check! ☞ http://paraiso-lang.org/wiki/

「**Paraiso**」project

for automated generation and tuning of hyperbolic partial differential equations solvers for parallel and accelerated computers in Haskell

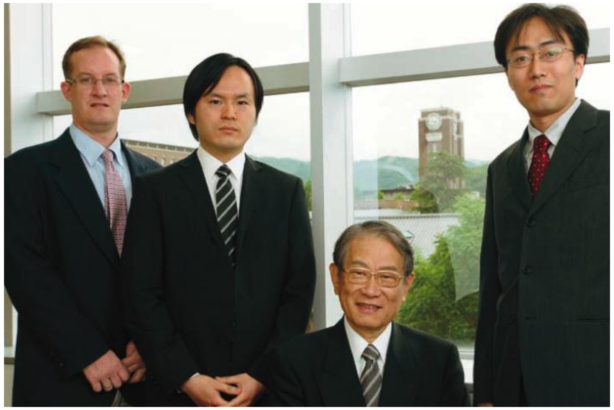
Takayuki Muranushi @nushio

Astrophysicist, Assistant professor at

The Hakubi Center, Kyoto University（2010-2015）

The **Hakubi** Center, Kyoto University

Since 2010



- Unique researchers wanted from all the world
- max. 20 people / year, 5 years position
- Any field of science: natural, life, engineering, social studies, philosophy, …
- Salary of Assistant Prof. / Associate Prof. + research funding
- No mid-career assessment & lay-off
- No education duty
- No PhD required to apply
- No tenure track

quick start guide

Install Haskell Platform and git, then type

```
> git clone https://github.com/nushio3/Paraiso.git
> cd Paraiso/
> cabal install
> cd examples/Life/        #Conway's game of life example
> make lib
> ls output/OM.txt
output/OM.txt              #this is analysis result for dataflow graph
> ls dist/
Life.cpp  Life.hpp         #an OpenMP implementation
> ls dist-cuda/
Life.cu  Life.hpp          #a CUDA implementation
> cd ../Hydro/             #hydrodynamics simulator example
> make lib                 #this takes half a minute or so
> ls output/; ls dist/; ls dist-cuda/    #same as above
```

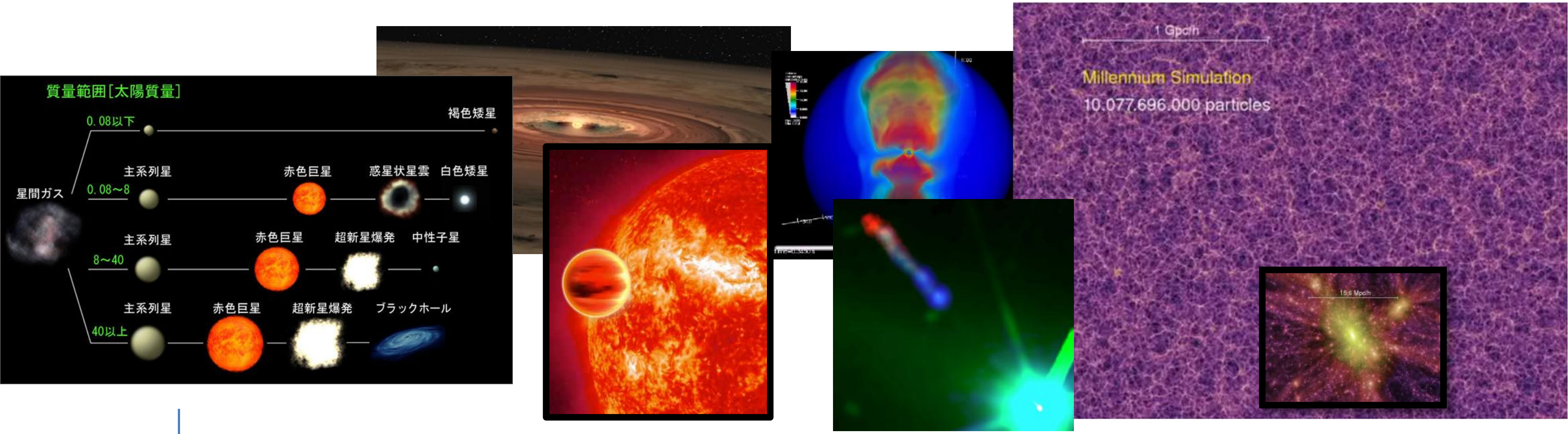
Outlines

1. Problem I want to solve & Related Projects
2. Paraiso Overview
   - 2-1. Orthotope Machine, its Formal Definition
   - 2-2. Frontend (Builder Monad)
   - 2-3. Backend (Code Generator)
3. Benchmark and Tuning Result
   - 3-1. Annotating by Hand
   - 3-2. Automated Tuning based on Genetic Algorithm

NEW

NEW

many categories of problems in astrophysics

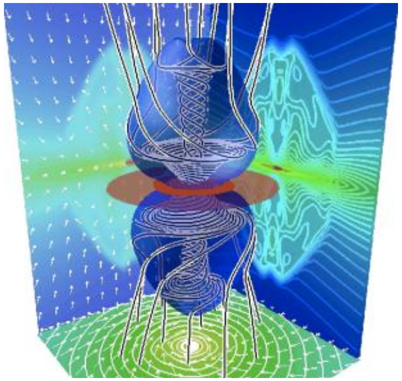


subset ∪

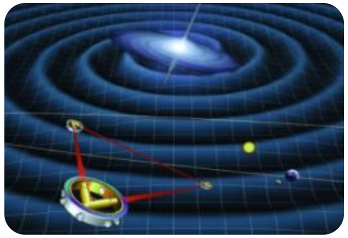
Target Problem of *Paraiso*:

Explicit Solvers of Partial Differential Equations

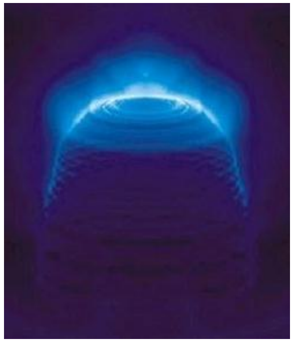
Hydrodynamics



Magneto-Hydrodynamics



General Relativity

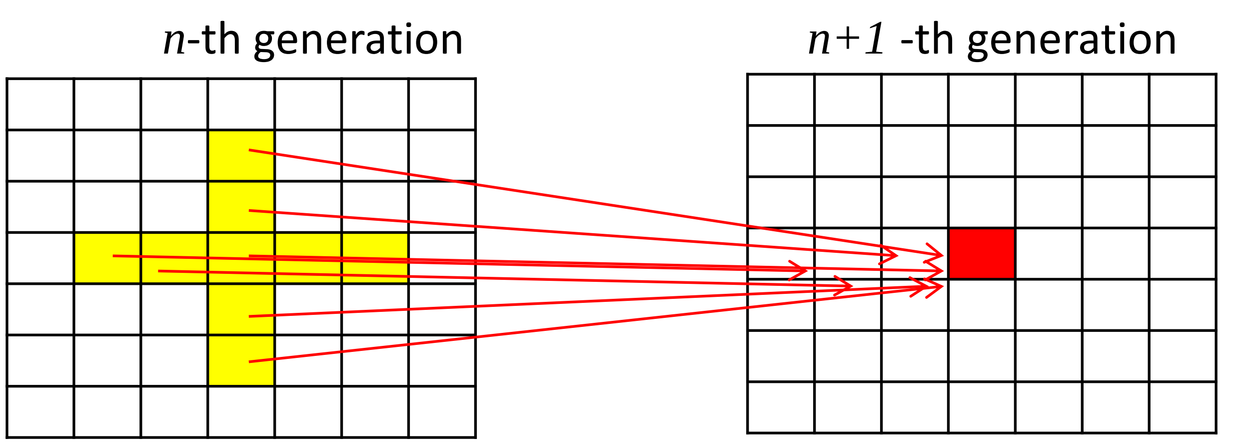


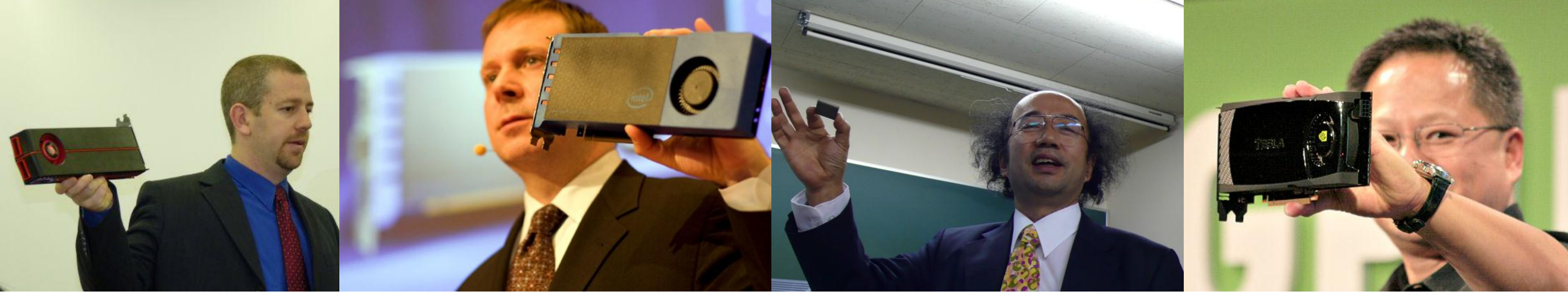
Radiative Transfer (Relativistic)

Target Problem: Partial Differential Equations, Explicit Solvers, on Uniform Mesh

From computational point of view:

- They are *d*-Dimensional, real-number cell automata. (also called *stencil calculations*)
- The state of each cell is a tuple of real numbers.
- The state of the cell at generation (n+1) is defiend as function of the states of its neighbor cells at generation (n). This locality makes distributed computation relatively easy.

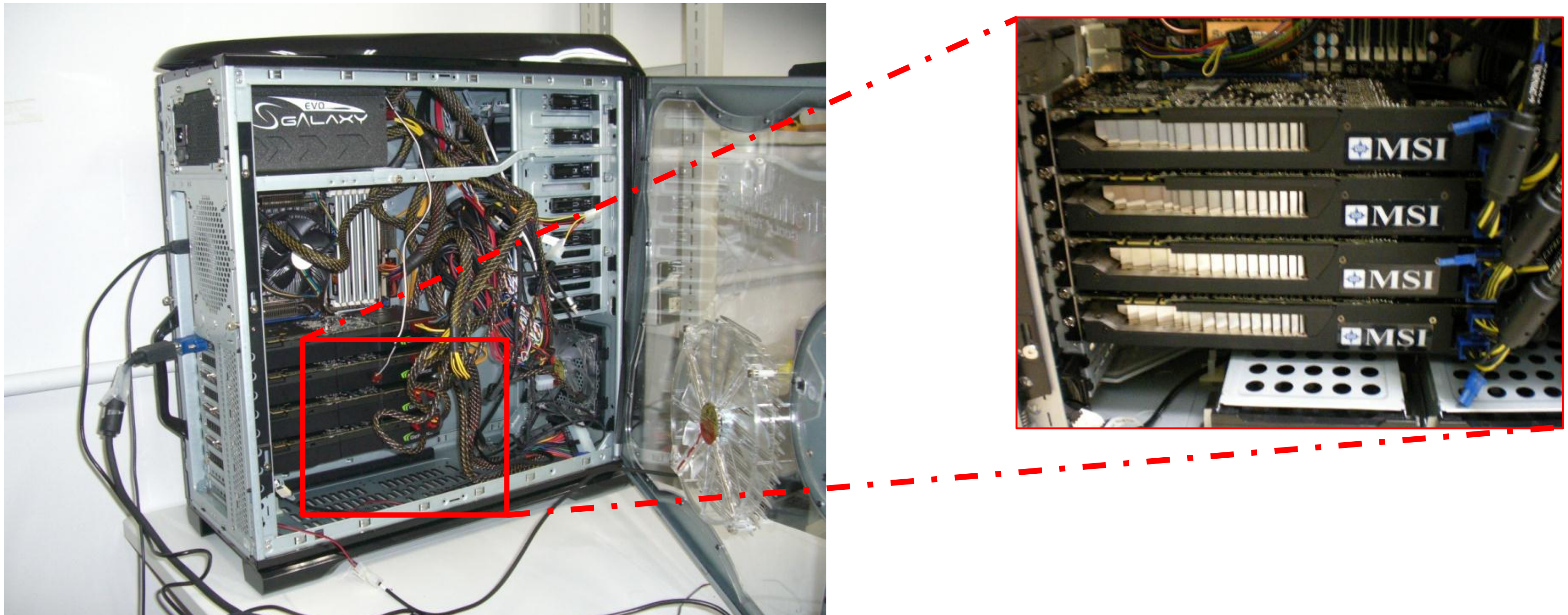




Target hardware: Parallelism!!

GPGPU: General-Purpose Computation on GPUs

M. Harris et al (2002) who coined the name



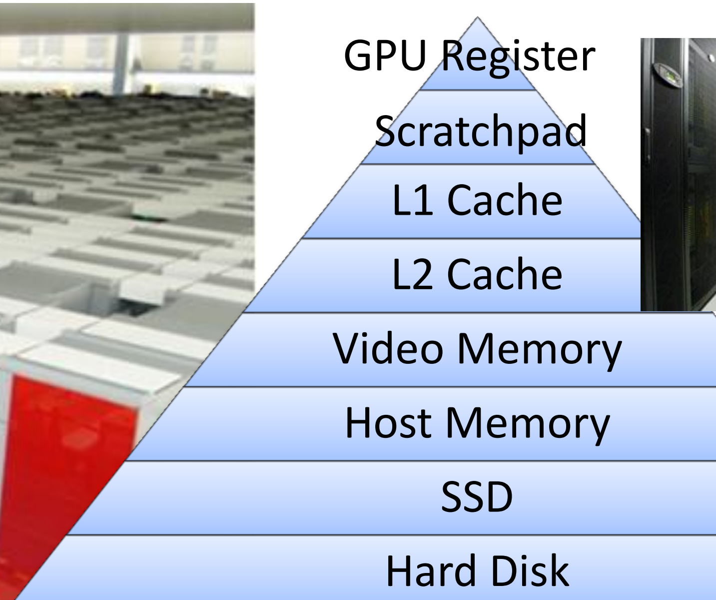
Target Machines for Paraiso

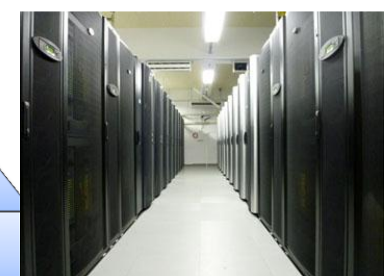
- Parallel computers (with / without accelerators like GPUs) programmed in CUDA, OpenCL or Fortran. Complex storage hierarchy
- We physicists are destined to use this kind of machines. then let's find fun ways of doing so!

