributes becomes available.

In the first instance the CAD is simply a list of node addresses (e.g. IP & port, or IB addresses). In due course, available memory, cores and accelerators may provide additional attributes.

Program anatomy

A data flow program consists of:

- actors including the channels they set up. They typically have parameters allowing fewer or more actors and channels to be created.
- a function that produces initialization data for the channels based on the CAD and scheduling information
- scheduling the actors
- initiating the data flow

Scope of application

DNA expresses the numerical analysis in terms of kernels (actors) which can leverage accelerators, data source and movements (channels).

It is not our aim to produce fine grained event driven programming - instead, generally, buffers of substantial size will be handed to actors for computation.

A Sample Cloud Haskell DNA program

This section describes how Cloud Haskell optionally with foreign function interfaces can be used to offer a DNA interface, following a simple example.

Specification of a distributed dot product program

As an example we may wish to compute on a distributed system the dot product of two vectors, one read from a file, one created in memory.

A distributed dot product program has the following components:

- a single collector actor summing floats from nodes that did not crash
- a set of dot product computation actors, one to be scheduled per processing element
- file read channels and vector compute channels: one for each dot product actors
- a collector channel from each dot product actor to the collector actor
- **start events** for each dot product actor, which in turn will start its file and compute channels

- a ***schedule function*** that produces setup parameters for actors and channels, based on the number of available processes

In pseudo code the program looks as follows:

```
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
```

File Channels

We choose to make our file channels efficient with the following approach:

1. The file channel functions are implemented in C and can perform 0-copy O_DIRECT I/O into the address space of the C-process. This avoids duplication of data between the kernel page-cache and application memory but is subject to alignment constraints.
2. When the C-program has read a buffer, its pointer is passed to cloud Haskell without any parsing of the data. This is not how cloud Haskell sends messages by default, as it bypasses all type-checks.

Scheduling

We choose the most basic way to schedule processes. The Slurm job scheduler is given a request for resources and when these become available, a cloud-Haskell shell process is started on all but one node. The unique node runs a ***controller (aka master, coordinator)*** process which is given the cluster architecture descriptor: basically the list of available cores. The controller process then computes the setup data for the actors and channels and instantiates and starts the data flow graph.

Cloud Haskell processes must run the same binary on every node if they want to use remote function invocations. Hence, more precisely, the process is started in a controller mode, and then spawns remote processes that instantiate the dataflow graph and resulting computations.

Actor anatomy

Each actor is represented by a cloud Haskell process, which may in turn spawn threads or use accelerators. The program's scheduler, asks the coordinator to assign each actor to a cloud Haskell node. We envisage that eventually a schedule is arrived at dynamically by an optimizer.

Channel anatomy

The common aspects of channels are:

- channels have a source and sink. Usually at least one endpoint of a channel is used by an actor, if it is the sink we say it is used by a receiving actor, otherwise it is used by a sending actor.
- data is delivered into the sink endpoint of a channel and the data structure holding the delivered data (typically a packed array of numbers) is accessible to the receiving process.
- one endpoint of a channel is designated to be responsible for control, which includes setup and teardown of the channel
- channels can receive and emit messages. In particular they send
    a. **ready** messages when setup is complete
    b. **yield** when the sink may consume data and
    c. **finish** when no more data is available.

  They receive
    1. **start** messages to initiate action,
    2. **release** messages to indicate the data buffer consumed may be re-used, and
    3. **exit** messages to self-destruct.
- channels are monitored for failures by their endpoints
- an actor is not ready until all its channels have delivered a **ready** message

There are several different types of channels, and in the first instances we need the following:

- a file reader: A thread in the receiving actor, which also functions as the control actor, opens the file and allocates a buffer. The actor sends a ready message to the control actor. Upon receipt of a start message the actor starts filling the buffer, when the buffer is full it emits a ready message to the receiving actor.
- a node-to-node small message communication channel: Either the receiving actor or the sending actor can perform setup. We will use this channel initially for the purpose of simple collectives only. In this case a ready message must be delivered to both endpoints. The start message may contain the small message that needs to be sent.

- data production channel: a computation thread is started in the receiving thread that fills a buffer.
- linking and monitoring: borrowed from Cloud Haskell
- RDMA node-to-node channels
- RDMA remote to/from accelerator channels: move data from accelerator memory to/from files or accelerator or RAM memory on other nodes.

A channel consists of threads in the actors, possibly an external process, and these are instantiated when the channels are set up. The buffer at the sink end of a channel is a (large) array in the actor containing the sink.

