Ypnos: Declarative, Parallel Structured Grid Programming

Dominic Orchard, Max Bolingbroke, Alan Mycroft

November 20, 2009

Computational Patterns

Many computational patterns with different parallelisation schemas[ABD+08]:

- Dense Linear Algebra (BLAS, MATLAB...)
- Spectral Methods (FFT)
- MapReduce
- N-Body Methods (Barnes-Hut)
- **Structured Grids** (Fluid Dynamics)
- etc...

Structured Grids

- Arrays representing discretised environments
- Apply a stencil function over an array
- Computes a new value for each array element from neighbours

(a) 5-point stencil (b) 9-point stencil

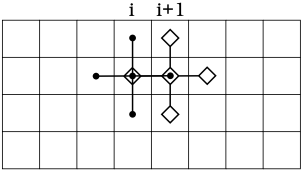
- Typical in scientific computing, graphics, games, etc.
- Usually solve approximations to differential equations over discrete space: fluid dynamics

Structured Grids: C Example

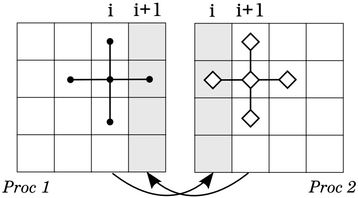
```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] = (A[i+1][j]+A[i-1][j]+
                       A[i][j-1]+A[i][j+1])/4.0;
     }
   }
   swap(Atemp, A);
}
```

Parallelisation

Traditional domain decomposition technique:



(c) Before decomposition



(d) After decomposition

Figure: 2D domain decomposition with a 5-point stencil

- Subgrids + overlap regions reside in separate processes
- Boundaries communicated between iterations

Problems with Current Approaches

Manual Approach

Typical approach: C/FORTRAN parallelised manually with MPI (Message Passing Interface).

- Prone to errors as hard
- Difficult to debug
- Tedious
- Difficulty increases with number of grids, dimensions, transformations

Imperative Languages

- Can be efficient, but easy to write incorrect parallel programs:

```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] = f(A, i, j);
     }
   }
   swap(Atemp, A);
}

f(A, i, j) {
      ... = global
      ...
      global = ...
}
```

Frameworks

- CUDA, Cg, and OpenCL
  - Specific to GPU execution
- OpenMP
  - Specific to shared memory
  - Programmer must ensure safety
- MPI
  - Pretty portable
  - But difficult

Lots of approaches are implementation specific.

Automatic Parallelisation

- Most languages are too general
  - Does program X fit computational model Y?
- Arbitrary and random-access indexing renders analysis and automatic compilation difficult (in general undecidable):
  - Do read and writes overlap?
  - How much boundary communication is required?
- Therefore, restrictions placed on programs: affine indices, affine control flow tests, etc
- Most scientific programs conform[BCG$^+$03], but difficult for novices.
- Can lead to discontinuous compiler behaviour.

Ypnos

Our Approach

- Ypnos: DSL specifically for structured-grid programming
  - DSLs offer problem or implementation specific
    - expressivity
    - optimisations
- Embedded into Haskell
  - Embedded DSLs
    - Inexpensive DSL
    - Reuse host language syntax, semantics, libraries, etc.

Ypnos

- Pure, functional semantics
- No manual parallelisation/optimisation
- Decidable compile-time information
- Programmer specified, guaranteed optimisation and parallelism
- Permits different backends $\therefore$ hardware-agnostic

Core of Ypnos

- Grid data structure
- Use-defined stencil functions
- Various primitive operations
- Haskell-like syntax, with our own novel augmentation

Grids

- Grid data structure, has type:

$$\text{Grid } D \; a$$

- D is a dimension identifier:

$$D := dim \mid D \; \times D$$

- E.g.

$$\text{Grid } (X \times Y) \; Float$$

  is akin to `float[][]`.
- While C just has one type `float[][]`, Ypnos has Grid $(X \times Y)$ *Float*, Grid $(Y \times Z)$ *Float*, ...