4,386,816 floating operations in Parallel

The K computer, Kobe, Japan

runs 90'832'896 CUDA Thread in Parallel





Tsubame2.0 at TiTech and its ~~awful~~ hierarchy

The Problem

```
#ifdef USE_MPI

__global__ void communicate_gather_kernel_y
(int displacement_int_inc, Real displacement_real_inc, Real relative_velocity_inc,
 int displacement_int_dec, Real displacement_real_dec, Real relative_velocity_dec,

 Real *buf_inc, Real *buf_dec, Real *density, Real *velocity_x, Real *velocity_y, Real *velocity_z,
 Real *pressure, Real *magnet_x, Real *magnet_y, Real *magnet_z ) {
  const int kUnitSizeY = gSizeX * gMarginSizeY * gSizeZ;

  CUSTOM_CRYSTAL_MAP(addr, kUnitSizeY) {
    int sx, sy, sz;
    depack(addr, gSizeX, gMarginSizeY , sx, sy ,sz);
    int inc_x0 = (sx + displacement_int_inc     ) % gSizeX;
    int inc_x1 = (sx + displacement_int_inc + 1) % gSizeX;
    int dec_x0 = (sx - displacement_int_dec - 1  + gSizeX) % gSizeX;
    int dec_x1 = (sx - displacement_int_dec      + gSizeX) % gSizeX;
    Real val_inc0 = density[ enpack(gSizeX, gSizeY, inc_x0, gSizeY - 2 * gMarginSizeY + sy, sz) ];
    Real val_inc1 = density[ enpack(gSizeX, gSizeY, inc_x1, gSizeY - 2 * gMarginSizeY + sy, sz) ];
    Real val_dec0 = density[ enpack(gSizeX, gSizeY, dec_x0, gMarginSizeY + sy, sz) ];
    Real val_dec1 = density[ enpack(gSizeX, gSizeY, dec_x1, gMarginSizeY + sy, sz) ];
    buf_inc[0 * kUnitSizeY + addr] = (Real(1)-displacement_real_inc) * val_inc0 + displacement_real
_inc * val_inc0
        ;
    buf_dec[0 * kUnitSizeY + addr] = displacement_real_dec * val_dec0 + (Real(1)-displacement_real_
dec) * val_dec0
        ;
  }

  CUSTOM_CRYSTAL_MAP(addr, kUnitSizeY) {
    int sx, sy, sz;
    depack(addr, gSizeX, gMarginSizeY , sx, sy ,sz);
    int inc_x0 = (sx + displacement_int_inc     ) % gSizeX;
    int inc_x1 = (sx + displacement_int_inc + 1) % gSizeX;
    int dec_x0 = (sx - displacement_int_dec - 1  + gSizeX) % gSizeX;
    int dec_x1 = (sx - displacement_int_dec      + gSizeX) % gSizeX;
    Real val_inc0 = velocity_x[ enpack(gSizeX, gSizeY, inc_x0, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_inc1 = velocity_x[ enpack(gSizeX, gSizeY, inc_x1, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_dec0 = velocity_x[ enpack(gSizeX, gSizeY, dec_x0, gMarginSizeY + sy, sz) ];
    Real val_dec1 = velocity_x[ enpack(gSizeX, gSizeY, dec_x1, gMarginSizeY + sy, sz) ];
    buf_inc[1 * kUnitSizeY + addr] = (Real(1)-displacement_real_inc) * val_inc0 + displacement_real
_inc * val_inc0
       -relative_velocity_inc ;
    buf_dec[1 * kUnitSizeY + addr] = displacement_real_dec * val_dec0 + (Real(1)-displacement_real_
dec) * val_dec0
       +relative_velocity_dec ;
  }

  CUSTOM_CRYSTAL_MAP(addr, kUnitSizeY) {
    int sx, sy, sz;
    depack(addr, gSizeX, gMarginSizeY , sx, sy ,sz);
    int inc_x0 = (sx + displacement_int_inc     ) % gSizeX;
    int inc_x1 = (sx + displacement_int_inc + 1) % gSizeX;
    int dec_x0 = (sx - displacement_int_dec - 1  + gSizeX) % gSizeX;
    int dec_x1 = (sx - displacement_int_dec      + gSizeX) % gSizeX;
    Real val_inc0 = velocity_y[ enpack(gSizeX, gSizeY, inc_x0, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_inc1 = velocity_y[ enpack(gSizeX, gSizeY, inc_x1, gSizeY - 2 * gMarginSizeY + sy, sz)
 ];
    Real val_dec0 = velocity_y[ enpack(gSizeX, gSizeY, dec_x0, gMarginSizeY + sy, sz) ];
    Real val_dec1 = velocity_y[ enpack(gSizeX, gSizeY, dec_x1, gMarginSizeY + sy, sz) ];
    buf_inc[2 * kUnitSizeY + addr] = (Real(1)-displacement_real_inc) * val_inc0 + displacement_real
_inc * val_inc0
        ;
    buf_dec[2 * kUnitSizeY + addr] = displacement_real_dec * val_dec0 + (Real(1)-displacement_real_
dec) * val_dec0
```

- We astrophysicists write beautiful codes
- With very beautiful repeating patterns
  - I mean, as beautiful as crystalline silicate
  - OK, but this is not the kind of beauty functional programmers are searching for

Our Parallel Programming is like this

The amount of programs we write in our life is the product of the factors who multiply by copying-and-pasting.

$$N_p = N_{eq} \times N_{int} \times N_{math} \times N_{hard} \cdots$$

I want it like this

Specify each of the sufficient knowledge modules, and programs like above are automatically generated

$$N_p = N_{eq} + N_{int} + N_{math} + N_{hard} \cdots$$

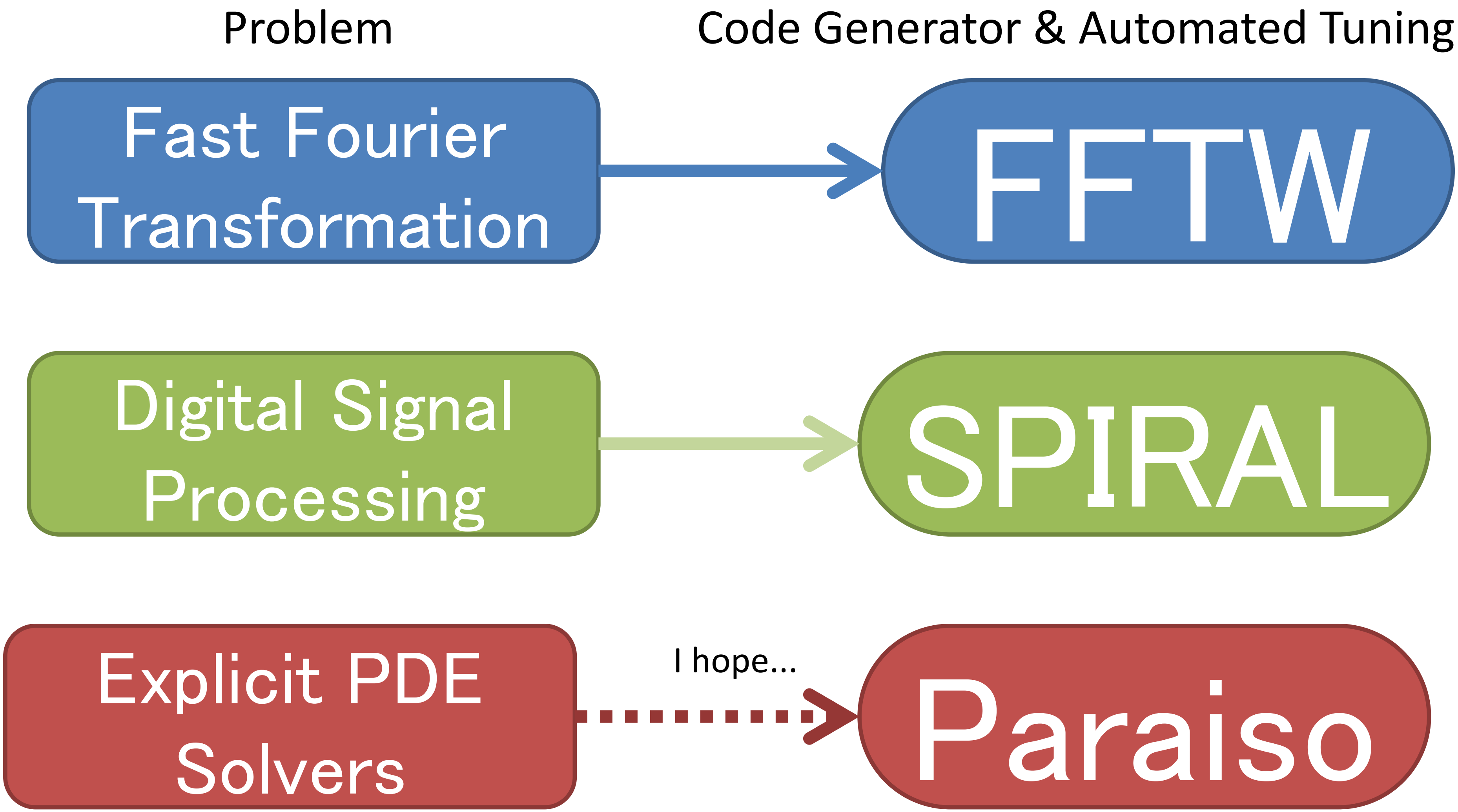
What a code generator aims for

- Generally you write $N_f \times N_{math} \times N_{eq} \times N_{int} \times N_{hw}$... lines of code
- You find a bug / improvement and want $N_{eq} = N_{eq} + 1$; then you need to re-write $N_f \times N_{math} \times 1 \times N_{int} \times N_{hw}$... lines

- With code generator you only have to write

  $N_f + N_{math} + N_{eq} + N_{int} + N_{hw}$... lines
- You want $N_{eq} = N_{eq} + 1$; then just add 1 line
- *You can concentrate on physics*
- *We have vast possibility for automated tuning*

related projects



related projects

repa-2.2.0.1: High performance, regular, shape polymorphic parallel arrays. | hackageDB | Style

The repa package

Repa provides high performance, regular, multi-dimensional, shape polymorphic parallel arrays. All numeric data is stored unboxed. Functions written with the Repa combinators are automatically parallel provided you supply +RTS -Nwhatever on the command line when running the program.

accelerate-0.8.1.0: An embedded language for accelerated array processing | hackageDB | Style

The accelerate package

This library defines an embedded language for regular, multi-dimensional array computations with multiple backends to facilitate high-performance implementations. Currently, there are two backends: (1) an interpreter that serves as a reference implementation of the intended semantics of the language and (2) a CUDA backend generating code for CUDA-capable NVIDIA GPUs.

Nikola: Embedding Compiled GPU Functions in Haskell

Geoffrey Mainland and Greg Morrisett
Harvard School of Engineering and Applied Sciences
{mainland,greg}@eecs.harvard.edu

Paraiso

- **cannot** invent new integration schemes for you

- offers tensor notations and algorithm transformers, to avoid repeating yourself.
- can generate programs instead of you
  - for CPUs, GPUs, and future machines ...
- can search for better memory & cache usage pattern for you
- (can search for better communication patterns for you)

Overall design

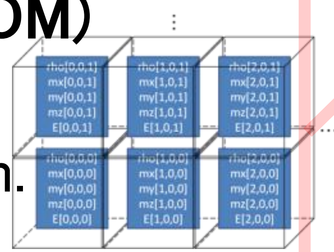
equation
you want to solve

$$\frac{\partial f}{\partial t} = g,$$
$$\frac{\partial g}{\partial t} = c^2 \frac{\partial^2 f}{\partial x^2},$$