Implementation Description

Channels

The behavior of the channel is:

- the instantiation of the channel is a synchronous function in the controlling actor
- the actions of an actor and its channels occur in separate threads
- an actor receiving from a channel can block or receive asynchronous notification of the results
- an actor can request a channel to deliver data multiple times, e.g. reading segments of data into a ring buffer layout
- actors can monitor their channels if they are implemented as cloud Haskell processes

ComputeChannel

A compute channel is a computation that fills a buffer in the spawning (receiving) process. It can be created with the following parameters:

- the results buffer of size k x sizeOf(a) in address space of the receiving actor
- the actor receiving its ouput identified by a ProcessId and/or an integer rank N
- the number k of computations it should perform
- the function that is being computed f: ProcessId -> Int -> a (a is the output type)
- note: ComputeChannels may require more parameters to do their computations

FileReadChannel

A file read channel reads (C language) data structures from a file and places them in a buffer residing in the address space of the receiving process.

- the results buffer of size k x s in address space of the receiving actor
- the actor receiving its ouput identified by a ProcessId and/or an integer rank N
- the filename from which to read data
- the offset o, the size s of each object, and count k of objects to read

AggregationChannel

Many actors can form a channel to a single collector process. To instantiate it the actor needs to know the processId of the receiving (collector) process.

The collector will typically need to know all nodes from which it can expect results. If some nodes may fail, it also needs knowledge how to handle the failures (e.g. ignore them in the collection phase or re-spawn the computations).

Other design considerations

High Availability

DNA has support to manage high availability. The controller process can monitor other processes and take actions to handle the failure. Sample actions can consist of:

1. Ignoring a failure, but removing the failed process from the processing by a collective operation
2. Re-directing requests after a failure

Depending on the scope of recovery after a failure, strongly consistent storage may be needed.

Monitoring tools

To monitor execution of jobs and detect program failures and errors, a standard set of Unix tools and SLURM monitoring utilities is used:

- **top** to display information about running processes, memory and CPU utilization
- **sacct** to report job or job step accounting information about active or completed jobs managed by SLURM
- **sinfo** to report the state of partitions and nodes managed by SLURM
- **squeue** to report the state of jobs or job steps

To examine and to report errors / failures in produced logs, the **watch** Unix command is used. **Watch** executes a command periodically, for example the following command will report every new failure found in the log file “out”, updating results every second:

```
watch -n 1 grep failure out
```

Additional monitoring tools and utilities can be developed and deployed in the future.

DNA DSL architecture

(this section was written by Aleksey Khudyakov)

This section describes a second approach taken to construct a DSL, this case DNA compiles to languages for target architectures. It contains some open questions, *in italics*.

Overview

Currently DNA DSL (simply DNA from now on) is realized as compiler from eDSL to cloud haskell (CH) program. Other backends (Halide, Legion, C+MPI are considered as well). It consists of several packages:

- **dna** — package containing compiler itself.
- **dna-runtime** — library for DNA programs compiled to cloud haskell.
- **dna-examples** — collection of example programs

Note that DNA is in very early development so this document reflects the current state and it may become out of date quite soon. Also there's no DNA/RC separation yet because DNA at this point is not mature enough to express any significant libraries.

Overall documentation is structured in following way: section 2 describes DNA language itself, section 3 compiler and section 4 gives overview of modules of **dna** package.

DNA language

General description

DNA is a dataflow language and a DNA program consists of set of actors which can send messages to each other. Note that there no one-to-one correspondence between actor and threads/processes/cluster nodes. Single node may execute several actors or single actor may use several nodes.

Every actor has a set of of destinations to which it can send a message. We will call such destination port and set of such destinations collection of ports. Connections between ports actors are specified during construction of dataflow graph. Actor itself have no knowledge about dataflow graph. It only could send message to port. Where sent message will actually go is defined by dataflow graph.

There are three sublanguages in DNA 1) small functional language for defining expressions and two very simple monadic DSLs for definition of actors and dataflow graph. Latter two are still very raw and could change very significantly in the future.

Functional language

We use small functional language to define calculations which are performed by actors. At this point language itself is very ad hoc and its design is yet to be finalized.

It's implementation is modelled after accelerate's approach. Internally language is represented using De Bruijn indices. They are however unsuitable as DSL since notation is cumbersome and difficult to follow. So user facing DSL uses higher-order abstract syntax representation which is convenient to program in. Then during compilation it's converted to the De Bruijn representation. (HOAS and conversion is not implemented yet).

There's also questions about description of effects. At the moment we pretend that all our computations are pure. In this case we are free to do many different program transformations. Most importantly we could parallelize some actors.

eDSL for actors

So far we have three different types of actors. Its minimal set of actors which is required for implementation of distributed dot product. This list is not final and more actor type could be added in the future or these actors could be modified.

Every actor could have several output ports. Set of ports is specified during construction of an actor and ports are connected to other actors during construction of dataflow graph.