Stencil Functions

We could abstract the C example with a stencil function `f`:

```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] = (A[i+1][j]+A[i-1][j]+
                       A[i][j-1]+A[i][j+1])/4.0;
     }
   }
   swap(Atemp, A);
}
```

Stencil Functions

We could abstract the C example with a stencil function `f`:

```
while(condition) {
   for (int i=0; i<N; i++) {
     for (int j=0; j<M; j++) {
        Atemp[i][j] = f(A,(i,j));
     }
   }
   swap(Atemp, A);
}
```

Stencil function, `f` :: Array a $\times$ $(Int \times Int) \rightarrow a$

Stencil Functions in Ypnos

- Stencil functions in Ypnos have type:

$$\text{Grid } a \rightarrow b$$

- cf. Array $a \; \times (Int \times Int) \rightarrow a$
- The current index $(Int \times Int)$ is hidden inside Grid, called the cursor or focal point.

Grid Access

- No random access functions for reading or writing Grid!
- Read via special pattern matching syntax: grid pattern
- Example:

  $$X : |\ \ l\ \ @c\ \ r\ \ |$$

  Binds variables to values in grid. Equivalent to:

  ```
  l = A[i-1];
  c = A[i];
  r = A[i+1];
  ```

  where `i` is the focal point
- @ syntax denotes which variables is bound to the focal point.
- All other bindings are relative to this point

Grid Patterns (2)

- One-dimensional patterns can be nested
- Also 2D pattern syntactic sugar. E.g 5-point stencil:

$$(X \times Y) : \begin{vmatrix} \_ & t & \_ \\ l & @c & r \\ \_ & b & \_ \end{vmatrix}$$

- Applying an N-dimensional pattern to an M-dimensional grid (where $N < M$) performs <u>slicing</u>.

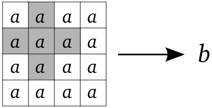
Example: Laplace Stencil Function

```
laplace :: Grid (X * Y) Double -> Double
laplace (X * Y): | _ t  _ | = (t+l+r+b)/4.0
                 | l @_ r |
                 | _ b  _ |
```

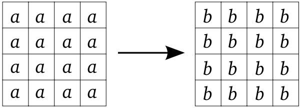
Applying Stencil Functions

$$\texttt{run} :: (\text{Grid } D\ a \to b) \to \text{Grid } D\ a \to \text{Grid } D\ b$$

- Applies a stencil function at every point in a grid.
- Collects results into a new grid.



$f :: \text{Grid } a \to b$



$\texttt{run } f :: \text{Grid } a \to \text{Grid } b$

Example

```
laplace (X*Y):| _ a _  | = (a+b+d+e)*0.25
              | b @_ d |
              | _ e _  |

g = grid <X = 10, Y = 10> data
g’ = run laplace (defaults 0.0 g)
```

Optimisation and Parallelisation

Single, Independent Writes (SIW) Property

Definition

Each stencil function application by `run` produces a single result belonging to a single, unique position in the results grid. Thus write operations to a results grid are all independent.

- SIW is guaranteed for all Ypnos programs.
- We can use this to give optimised and parallelised forms of stencil application.

Optimisation

- Ypnos is pure, functional ∴ `run` creates a new grid in memory.
- In the C-example, two array allocations are destructively updated:

```
while(condition)
   for (int i=0; i<N; i++)
     for (int j=0; j<M; j++)
        Atemp[i][j] = (A[i+1][j]+A[i-1][j]+
                       A[i][j-1]+A[i][j+1])/4.0;


   swap(Atemp, A);
```

Optimisation: Iterate (1)

- Usually want to iteratively apply `run` until some stop condition:

$$\begin{aligned}\texttt{iterate}\ f\ \ r\ \ g = &\ \text{if}\ \texttt{condition} \\ &\ \text{then}\ \texttt{iterate}\ f\ \ r\ (\texttt{run}\ f\ \ g) \\ &\ \text{else}\ g\end{aligned}$$

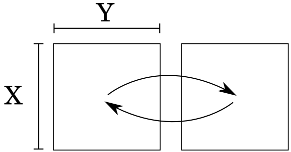
- Ypnos provides an `iterate` primitive using local mutable state as an optimisation

Optimisation: Iterate (2)

$$\texttt{iterate} :: (\text{Grid } D\ a \rightarrow a) \rightarrow \text{Reducer } a\ Bool$$
$$\rightarrow \text{Grid } D\ a \rightarrow \text{Grid } D\ a$$

where Reducer is a special reduction operation structure.