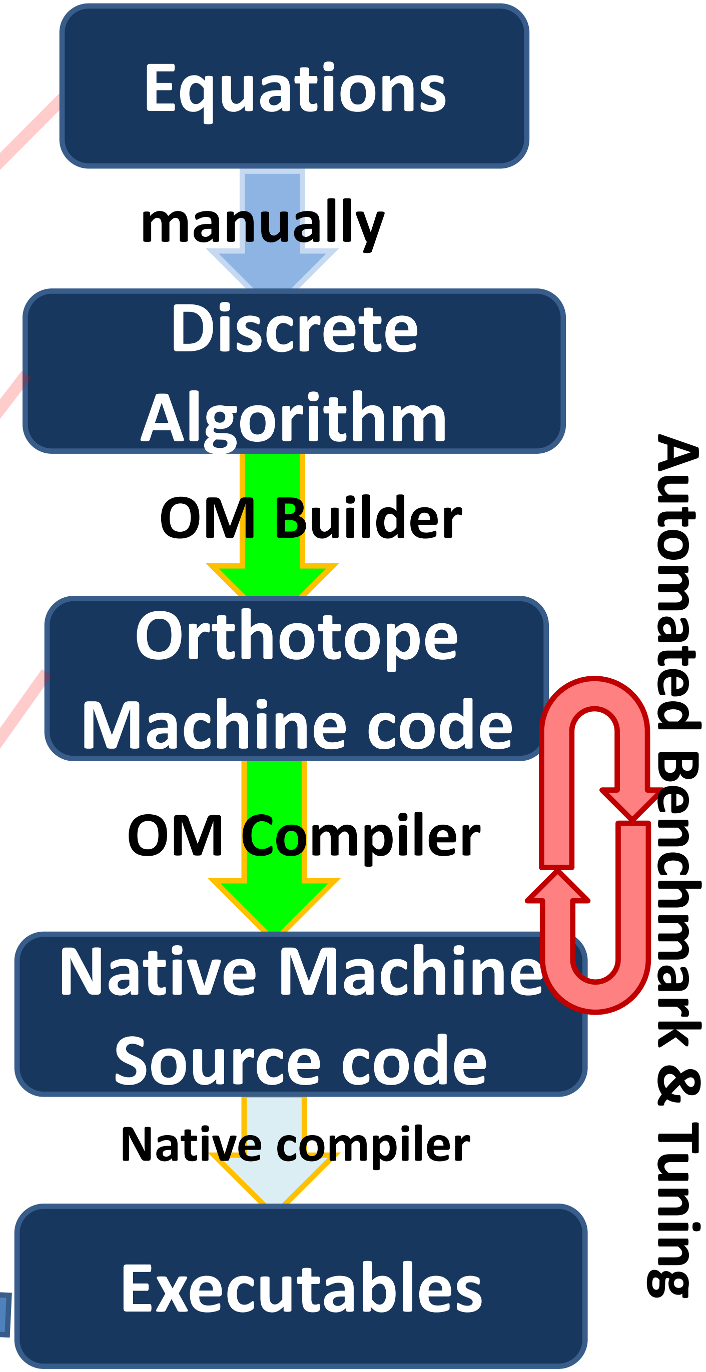
solver algorithm described in simple mathematical notation

$$f^{n+1}[\mathrm{i}] = f^n[\mathrm{i}] + \Delta t\, g^n[\mathrm{i}],$$
$$g^{n+1}[\mathrm{i}] = g^n[\mathrm{i}] + \frac{c^2\,\Delta t}{\Delta x^2}\left(f^{n+1}[\mathrm{i}+1] + f^{n+1}[\mathrm{i}-1] - 2f^{n+1}[\mathrm{i}]\right),$$

Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays



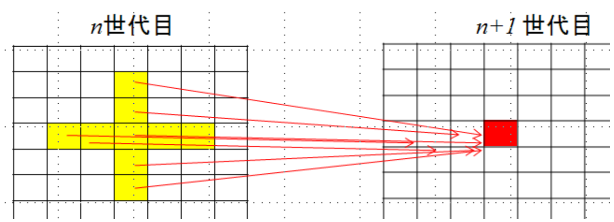
result



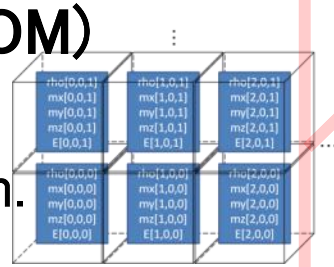
Orthotope Machine

equation
you want to solve $\frac{\partial U}{\partial t} + \nabla \cdot \boldsymbol{F} = 0$

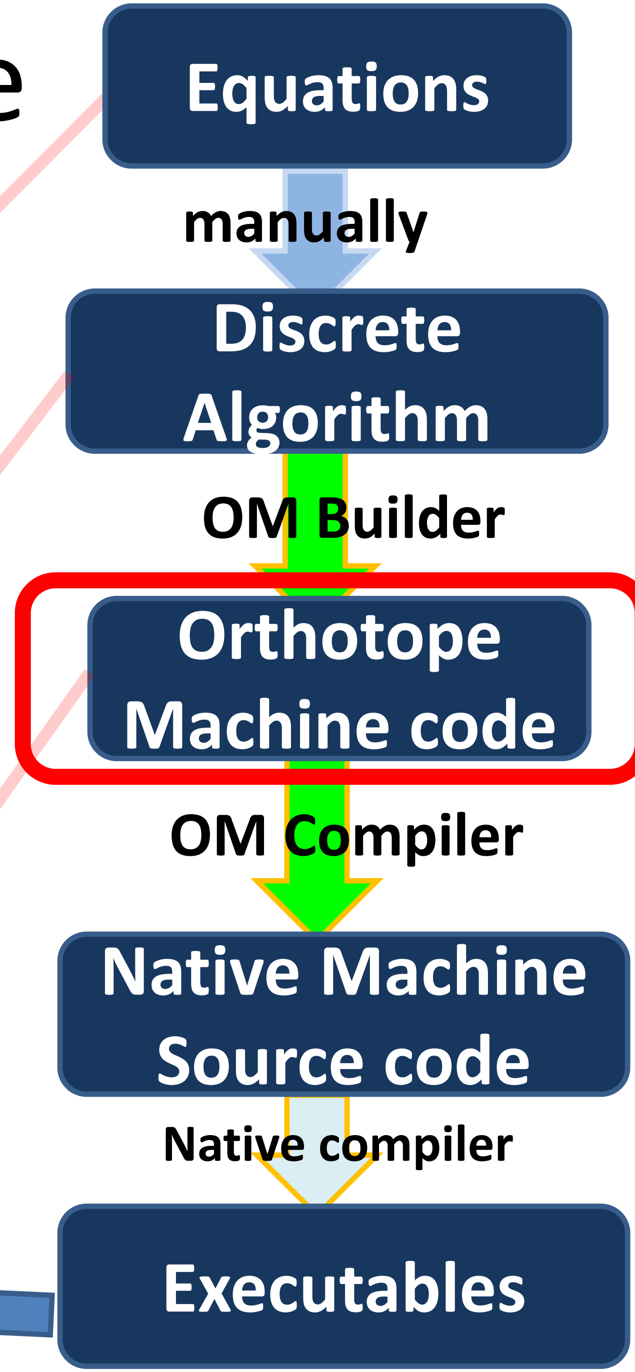
solution algorithm described in
OM Builder Monad



Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays



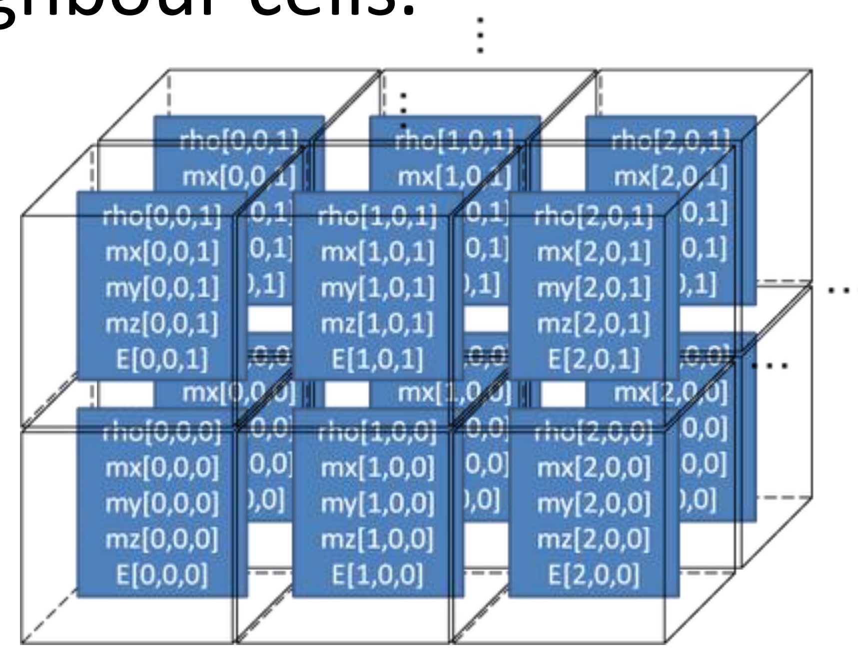
result



Orthotope Machine (OM)

- A virtual machine much like vector computers, each register is multidimensional array of infinite size
- arithmetic operations work in parallel on each mesh, or loads from neighbour cells.

No intention of buiding a real hardware:
a thought object to construct a dataflow graph



Instruction set of Orthotope Machine

and as a physicist I can assure this tiny set can cover any hyperbolic PDE solving algorithm (for uniform mesh)

```haskell
data Inst vector gauge
  = Imm Dynamic
  | Load Name
  | Store Name
  | Reduce R.Operator
  | Broadcast
  | Shift (vector gauge)
  | LoadIndex (Axis vector)
  | Arith A.Operator
instance Arity (Inst vector gauge) where
  arity a = case a of
    Imm _       -> (0,1)
    Load _      -> (0,1)
    Store _     -> (1,0)
    Reduce _    -> (1,1)
    Broadcast   -> (1,1)
    Shift _     -> (1,1)
    LoadIndex _   -> (0,1)
    Arith op    -> arity op
```

