SKA Imaging Software
Designing with domain specific languages

Braam Research, LLC
1983 - 2000 Academia
• Maths & Computer Science

Entrepreneur with startups
• 4 startups
• Lustre emerged
• Held executive jobs with acquirers

2014 – Independent research
• Primarily work with SKA SDP @ Cambridge
• Work on Imaging HPC software and storage
• Help others
What is the project about?

- Imaging software for radio telescopes has proven to be complex
- Can we leverage state of the art programming language techniques to make it much simpler?

Key requirements remain:
- Separate layers for application software and low level compute kernels
- Easy modifiability
- Automatic optimization
- Data flow approach
- Plan to integrate work from others
- Use state of the art compute cluster & cloud ideas
Content of talk

- Very quick sketch of the imaging problem
- Data flow programming
- Automatic optimization
- Radio Cortex (RC) and Declarative Numerical Analysis (DNA)
- Next steps
The Radio Cortex / DNA project will produce quarterly reports

- What was the focus
- Results from study
- Results from prototyping
- References

Gradually more information will appear on GitHub to allow others to experiment

- Report 1: http://goo.gl/0n75aa
- Report 2: expected Dec 15
Imaging software description
Input is data from baselines

There are ~1000 antenna’s
Hence 500,000 baselines

Each baseline measures 256K frequency channels

Correlator gives a measurement ~2x / sec
Each measurement is 3 complex numbers and 3 coordinates (~30 bytes)

30b x 256K x 500K x 2/sec ~ 7.5 TB /sec

Baselines are not regularly laid out on a grid
Imaging pipeline

Operations are simple:

- Put baselines on grid (“gridding”)
- Project one grid onto another
- Fourier transform
- Subtract known grid values
- De-grid

Several steps are repeated, some 10 times

Optimizations

- Data locality
- Data movement
- Fast computation

The optimizations require a data centric approach as much as choosing good algorithms
Another perspective

- Exploit frequency independence
- Sort and distribute visibility data and target
- Gather target grids
- Further data parallelism in locality in UVW-space
- Use to balance memory bandwidth per node
- Some overlap regions on target grids needed
- UV data buffered either on a locally shared object store of locally on each node

Feb 2015

Copyright: Braam Research, LLC
Architectural principles

- **DSL**
  - Domain Specific Languages & Data Flow
  - Express algorithms use of kernels concisely

- **Strategy**
  - Determine what to compute where
  - Address parallelization & locality

- **Compile**
  - Compile optimized kernels
  - Schedule on cluster

- **Execute**
  - Run the program
  - Adapt to failures
Data Flow Programming
Basic Principles

Express a computation using actors connected with data channels where:

- the actors fire when all required data on their input channels is available
- data is exclusively owned by an actor or a channel

Contrast with multithreaded programming:
- avoid state accessed by all threads as part of progress of the computation (concurrency control).
- All actors and channels do compute concurrently.
Map reduce as dataflow
Variations

**Variations**
- How is the graph encoded
- Actors can spawn dynamically
- One or more input channels
- Channels perform *matching*
- Are channels ordered?
- Are actors stateless / state full
- Select expected messages from channels
- Unexpected msg stay in the channel or may crash an actor
- Channels reliable / unreliable

**Examples**
- Hardware design languages
  - clocks
- Actor model
  - no fixed graph
  - deep theory
- Reactive programming
  - more stream oriented
- Event driven programming
  - origins in GUI
- Cell driven programming
  - like spreadsheets
Lofty claims and lots of confusion

Actor model wikipedia: This section may be confusing or unclear to readers. In particular, links between paragraphs are unclear. Seeming non sequitur, or confusing language in second paragraph. (September 2014)

According to Carl Hewitt, unlike previous models of computation, the Actor model was inspired by physics including general relativity and quantum mechanics.