1. **State machines actors**. These actors receive messages and perform computation. It takes message and current actor state as input and produces new actor's state and messages which actor could send. Such transition rule could be summarized as:

   ```
   (state, message) → (new state, messages to send)
   ```

   Actor could have several such transition rules. They however must all have different message types since message are dispatched by type.

2. **Data producers.** These actors produce infinite stream of data. They are required since state machine actors could only send messages in response to received messages thus dataflow graph which consists only from dataflow actors could not make any progress.

   Data producer performs single computation in the infinite loop. Given current state it computes new state and set of messages to send. It could be summarized as:

   ```
   state → (new state, messages ∘ send)
   ```

   It's important to notice that if downstream actor consumes data slower than producer generates it then former's mailbox will be flooded with messages. To avoid it we may

need to block until downstream actor is ready to receive more messages.

Another concern is finitiness. We may need data producer which could only produce only finite amount of data. This question is closely coupled with termination problem described above.

3. **Scatter-gather actors.** They are actors for evaluating pure function in parallel. For every incoming message it calculate function's result and sends its. Scatter-gather actor uses function of special form which could be decomposed into triple:

   i. Scatter function which divide function input into N inputs for N independent subtasks. (Its type: `Int → a → [c]`)
   ii. Worker function which is computed in parallel on N worker nodes
   iii. Gather function which merge values into final answer.

   Operationally scatter-gather actor composed from single master process and N worker processes. When master process receive message it generate N inputs for worker processes and sends one to each worker. Workers calculate partial results and send them back to the master which merge them with gather function and sends final result to other actor.

   So far it's only example of actor which doesn't corresponds to single process in generated code.

   It could be possible to generalize scatter-gather actors to have state like state machines and have more than one handler for incoming messages.

   It's possible to add fault tolerance to this actor in form of handling actor failures. There are two possible strategies: fail-out when we ignore failed nodes and restarts when we resend message sent to failed worker to another one.

All these actors are defined using simple monadic DSL.

eDSL for defining dataflow graph

Again very simple monadic DSL is used to define dataflow graph. Monad is called dataflow. It only contain two primitives:

```
use :: ConnCollection outs => Actor outs -> Dataflow (A, Connected outs)
```

It binds actor into dataflow graph and returns handle to the actor and collection of actor's outgoing connections. We need this function because Actor data type contains description of

internal working of an actor. For example we can have more than one identical actor in the dataflow graph.

```
connect :: Connection a -> A -> Dataflow ()
```

Second primitive connects connection port of actor to another actor. `Connection` data type holds both actor's ID and port ID inside an actor.

`ConnCollection` is type class for description of collection of actor's ports. We need because inside we need to add actors's ID in the dataflow graph to each port.

`Connected` is associated type for `ConnCollection` type class. It's type for connection to which we added actor's ID. For single port we it's `Connection a` and for tuple of ports it's tuple of `Connection`

DNA compiler

DNA compiler is rather straightforward and include following stages:

1. Checking of dataflow graph. During this stage we check that every actor is correctly constructed and each port of every actor is connected somewhere.

2. Scheduling. During this stage we determine how to spread actors over available nodes. In particular scheduler decides how many actors allocate to scatter-gather actors. As input it takes validated dataflow graph and cluster architecture description (currently it's number of available nodes).

   Current scheduler is very simplistic it schedules for fixed number of nodes. If program was scheduled for N nodes is will use at most N nodes and fail if not enough nodes was supplied. Nor it takes into account execution cost of an actor. For every actor we allocate whole node.

3. Code generation. Code generation for cloud haskell (CH) is very simple. Compilation of functional language is very close to simply unparsing AST.

   Actors corresponds to the CH processes. We create top level definition of type `Process ()` for state machine and producer actors and two for scatter gather actors (for master and worker). We also create master function which spawns processes and monitor their execution. Most of the reusable functionality is in the **dna-runtime** package.

   Generated program will take following command line parameters:

   ```
   ./program [master|slave] host port
   ```

They have following meaning. Master means that program will actor as master node and will spawn processes on other nodes and monitor. There should be only one master. Host and port determine on which host and port program will listen.

Note that we don't have any optimizations yet. Optimizations are allowed to change dataflow graph. They could split actors or merge actor into one. Whether such transformations are beneficial could depend on number of nodes so it's tightly interwoven with scheduling phase.

Overview of dna package

Here is list of module and their purpose. This is current sate of package and things will undoubtedly change.

Module DNA

Public interface to the DNA. It provide reexports and DSLs for definition of actors and dataflow graph. It's not complete and export from the internal modules are required so far.

Module DNA.AST

Abstract syntax tree for the functional language. It's internal representation which uses De Bruijn indices.

Module DNA.Actor

Data type for definition of dataflow graph.

Module DNA.Compiler.Basic

Simple checks for the dataflow graph

Module DNA.Compiler.Types

Common data type for the compiler. It include monad for error handling and source of fresh names.