`Imm`
load constant value

`Load (graph starts here)`
read from named array

`Store (graph ends here)`
write to named array

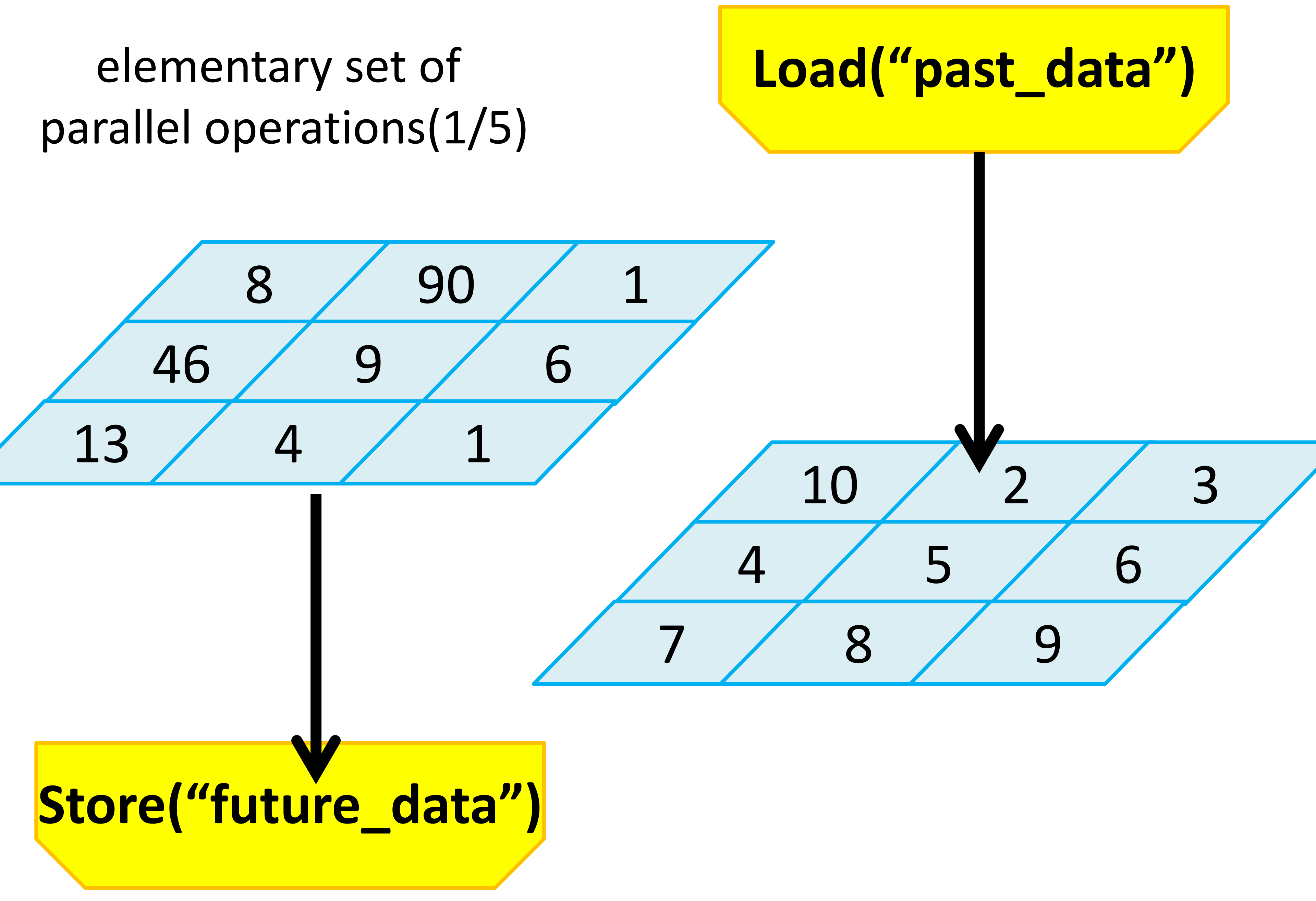
`Reduce`
array to scalar value

`Broadcast`
scalar to array

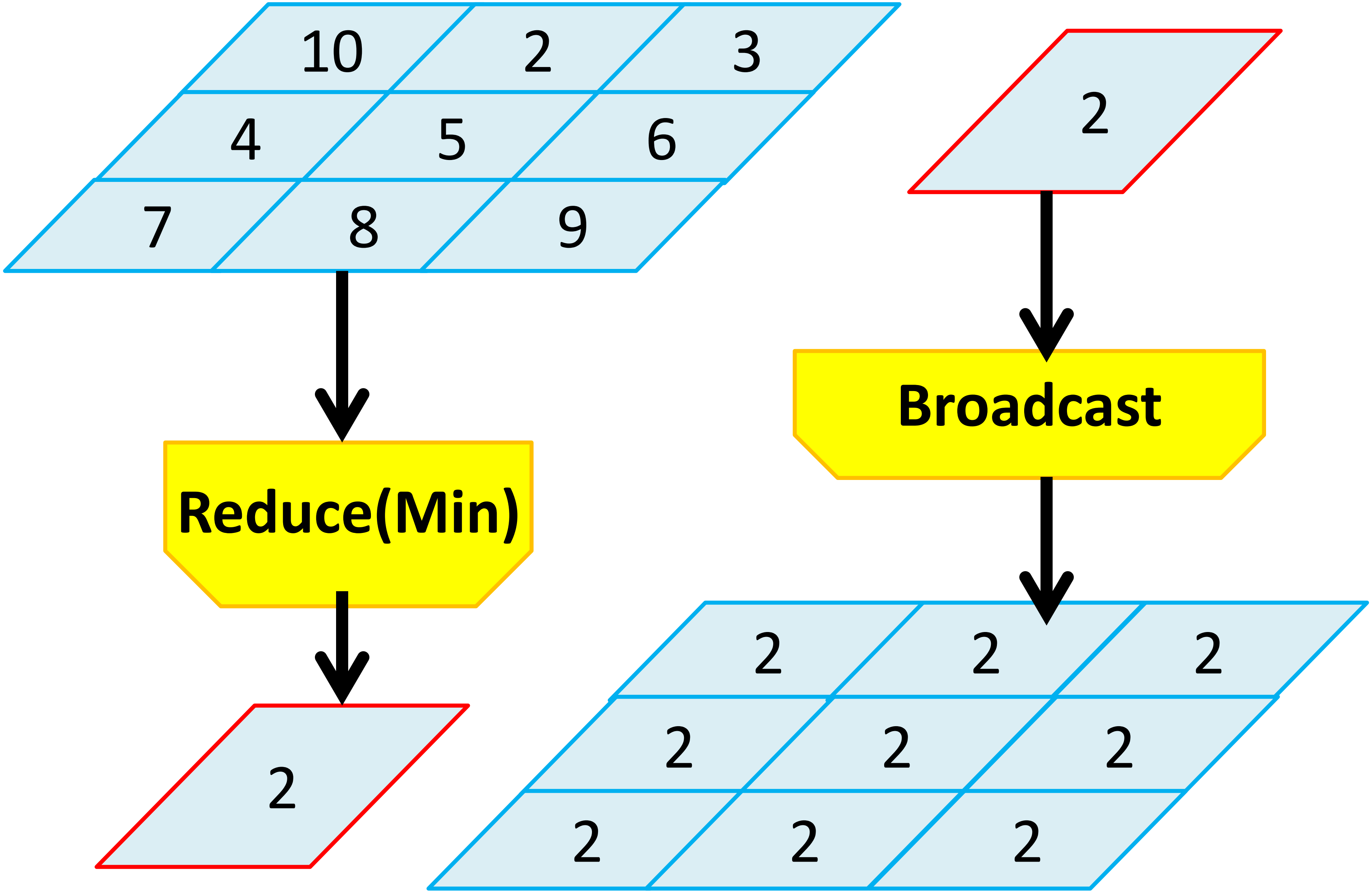
`Shift`
copy each cell to neighbourhood

`LoadIndex & LoadSize`
get coordinate of each cell
get array size

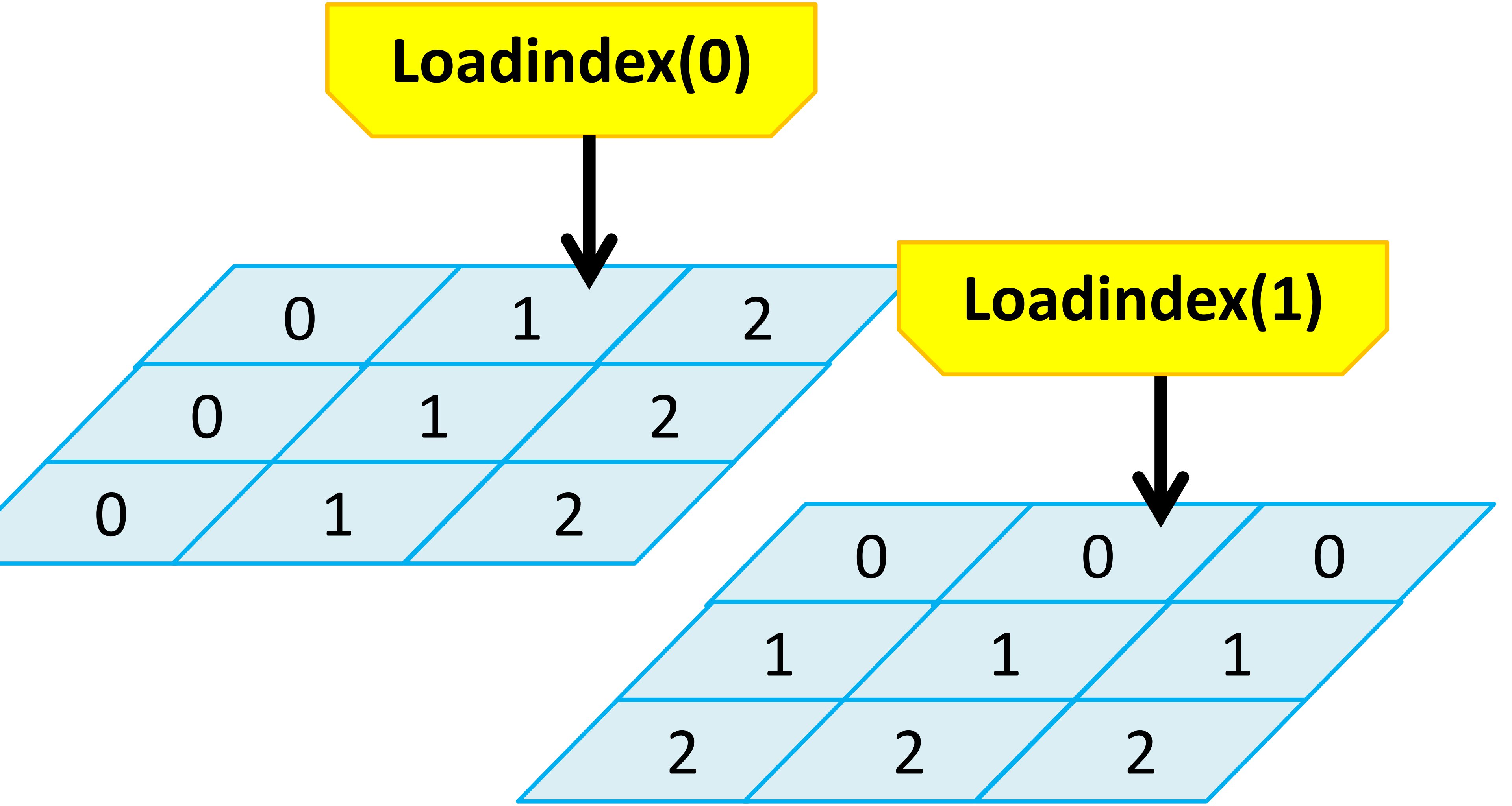
`Arith`
various mathematical operations


elementary set of
parallel operations(1/5)
Load("past_data")
8
90
1
46
9
6
13
4
1
10
2
3
4
5
6
7
8
9
Store("future_data")

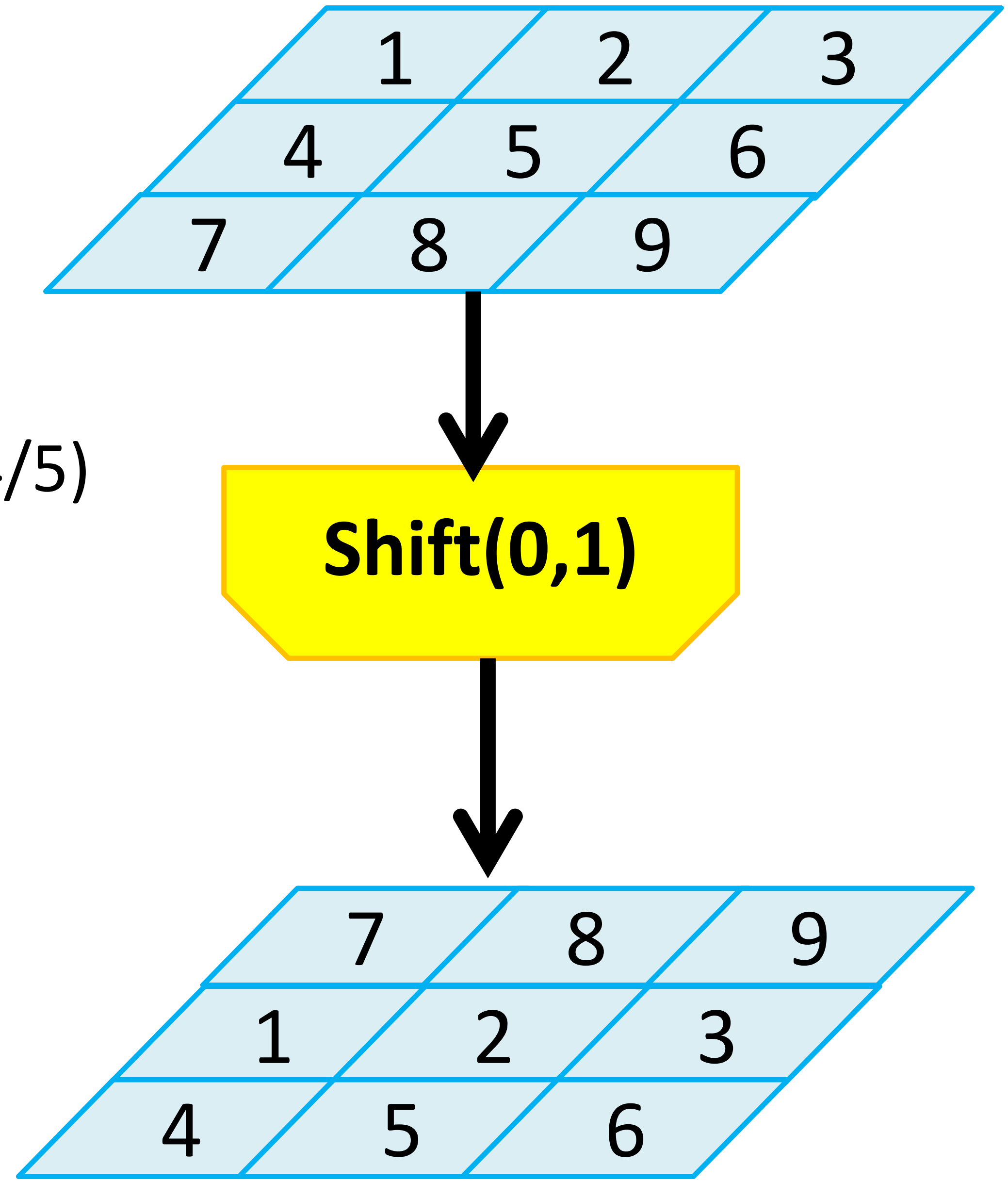
elementary set of parallel operations(2/5)



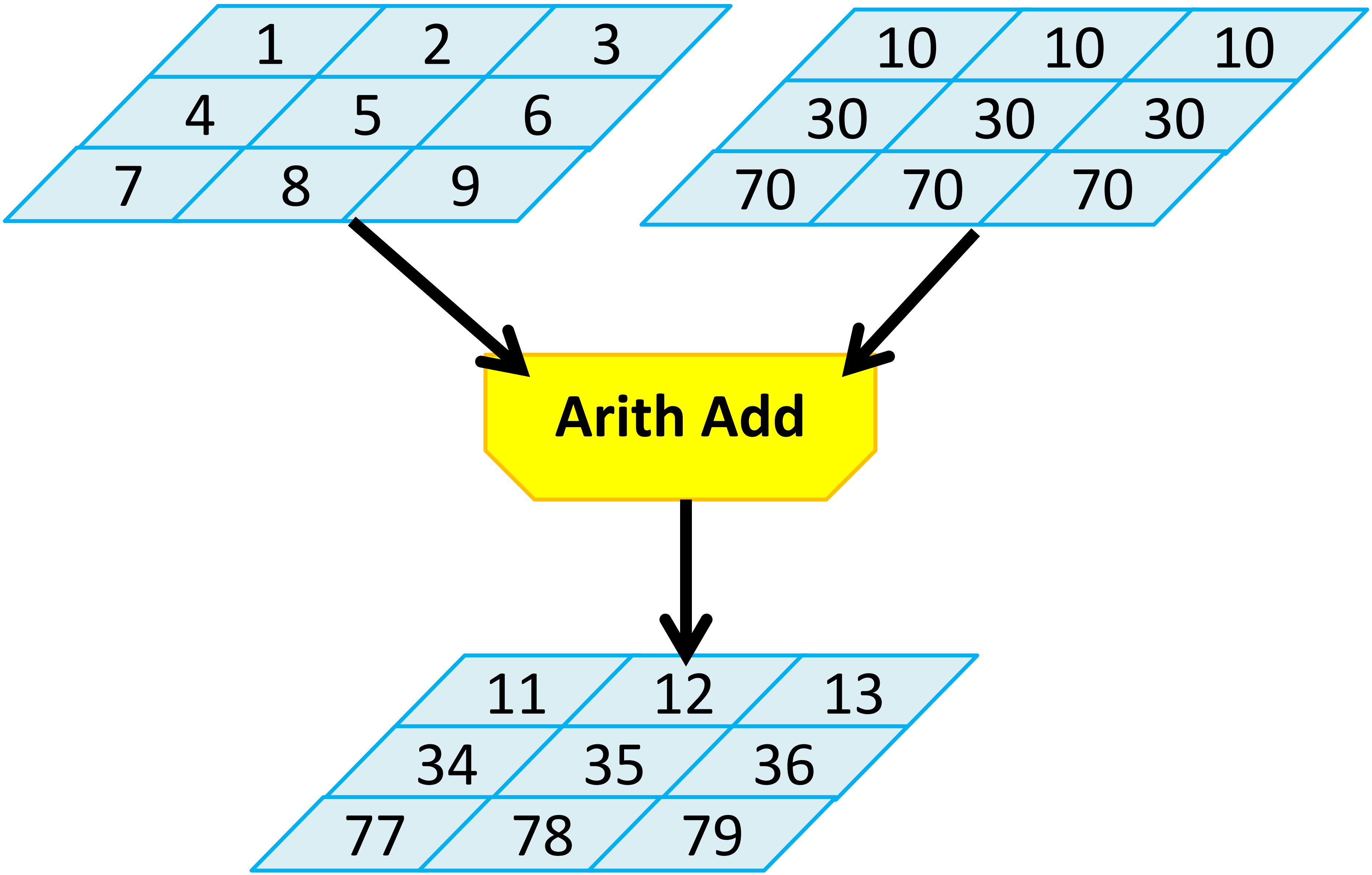
elementary set of parallel operations(3/5)