I now realize that Robin [Milner]’s work [on Calculus of Communicating Systems (CCS) and pi-Calculus] should really have been included in the previous chapter, but I just wasn’t aware of it when I wrote my book.
History

- Goes back to the 60’s (IBM)
- Is absolutely vast
- Includes some of the finest computer science literature, such as Robin Milner’s work on the pi-Calculus
- Many dozens of deep theoretical models
- Many 100’s of languages
- It’s a darling of many areas, including super computing
Examples

- Key primitives: send data items and wait for them
- “Ebay” – the channels build up complex state and do “joins”
  ```
  expect Buyer itemTypeA, Seller itemTypeA -> arrange sale
  ```
- Simple to deadlock
  ```
  actor A = do
  b <- waitfor: from B
  send a
  actor B = do
  a <- waitfor: from A
  send b
  ```
- Different to debug – history vs stack
- Easy to get very complicated things
Automatic Optimization
General structure

- I do not know the full history, there are dozens of automatic optimizers

- Famous example is FFTW
  - DFT’s can be factored. Locality of data is key.
  - FFTW automatically generates numerous strategies and returns optimal one
  - Core algorithm is (monadic) functional program, output is C (or lower)

- How does it work?

```c
fftw_complex *in, *out;
fftw_plan p;
in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
fftw_execute(p);
```
Optimization using Halide

Halide is a language for image processing – used for cameras.

**Algorithm:**
what is computed?

**Schedule**
Question 1: In what order should it compute the output
Question 2: In what order should it compute its inputs

Separation of Algorithm and Schedule is much better:
tinkering with optimizations can't break the algorithm

Halides’ optimizations
- **parallelism**: threads, SIMD vectors
- **locality**: tiling, fusion (including re-computation, duplicating data)
- unfortunately not yet “binning” our 500K baselines
Example - blurring

Var x,y
Function blurx, blury
blurx(x,y) = (inp(x-1,y) + inp(x,y) + inp(x+1,y))/3
blury(x,y) = (blurx(x,y-1) + blurx(x,y) + blur(x,y+1))/3

Multiple scheduling strategies will be shown in a movie

Play 2 short segments from Halide movie:
14:25 – 17:28
19:17 – 21:31
Halide Compiler

Halide Functions    Halide Schedule

Imperative Blob

LLVM bitcode

X86 (with sse)      ARM (with neon)      CUDA
Gridding with Halide

result(u, v, pol, x) = (T)0.0;
result(u, v, pol, 0) +=
    weightr(bl, intU(bl, timest), intV(bl, timest),u,v)
    * visibilityr(bl, timest, pol);

- Didn’t work so well ....
  - += obtains much concurrency (which a different scheme can avoid)
  - Concurrency is expensive

- Halide does like “indexing” arrays with the values of others, e.g. by using
  the baseline coordinates (Halide is built for “whole” regular camera
  images”)

- Yet, Halide demonstrates extremely well how to organize the code

- And searches automatically for optimized algorithms.
Radio Cortex – RC & Declarative Numerical Analysis - DNA
Target is to produce a compelling design & prototype (2 years)

Milestone 1:
Problem: dot product of a computed vector and vector in a file
Used Lustre shared storage and cloud-per node storage model
Ran it with Slurm
Implemented data flow program with cloud Haskell
Ran it up to 1200 cores on Wilkes
Had high availability operational in version 1
Did careful profiling
Integrated it with C-code

We learned things needed to become a lot simpler!
Milestone 2: gridding

Basic Gridders – turned out to be basic and not so basic
Compare with Romein’s gridding
Much simpler DSL
Much easier debug & profile log management
More precise profiling
Use GPU’s and CPU’s
Run again at scale
Keep high availability
Tighter integration with Slurm
Simpler DSL – map reduce

master_actor (CAD, M, R)
  mapProcs = schedule(M, CAD)
  reduceProcs = schedule(R, CAD)
  fork(nodes:reduceProcs, process:reduce_actor, input:mapProcs, crash:fail, output: File)
  result = wait(reduceProcs)
  start

map_actor
  map computations ....
  join(zip(map_outputs, reduceProcs))  -- sends the map output to the reduceProcs and exits

reduce_actor
  wait(input, mapProcs)
  reduce computations ....
  join(result, parent)
What’s next with RC / DNA?
Possible next steps

- By end of 2015 fully working prototype for imaging
- Collaboration with Intel and nVidia to explore integration of fast kernels
- Collect domain specific knowledge into systematic design documents
- Plan for success!
Thank you! Questions?