- Allocations are used cyclically (cf. `swap` in the C-example)



- Mutable grids are produced and consumed inside `iterate`: effects can't leak out.
- Guaranteed to be safe by SIW property.

Optimisation

What if we want to use earlier versions of a grid in a computation?

- Alias grids between calls to `run`
- To optimise with destructive update, try to do an alias analysis
- But intermediate grids may escape and be referenced later after deallocation, update etc.

Optimisation: IterateT

- `iterateT` lifts a computation into a reserved temporal dimension, reusing grid patterns over temporal dimensions.

$$\texttt{iterateT} :: (\text{Grid } (T \times D)\ a \rightarrow a) \rightarrow \text{Reducer } a\ Bool \\ \rightarrow \text{Grid } D\ a \rightarrow \text{Grid } D\ a$$

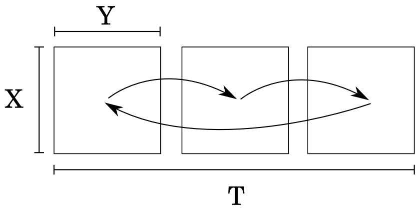
- Grid patterns in time are historic patterns.
- Historic patterns give decidable compile-time information about the number of grids involved in a computation.

Optimisation: IterateT (2)

Example:

$$T : | \ g'' \ \ g' \ \ \texttt{@\_} \ |$$

`iterateT` creates three intermediate allocations:



Optimisation (4)

- `iterate` and `iterateT` are optimised primitives
- Neither extends expressive power of Ypnos
- Optimisation is not result of analysis+transformation.
- Programmer-specified. SIW guarantees safety.
- Code communicates to the reader and compiler when optimisation occurs $\therefore$ predictable.

Parallelisation: SIW wins again

- Stencil application can be performed in any order, or in parallel, as write operations don't overlap.
- ∴ we could execute each application of a stencil function in parallel
- But- too much overhead, use domain decomposition technique.

Domain decomposition

- Grid patterns provide size of overlap regions.
  - Syntactical inconvenient to write large access patterns ∴ communication overhead is small.
- Tells us if communication across some dimensions is unnecessary, e.g.

```
f (X * Y): | _ _  _ | = g(l, c, r)
           | l @c r |
           | _ _  _ |
```

- runPar, iteratePar, and iterateTPar:
  - Decompose a grid
  - Distribute to processing elements
  - Locally apply sequential `run`, `iterate`, etc.
  - Communicate boundaries between iterations for `iterate` and `iterateT`
  - At the end, gather subgrids back and return pure grid.
- All safe by SIW.

Further Ypnos Constructions

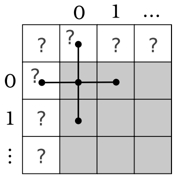
Reducers

$$\begin{array}{rlll} \text{mkReducer} :: & (a \to b \to b) & \textit{partial reduce} \\ \to & (b \to b \to b) & \textit{combine partials} \\ \to & b & \textit{initial partial result} \\ \to & (b \to c) & \textit{final conversion} \\ \to & \text{Reducer } a\ c & \end{array}$$

- Allows parallel reductions when data is distributed.
- Used by `iterate` and `iterateT` or:

$$\text{reduceR} :: \text{Reducer } a\ b \to \text{Grid } D\ a \to b$$

Boundaries



- Must deal with boundary behaviour
- Ypnos has two approaches

Boundary Approach 1

Manually account for boundaries

- Provided by # pattern:

$$X : | \; l \;\; @c\#i \;\; r \; |$$

  binds an index expression to i which can be tested to see if application is at a boundary.
- Requires extra, tedious code
- Requires repeated testing of boundary conditions.