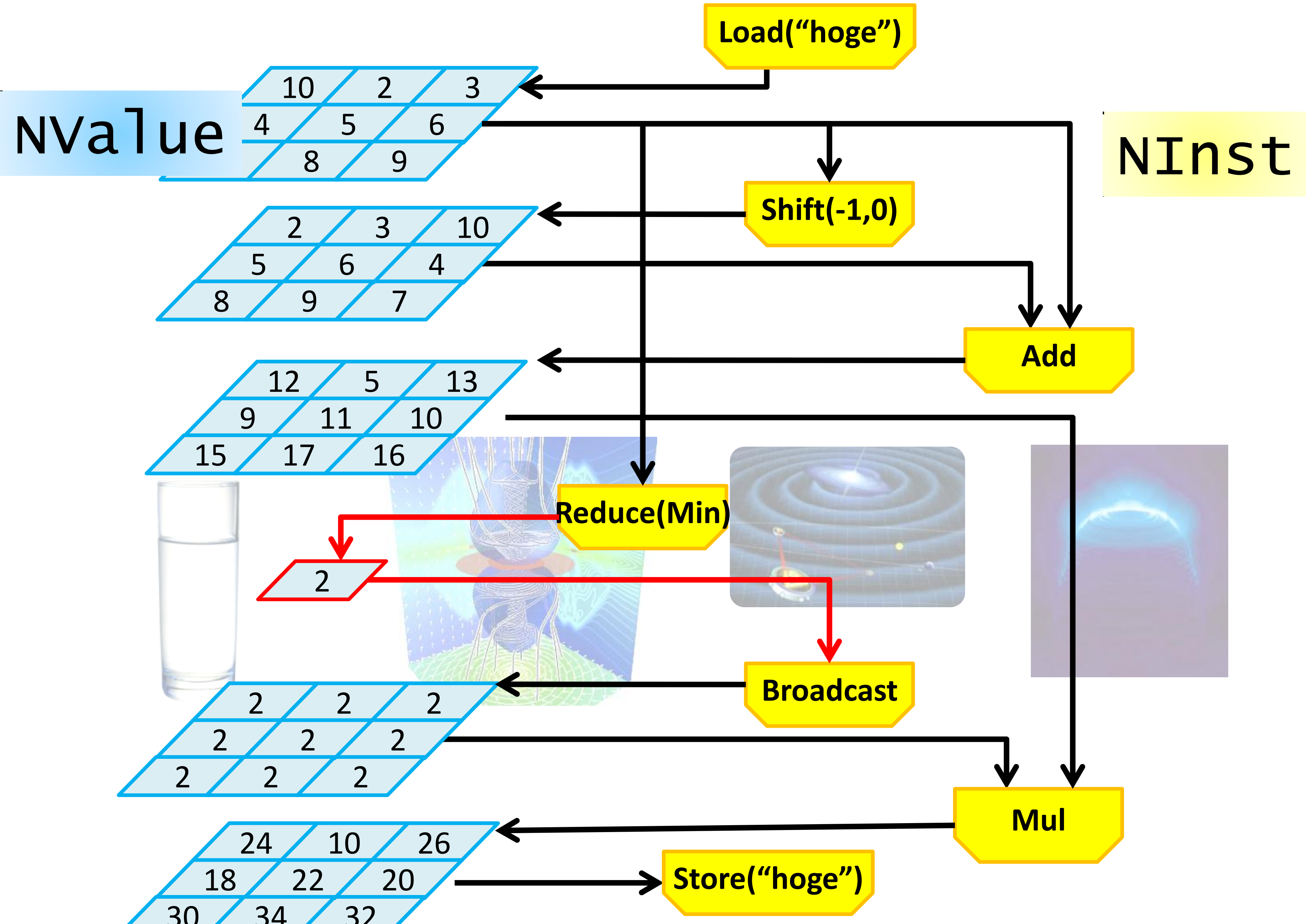
elementary set of
parallel operations(4/5)



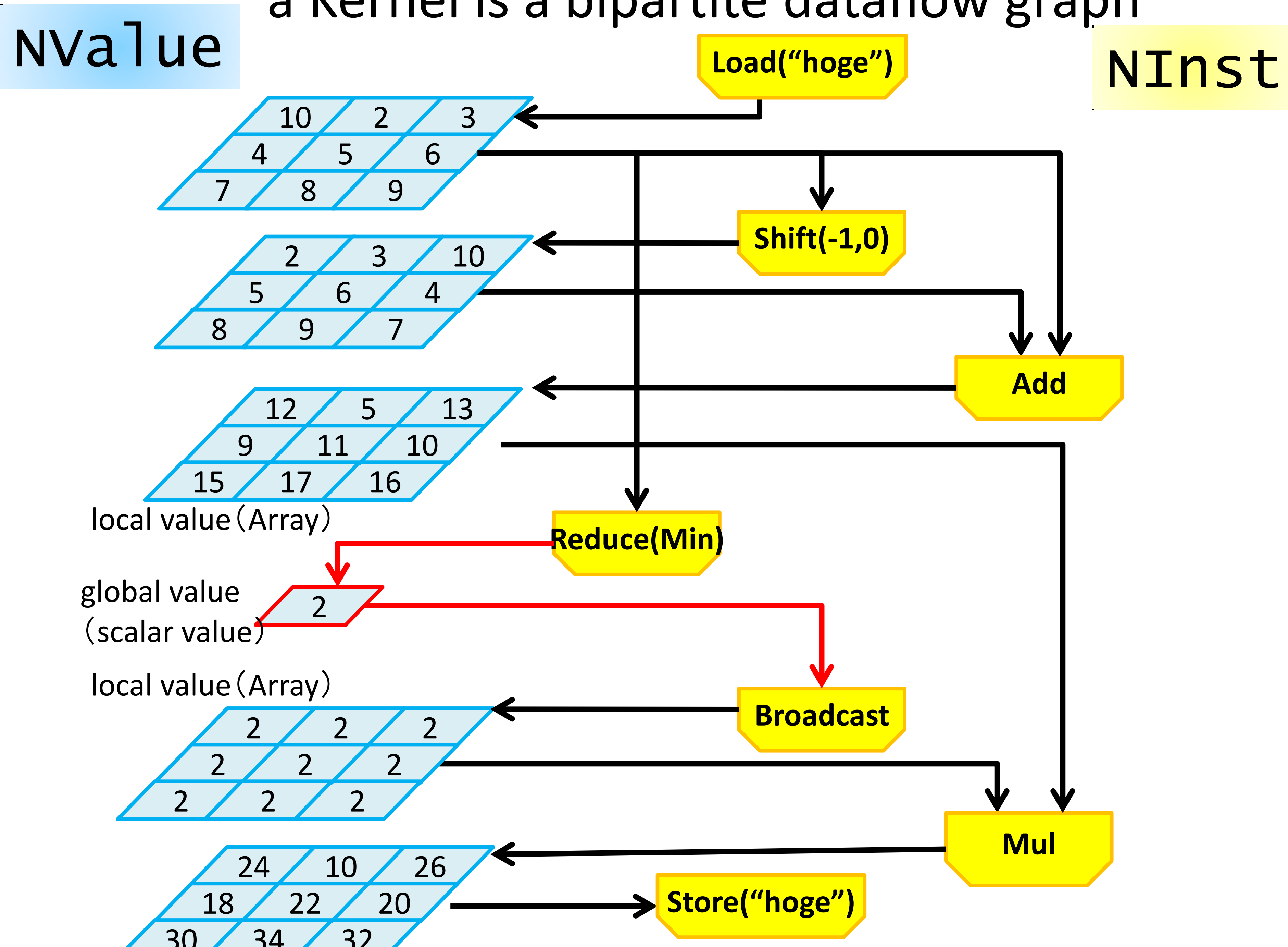
elementary set of parallel operations(5/5)



Any (mesh) Program is composed of these elements

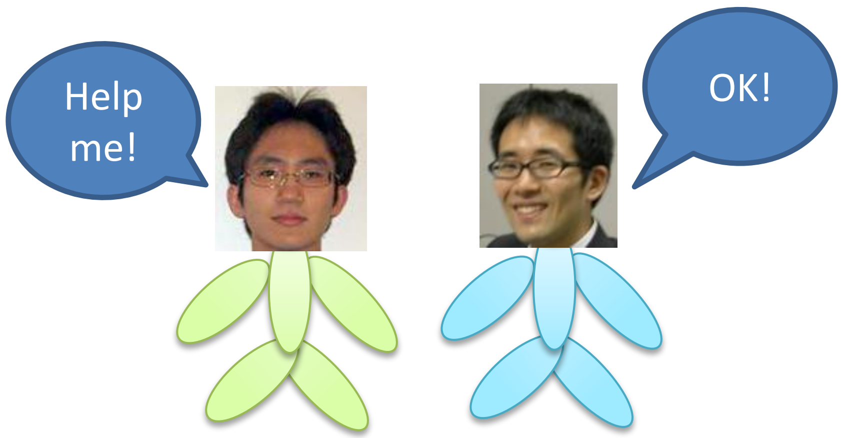


a Kernel is a bipartite dataflow graph



Formal definition of Orthotope Machine Semantics

- Thanks to Kohei Suenaga, the 3rd batch Hakubi Member



- math ahead warning

goal

$$
\begin{aligned}
(\emptyset, \texttt{Imm}\ a, \{y\}) \in C \quad &\Rightarrow \quad E_T(y) = \lambda i.a && (1)\\
(\emptyset, \texttt{Load}\ s, \{y\}) \in C \quad &\Rightarrow \quad E_T(y) = E_S(s) && (2)\\
(\{x\}, \texttt{Store}\ s, \emptyset) \in C \quad &\Rightarrow \quad E'_S(s) = E_T(x) && (3)\\
(\{x\}, \texttt{Reduce}\ r, \{y\}) \in C \quad &\Rightarrow \quad E_T(y) = \lambda i.r(E_T(x)) && (4)\\
(\{x\}, \texttt{Broadcast}, \{y\}) \in C \quad &\Rightarrow \quad E_T(y) = E_T(x) && (5)\\
(\{x\}, \texttt{Shift}\ i', \{y\}) \in C \quad &\Rightarrow \quad E_T(y) = \lambda i.E_T(x)(i + i') && (6)\\
(\emptyset, \texttt{LoadIndex}\ ax, \{y\}) \in C \quad &\Rightarrow \quad E_T(y) = \lambda i.i!ax && (7)
\end{aligned}
$$

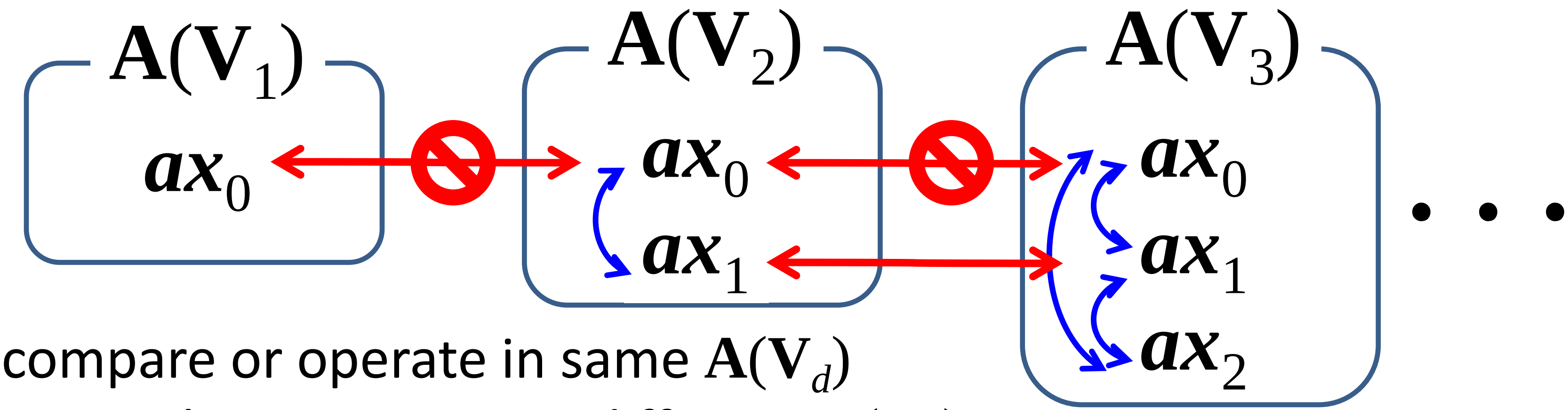
$$
\begin{aligned}
&(\{x_0, \cdots, x_{m-1}\}, \texttt{arith}\ op, \{y_0, \cdots, y_{n-1}\}) \in C\\
&\quad \Rightarrow E_T(y_\beta) = \lambda i.op_\beta(E_T(x_0)(i), \cdots, E_T(x_{m-1})(i)) \qquad (8)
\end{aligned}
$$

Def. An *d-dimensional vector of gauge* $\mathbb{G}$, denoted by $\mathbb{V}_d(\mathbb{G})$, is basically a d-tuple of $\mathbb{G}$ with vector arithmetic defined.

much like a C++ template programming
when you say
`vector2<int>` or
`vector3<double>`

Def. An *d-dimensional vector of gauge* $\mathbb{G}$, denoted by $\mathbb{V}_d(\mathbb{G})$, is basically a d-tuple of $\mathbb{G}$ with vector arithmetic defined.