Module DNA.Compiler.Scheduler

Very simple scheduler for dataflow graph.

Module DNA.Compiler.CH

Code generation of for cloud haskell.

Runtime Scheduler

This section defines how a Slurm scheduling script interacts with data ingest into node local storage and with the distributed Cloud Haskell program. The most tricky technical aspect of this work is to maintain the sequences of data chunks ingested exactly in sync with the distributed CH processes accessing them.

Requirements

1. Define number of nodes
2. Define processes per node
3. Define itemcount
4. Create INPUT symbolic link pointing to /ramdisks/INPUT.$SLURM_JOB_ID
5. Key issue to address is that the i-th node in the CAD file handles the same chunk number as the Haskell program does. The Haskell program is instructed by the coordinator which parses the CAD file. We must parse the CAD file to get the same sequence number of the node. NOTE: the Slurm task number is not known to the slave nodes
6. We assume the program is started in the ddp-erlang-style source directory (TODO: get everything in the path)
7. We create a subdirectory D=$SLURM_JOB_ID
8. We create subdirectories D/IP/PORT in which the profiling (eventlog) files will be placed

Description

Phases of the program

1. sbatch reserves the resources and sets environment variables
2. a CAD file is generated
3. input files are created -
4. the Haskell program is started - $PROC_PER_NODE times ; move eventlog file
5. clean up input files

Step 3 requires to find which chunknumbers are to be created on a node, given its IP address. This must be processed in the same manner as the Haskell program does, otherwise chunk numbers on a node go wrong. Haskell uses the line number in the cad file. For each IP we should find PROC_PER_NODE lines and associated chunks.

Step 4 must determine have the ip and create a port number. As during the generation of the CAD file, the port number for each IP go from 44000 through 44000 + $PROC_PER_NODE -1

Implementation

dna-sbatch.sh

1. Slurm resources
    a. --nodes=100
    b. --tasks=100
    c. remainder as in peter's job file
2. execute
    a. export $ITEM_COUNT=15000000
    b. export $PROCS_PER_NODE=12
    c. export $NODES=nodes.txt
    d. export $CAD=CAD_dna.txt

e. mkdir $SLURM_JOB_ID
f. chdir $SLURM_JOB_ID
g. export NODEFILE=`generate_pbs_nodefile`
h. cat $NODEFILE | uniq > nodes.file ; rm $NODEFILE
i. dna_cad.py $NODES $CAD
j. ln -f /ramdisks/INPUT.$SLURM_JOB_ID INPUT
k. srun --exclusive ../create_floats.py cadfile
l. srun --exclusive ../ddp.py cadfile
m. srun --exclusive rm /ramdisks/INPUT.$SLURM_JOB_ID
n. rm INPUT

Sub program specification:

1. dna_cad.py - generate CAD file from Slurm hosts (or use Jaya's)
2. dna_lib.py - have available the following functions
    a. i -> 0 .. (lines in cadfile -1)
        i. host(i, cadfile) - get from cad file
        ii. ip(i,cadfile) - compute from host(i)
        iii. port(i) = 44000 + mod i 12
        iv. chunk_number(i, cadfile) = i==0 || i ==1 : then 0 else i-1
        v. chunk_count(cadfile) == lines(cadfile) - 2
        vi. role(i) = i=0: master, else slave
    o. ip address and cadfile →
        i. my_lines(ip, cadfile) -- list of line numbers in cadfile (i's) matching ip. ASSERT the number of these matches $PROCS_PER_NODE
    p. myip() -- get it from hostname
3. create_floats.py cadfile: for data writing run in D should get $PROCS_PER_NODE execution of create_floats.c)
    a. port = 44000
    b. for i in my_lines(cadfile) - execute
        i. debug print: myip, port, chunk_number(i, cadfile)
        ii. call ./create_floats INPUT $ITEMCOUNT chunk_count(cadfile) chunk_number(i, cadfile)
        iii. port = port + 1
4. ddp.py cadfile: run in directory D; should get $PROC_PER_NODE invocations of ddp-erlang-style which have to run in subdirectories.
    a. import dna_lib.py
    b. my_ip() -- who am I?
    c. mkdir my_ip()
    d. i is in my_lines(ip, cadfile), port = 44000 .. 44012 -- execute in the background:
        i. mkdir ip(i,cadfile)/port(i,cadfile)
        ii. chdir ip(i,cadfile)/port(i,cadfile)
        iii. if (role(i) == master) sleep 5

iv. ../../../ddp-erlang-style role cadfile --ip ip(i, cadfile) --port port(i) ../../INPUT +RTS …..

v. mv ddp-erlang-style.eventlog ../../eventlog.my_ip().port

e. **wait** for all of (c) to complete, but let them run in parallel (cannot wait for last one!)