Boundary Approach 2

Lifted grids.

- Grids are finite; lifted grids are finite grids embedded in a potentially infinite space.
- Requires a special *facets* structure describing boundary behaviour for each face, edge, etc.

$$\begin{aligned}\text{lift} &:: \text{Grid}\ D\ a \rightarrow \text{Facet}\ D\ a \rightarrow \text{Grid}^{\infty}\ D\ a \\ \text{unlift} &:: \text{Grid}^{\infty}\ D\ a \rightarrow \text{Grid}\ D\ a\end{aligned}$$

- Or simple defaulting behaviour:

$$\text{defaults} :: \text{Grid}\ D\ a \rightarrow a \rightarrow \text{Grid}^{\infty}\ D\ a$$

- `iterate`, and `iterateT` are overloaded on lifted grids.
- But, `run` loses its lifting, as the output grid element type is allowed to differ from that of the input grid, thus:

$$\text{run}^{\infty} :: (\text{Grid}^{\infty}\ D\ a \rightarrow b) \rightarrow \text{Grid}^{\infty}\ D\ a \rightarrow \text{Grid}\ D\ b$$

Example

Conway's Game of Life

```
life (X*Y):| a b  c | = let local = (a+b+c+d+e+f+g+h+i)
           | d @e f |   in  if (e==1) then
           | g h  i |         if (local<2 || local>2)
                               then 0 else 1
                             else
                               if (local==3)
                               then 1 else 0

-- Create environment
initalState = grid <X=10, Y=10> randomConfiguration

untilMostlyDead = Reducer (+) (+) 0.0 (\x -> (x<10))
stopCondition = (untilMostlyDead `orReducer` (ntimes 100))

intialState' = defaults 0.0 initialState
finalState = iterate life stopCondition initialState'
```

Implementation

Implementation

- Currently Haskell EDSL.
- Haskell's strong typing rejects non-confirming programs.
- Implementation can be parameterised by Grid giving different backends, giving:
  - Pure, sequential exection
  - Parallel execution via domain decomposition
  - Or, further code for compilation (C, CUDA, MPI, etc)

Implementation (2)

- Current implementation is simple, with no domain-decomposition
- Hopefully fully parallelising implementation for January (performance numbers).

Conclusions and Further Work

Further Work - Domain Decomposition

- Currently no control over the fine-grained detail of domain decomposition, e.g.
  - Tile size
  - Memory layout
  - Processor Layout
- Some further configuration by the programmer, perhaps with some inline compiler directives, would give better performance control.

Further Work - Supporting Vertex Operations

- Ypnos supports *pixel*, or *fragment*, -style gather operations of type Grid $a \longrightarrow b$ (coKleisli arrows for those in the know!)
- Could *vertex*-style scatter operations of type $a \longrightarrow$ Grid b be supported (Kleisli arrows): **mould** functions, with a dual of grid patterns

Further Work - etc.

- Multi-scale methods (adaptive mesh)
- Compiler hints on dimension types e.g.

  $$[[X]] \times Y$$

  means, decompose X dimension but not Y, or I don't care about Y.
- Execution masks for red-black application

Conclusions

- Ypnos is a simple DSL for structured grid programming
- It is sufficiently restricted as to support the SIW property
- This allows optimised and parallelising forms of its grid application primitives that are guaranteed to perform optimisations and parallelisation
- Easier to use for non (parallel) programming experts.
- Possibly slower than expert-programmed manual approach, but development time may be as much of a factor as execution time.

Krste Asanovic, Ras Bodik, Demmel, et al., The Parallel Computing Laboratory at U.C. Berkeley: A research agenda based on the Berkeley view, Tech. Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.

Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam, Putting Polyhedral Loop Transformations to Work, Research Report RR-4902, INRIA, 2003.