Def. $\mathbb{A}(\mathbb{V}_d)$ is the *axis space* for vector type-constructor $\mathbb{V}_d$, and consists of d elements; $\mathbb{A}(\mathbb{V}_d) = \{ax_0, \cdots, ax_{d-1}\}$.



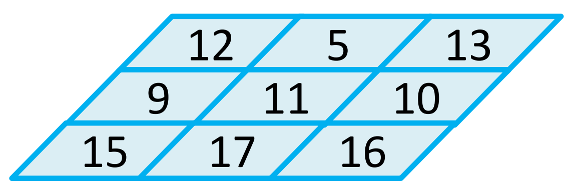
can compare or operate in same $\mathbf{A}(\mathbf{V}_d)$
but cannot between axes in different $\mathbf{A}(\mathbf{V}_d)$

Def. ! is the component access operator. For the pair $v \in \mathbb{V}_{d'}(\mathbb{G})$ and $ax_\alpha \in \mathbb{A}(\mathbb{V}_d)$, $v!ax$ is defined as the α-th component of the vector v if $d = d'$. $v!ax$ is not defined if $d \neq d'$.

$$! :: \quad \mathbf{V}_d(\mathbf{G}) \longrightarrow \mathbf{A}(\mathbf{V}_d) \longrightarrow \mathbf{G}$$

`Vec :~2 :~3 :~5`   `Axis 1`   `3`

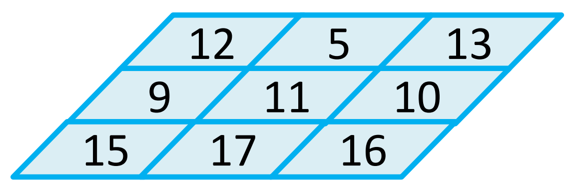
Def. An *orthotope value* of dimension d, gauge $\mathbb{G}$ and element type $\mathbb{E}$ is a function of type $\mathbb{V}_d(\mathbb{G}) \to \mathbb{E}$. The domain of the orthotope value $\mathbb{V}_d(\mathbb{G})$ is called the index space, or simply the index of the orthotope. It can be regarded as d-dimensional array of infinite size with elements of type $\mathbb{E}$.



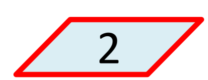
an orthotope value（Array）

Def. The *realm* of an orthotope value is either *global* or *local*. The realm of an orthotope value x is global iff. for all pair of index (i, j) it satisfies $x(i) = x(j)$. The orthotope value is local otherwise.

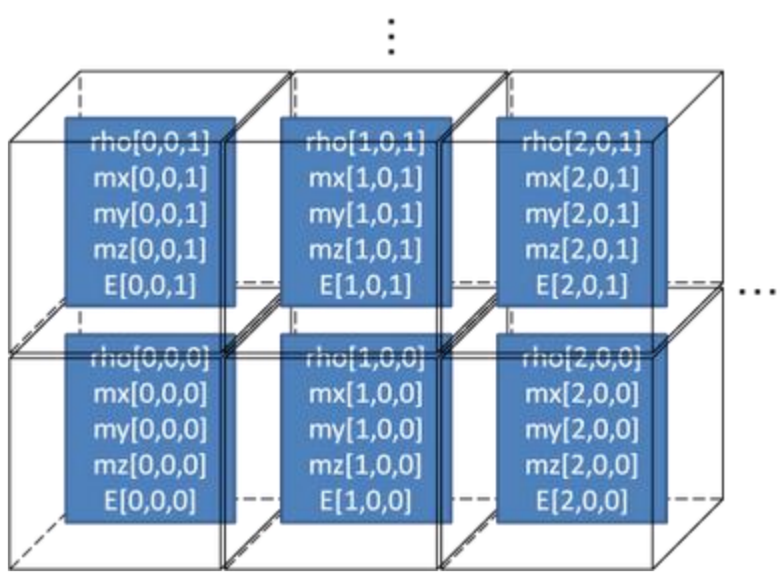
local value（Array）

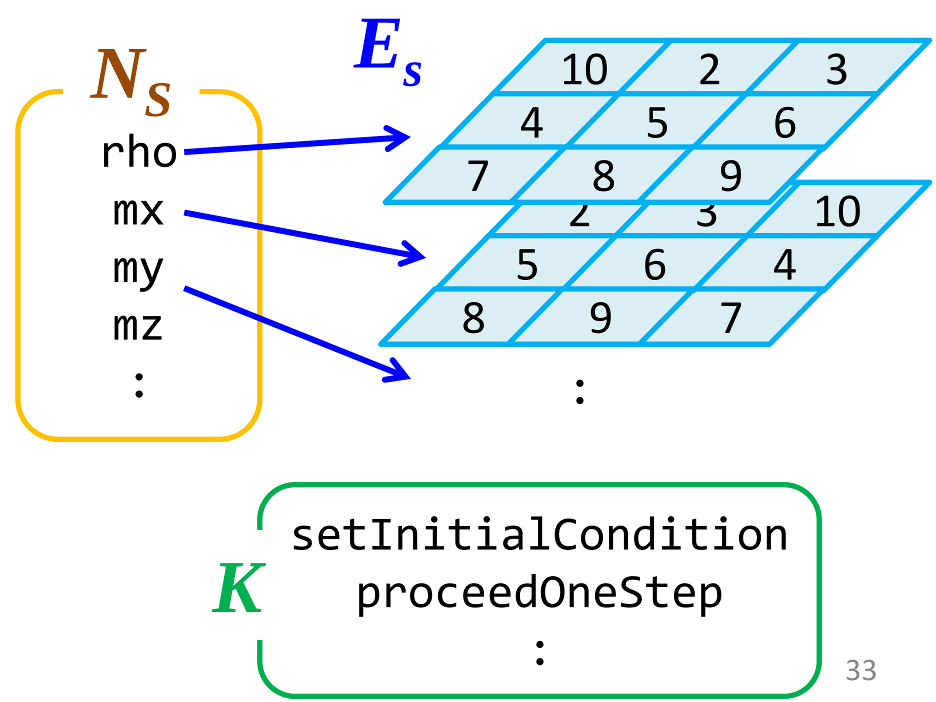


global value
（effectively a scalar value）

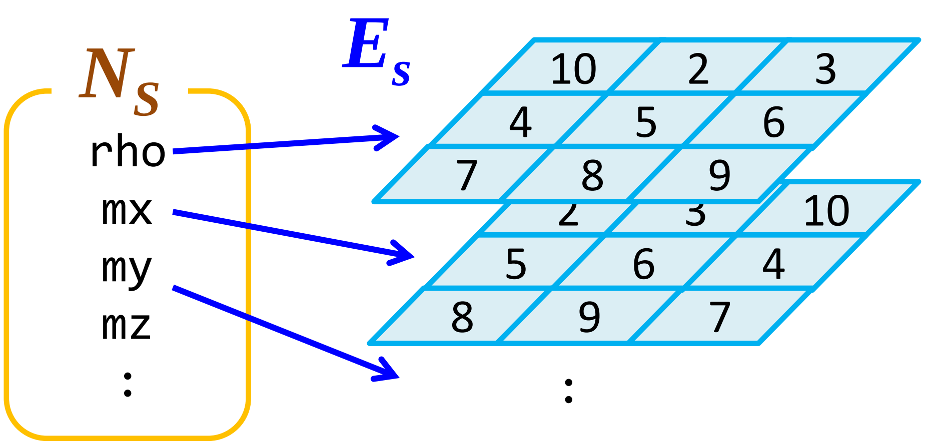


Def. An *orthotope machine* of dimension d, gauge $\mathbb{G}$ is a tuple (N_S, E_S, K) where N_S is the set of the static value names, E_S is the static value environment, and K is the set of kernels of the machine.

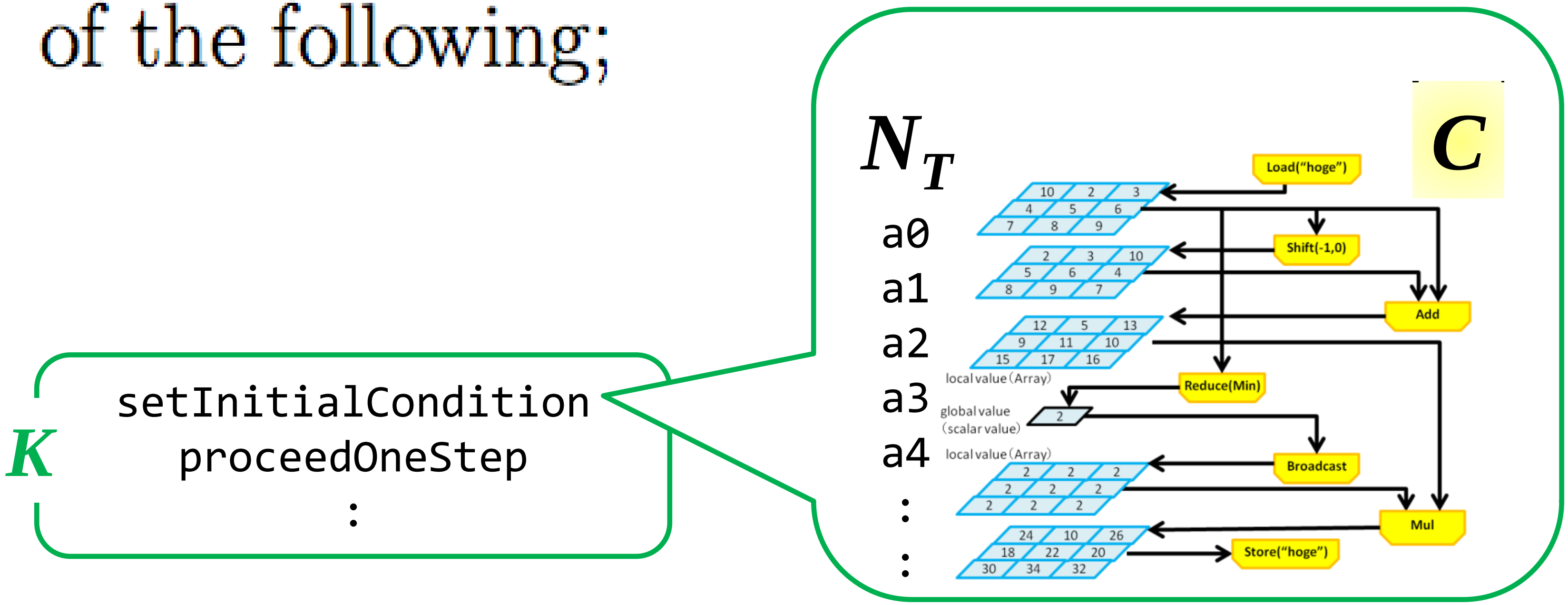




N_S is just a set of some identifiers (e.g. a set of strings). E_S is the function from N_S to orthotope value: for $s \in N_S$, $E_S(s)$ is of type $\mathbb{V}_d(\mathbb{G}) \to \mathbb{E}_s$ where $\mathbb{E}_s$ is the element type of static value s. The dimensions d and gauges $\mathbb{G}$ of all the orthotope values are the same as those of the orthotope machine itself.



A kernel $k \in K$ is a pair (N_T, C) where N_T is the set of temporal value names, and C is the set of commands. A command $c \in C$ is a triple $(xs, inst, ys)$. The two $xs, ys \subset N_T$ are the domain and the codomain of the command, and $inst$ is one of the following;

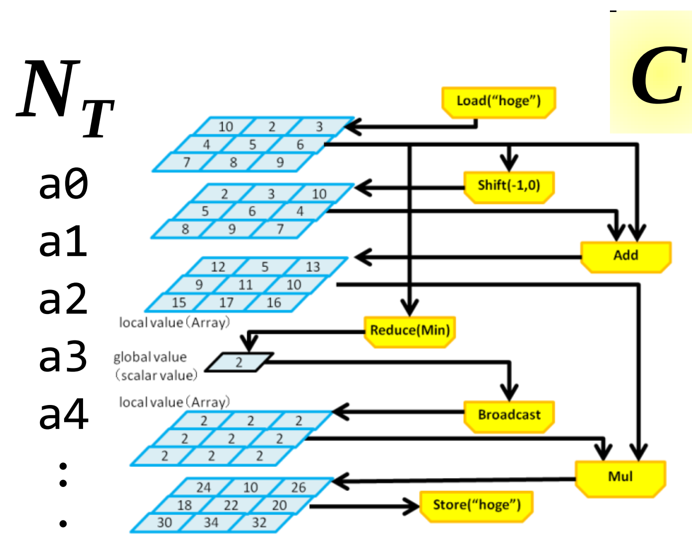


instruction	arity	constraint
`Imm` a	(0,1)	a is some value of an element type.
`Load` s	(0,1)	$s \in N_S$.
`Store` s	(1,0)	$s \in N_S$.
`Reduce` r	(1,1)	$r \in (\mathbb{V}_d(\mathbb{G}) \to \mathbb{E}) \to \mathbb{E}$ is a reduction operator for a certain type $\mathbb{E}$.
`Broadcast`	(1,1)	
`Shift` i'	(1,1)	$s \in \mathbb{V}_d(\mathbb{G})$.
`LoadIndex` ax	(0,1)	$ax \in \mathbb{A}(\mathbb{V}_d)$.
`arith` op	(m, n)	op is a function of arity (m, n).

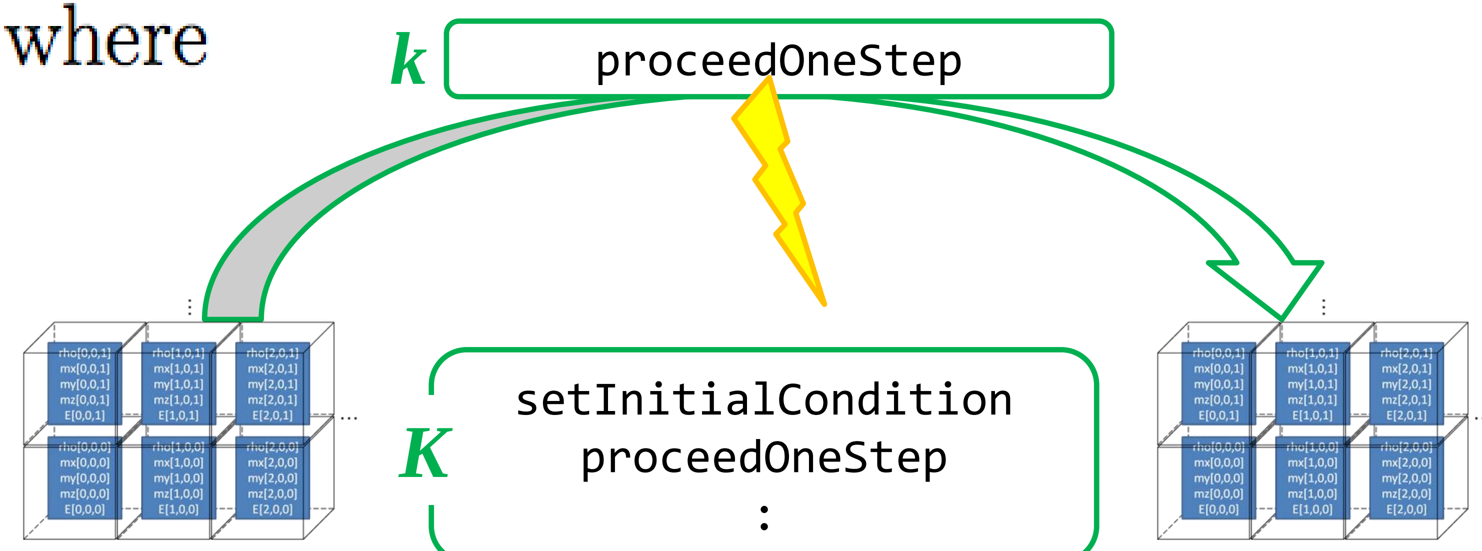
Each command must comply with the aritiy of its $inst$, i.e. $(xs, inst, ys) \in C \Rightarrow (|xs|, |ys|) = \text{arity}(inst)$.

Def. The *dataflow graph* of a kernel (N_T, C) is a directed graph (V, E) where the vertices $V = N_T \oplus C$, and the edges E are defined as follows:

$$
\begin{aligned}
(a, b) \in E \Leftrightarrow \\
\exists c \in C,\ c = (xs, inst, ys) \text{ s.t.} \\
(a \in xs \wedge b = c) \\
\vee\, (a = c \wedge b \in ys)
\end{aligned}
$$



Def. An *execution* of a kernel. The state of an orthotope machine, E_S, is updated by executing its kernels, one at a time. The initial state is undefined everywhere; $E_S(s)(i) = \perp$ for all $s \in N_S, i \in \mathbb{V}_d(\mathbb{G})$. The state of the machine after the execution of the kernel k is $NextGen(k, E_S)$, where



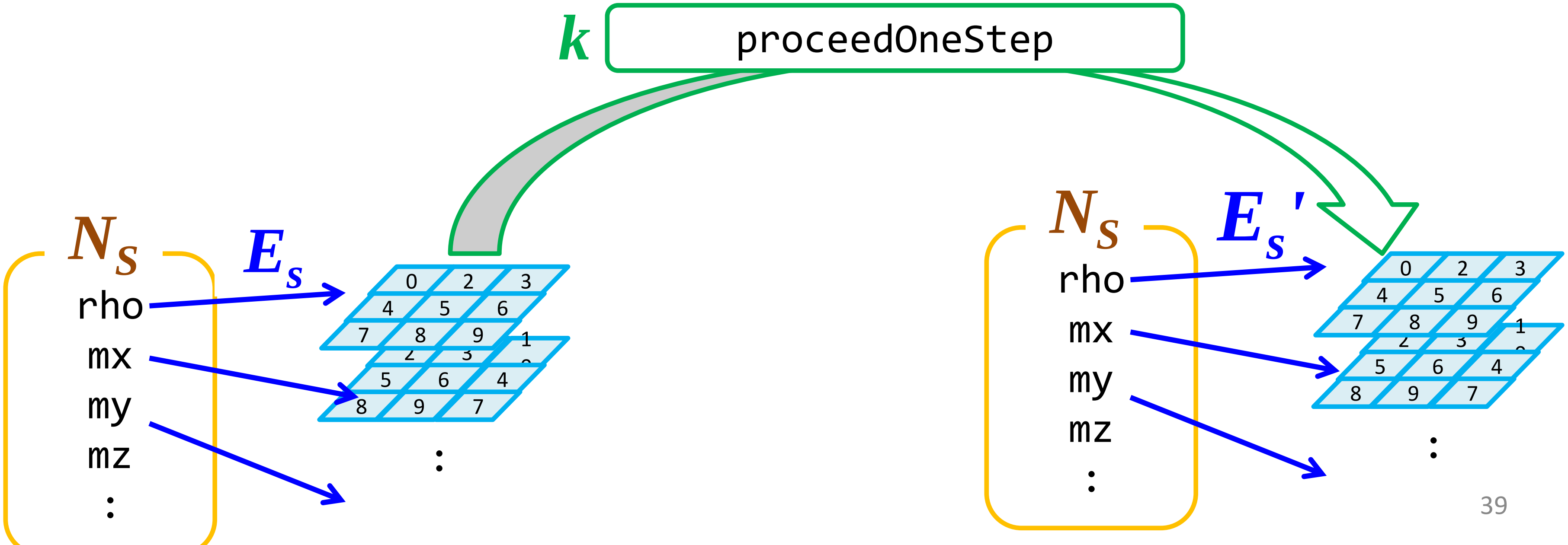
$$NextGen(k, E_S) = E'_S \overset{def}{\iff}$$

$$\exists E_T . Feasible((k, E_S), (E_T, E''_S))$$

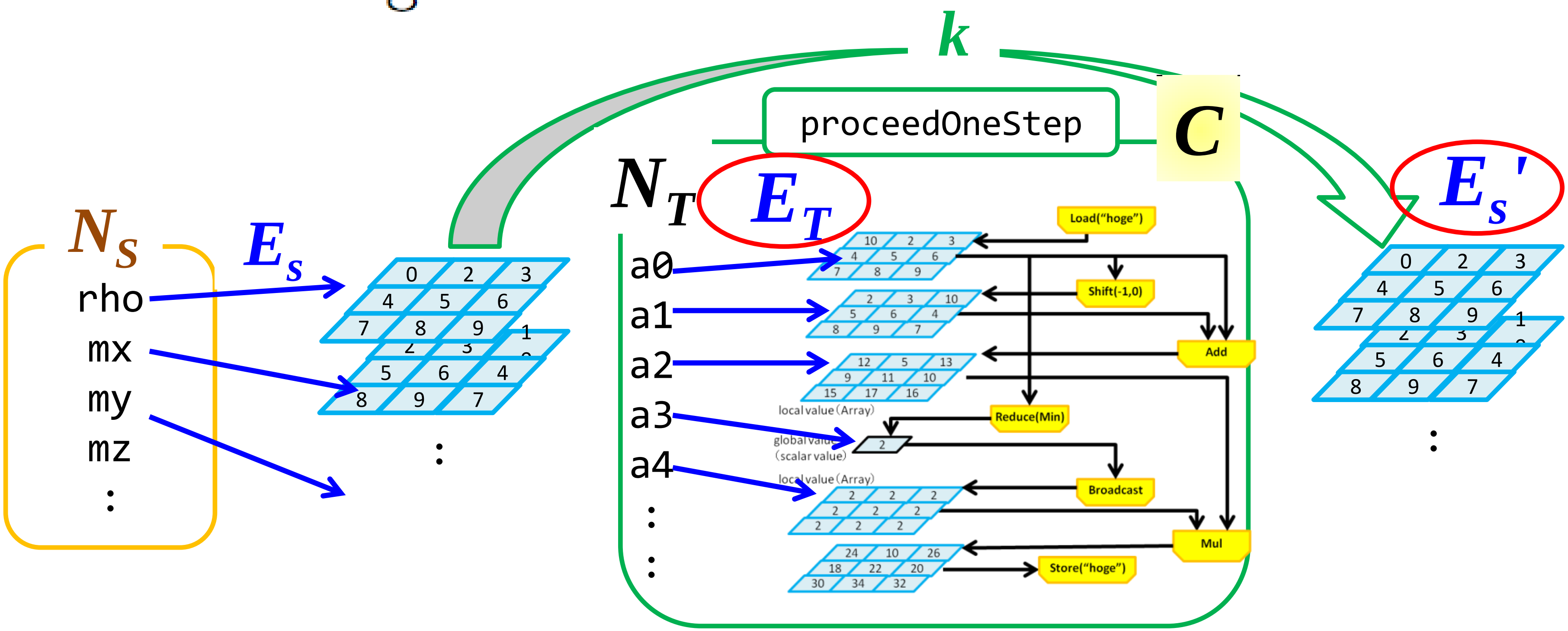
and

$$E'_S(s) = \begin{cases} E''_S(s) & \text{if } s \in \mathrm{dom}(E''_S) \\ E_S(s) & \text{otherwise} \end{cases}$$

where $s \in N_S$.



Def. The *feasibility* of an execution. Given a kernel $k = (N_T, C)$ and an environment E_S of an orthotope machine, we say $Feasible((k, E_S), (E_T, E'_S))$ iff. (E_T, E'_S) is the least predecessor that satisfies the following conditions.



$$(\emptyset, \texttt{Imm}\ a, \{y\}) \in C \Rightarrow E_T(y) = \lambda i.a \quad (1)$$

$$(\emptyset, \texttt{Load}\ s, \{y\}) \in C \Rightarrow E_T(y) = E_S(s) \quad (2)$$

$$(\{x\}, \texttt{Store}\ s, \emptyset) \in C \Rightarrow E'_S(s) = E_T(x) \quad (3)$$

$$(\{x\}, \texttt{Reduce}\ r, \{y\}) \in C \Rightarrow E_T(y) = \lambda i.r(E_T(x)) \quad (4)$$

$$(\{x\}, \texttt{Broadcast}, \{y\}) \in C \Rightarrow E_T(y) = E_T(x) \quad (5)$$

$$(\{x\}, \texttt{Shift}\ i', \{y\}) \in C \Rightarrow E_T(y) = \lambda i.E_T(x)(i + i') \quad (6)$$

$$(\emptyset, \texttt{LoadIndex}\ ax, \{y\}) \in C \Rightarrow E_T(y) = \lambda i.i!ax \quad (7)$$

$$(\{x_0, \cdots, x_{m-1}\}, \texttt{arith}\ op, \{y_0, \cdots, y_{n-1}\}) \in C$$
$$\Rightarrow E_T(y_\beta) = \lambda i.op_\beta(E_T(x_0)(i), \cdots, E_T(x_{m-1})(i)) \quad (8)$$

Parallelism in array index ***i***

Parallelism in execution order of commands.

No specified order of execution; there are dependencies, though.

A kernel (N_T, C) must satisfy the following conditions.

- For any $y \in N_T$, there exists exactly one $(xs, inst, ys) \in C$ such that $y \in ys$.
- For any $s \in N_S$, there exists at most one $(xs, inst, ys) \in C$ such that $inst = \texttt{Store}\ s$.
- The dataflow graph of the kernel is acyclic.

Lem. For any kernel $k = (N_T, C)$ and any environment E_S of an orthotope machine, there exists unique (E_T, E'_S) that satisfies $\mathit{Feasible}((k, E_S), (E_T, E'_S))$.

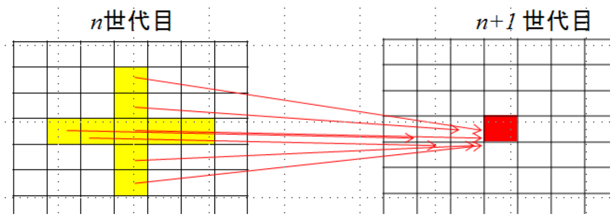
Lem. For any kernel $k = (N_T, C)$ and any environment E_S of an orthotope machine, there exists unique (E_T, E'_S) that satisfies $Feasible((k, E_S), (E_T, E'_S))$.

Exercise for the readers : prove the lemma.

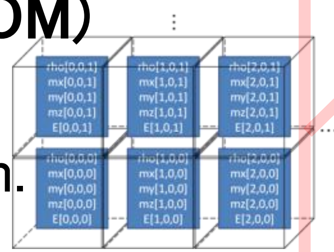
The Frontend

equation
you want to solve $$\frac{\partial U}{\partial t} + \nabla \cdot \boldsymbol{F} = 0$$

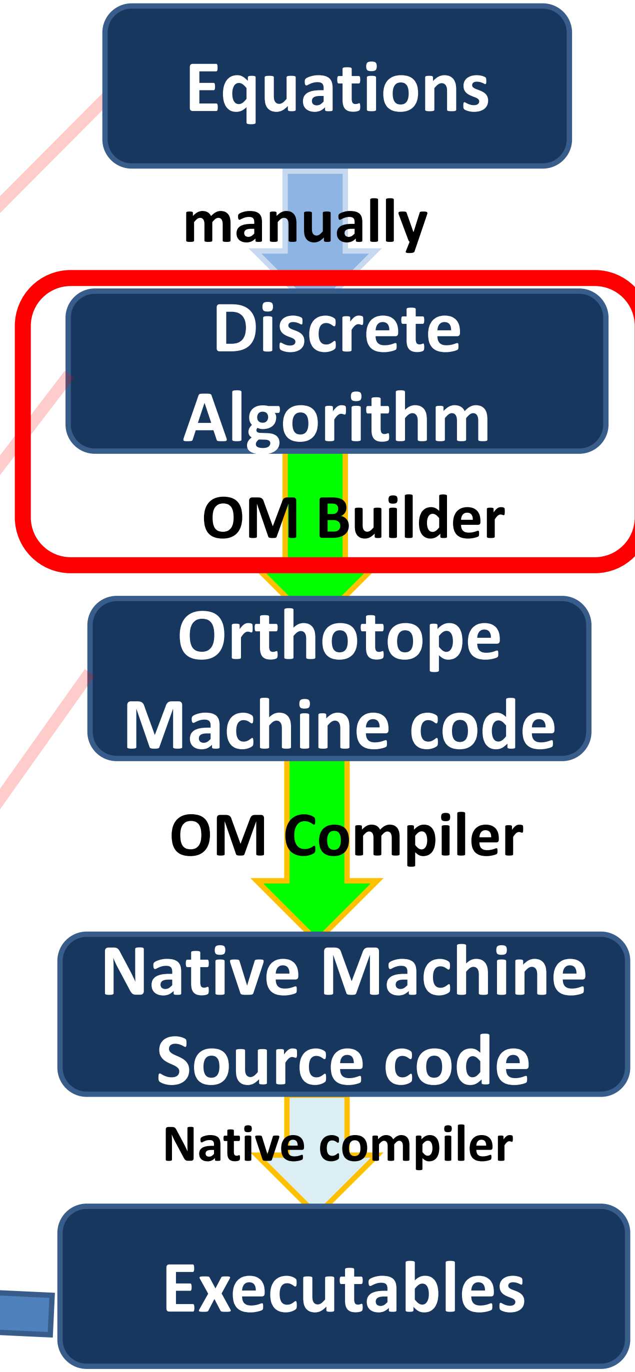
solution algorithm described in
OM Builder Monad



Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays



result



typelevel-tensor

Notation in Haskell using typelevel-tensor

```haskell
a :: Vec4 (Vec3 Double)
b :: Vec3 (Vec4 Double)
c = compose $ \i ->
    contract $ \j ->
      compose $ \k ->
        a!i!j * b!j!k
```

Einstein's notation

$$C_{ik} = A_{ij} B_{jk}$$

Implementation in C++

```cpp
double a[4][3], b[3][4];
double c[4][4];
for (int i = 0; i < 4; ++i) {
  for (int k = 0; k < 4; ++k) {
    c[i][k] = 0;
    for (int j = 0; j < 3; ++j) {
      c[i][k] += a[i][j] * b[j][k];
    }
  }
}
```

notation in standard mathematics terminology

$$C_{ik} = \sum_{j=1}^{3} A_{ij} B_{jk}$$

The tensor is Traversable

```
traverse :: Applicative f => (a -> f b) -> t a -> f (t b)

instance Traversable Vec where
  traverse _ Vec = pure Vec
instance (Traversable n) => Traversable ((:~) n) where
  traverse f (x :~ y) = (:~) <$> traverse f x <*> f y
```

- `t` : our tensor type-constructor
- `f` : some context —a code generation context
- `a, b` : elements of our tensor

`(a->f b)` : code generators for one element
`t a` : a tensor whose elements are of type a
`f (t b)` : the code generator for the entire tensor

programming language Paraiso lacks a usual frontend

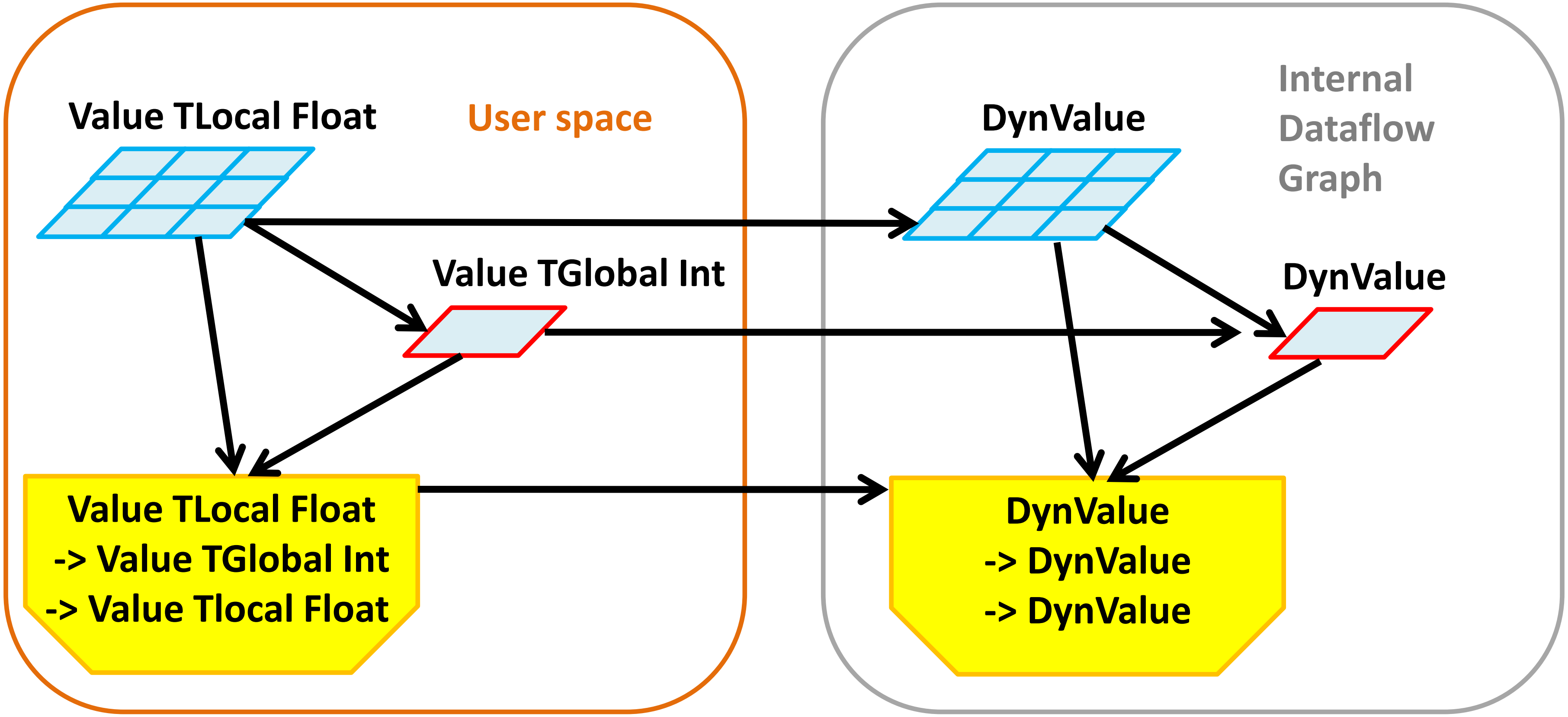
- its source code is not a string
- no Lexer, no Parser

- Paraiso is an embedded DSL in Haskell, its programme written in terms of **Builder monads and their combinators**

Builder Monads constructs dataflow graph

(a state monad that carries the half-built graph)

```haskell
-- | The 'Builder' monad is used to build 'Kernel's.
type Builder (vector:: * -> *) (gauge:: *) (anot:: *) (val:: *)
  = State.State (BuilderState vector gauge anot) val

data BuilderState vector gauge anot = BuilderState
    { setup   :: Setup vector gauge anot,
      context :: BuilderContext anot,
      target  :: Graph vector gauge anot} deriving (Show)

data BuilderContext anot =
  BuilderContext
  { currentAnnotation :: anot } deriving (Show)
```



- User interface is in Type-level
  - The type-checker helps user
  - and assures type-consistency for the backend
- Dataflow graph under cover is Value-level
  - can handle the graph in one type.

a helper function to define binary operators for Builder Monad

```
-- | Make a binary operator
mkOp2 :: (TRealm r, Typeable c) =>
         A.Operator                          -- ^The operator
      -> (Builder v g a (Value r c)) -- ^Input 1
      -> (Builder v g a (Value r c)) -- ^Input 2
      -> (Builder v g a (Value r c)) -- ^Output
mkOp2 op builder1 builder2 = do
  v1 <- builder1
  v2 <- builder2
  let
      r1 = Val.realm v1
      c1 = Val.content v1
  n1 <- valueToNode v1
  n2 <- valueToNode v2
  n0 <-  addNodeE [n1, n2] $ NInst (Arith op)
  n01 <- addNodeE [n0] $ NValue (toDyn v1)
  return $ FromNode r1 c1 n01
```

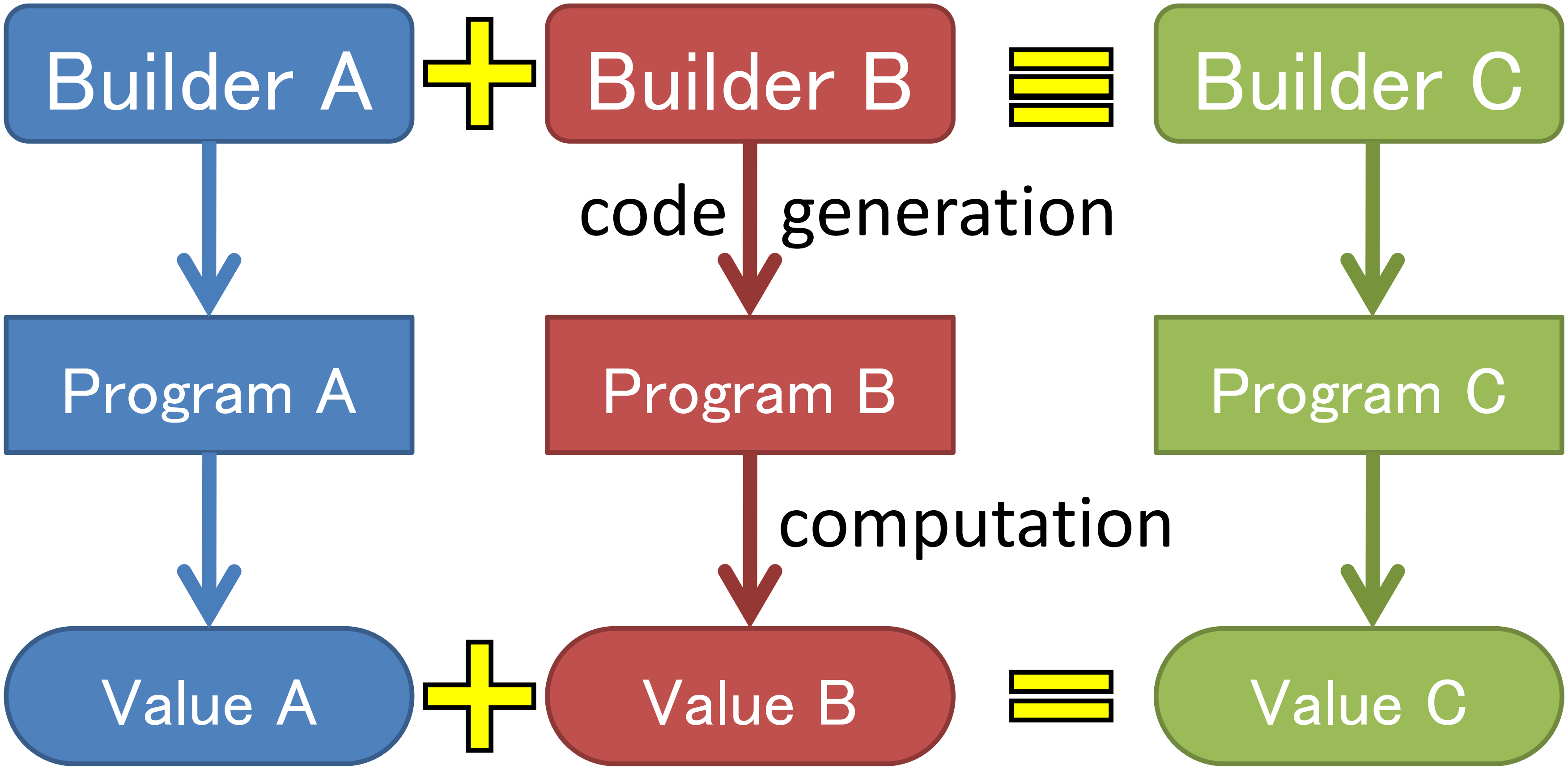
Typed user interface

Builder monad being an Additive
Builder monad being a Ring ...

```haskell
-- | Builder is Additive 'Additive.C'.
-- You can use 'Additive.zero', 'Additive.+', 'Addi
instance (TRealm r, Typeable c, Additive.C c)
 => Additive.C (Builder v g a (Value r c)) where
  zero = return $ FromImm unitTRealm Additive.zero
  (+) = mkOp2 A.Add
  (-) = mkOp2 A.Sub
  negate = mkOp1 A.Neg
```

```haskell
-- | Builder is Ring 'Ring.C'.
-- You can use 'Ring.one', 'Ring.*'.
instance (TRealm r, Typeable c, Ring.C c) => Ring.C (Builder v g a (Value r c)) where
  one = return $ FromImm unitTRealm Ring.one
  (*) = mkOp2 A.Mul
```

Builder Commutative Diagram



We define various mathematical operations between `Builder Monad` in a consistent manner (c.f. Fig. 3). For any operator $\oplus$, `Builder A` $\oplus$ `Builder B` $=$ `Builder C` is defined by `Value A` $\oplus$ `Value B` $=$ `Value C`, where `Value` i is the value computed by `Program` i which is generated by `Builder` i.

All these combined...

We can write equations compactly,
which are automatically code generators,
that generate codes corresponding to the equations!

```
hllc :: Axis Dim -> Hydro BR -> Hydro BR -> B (Hydro BR)
hllc i left right = do
  densMid  <- bind $ (density left    + density right    ) / 2
  soundMid <- bind $ (soundSpeed left + soundSpeed right) / 2
  let
      speedLeft  = velocity left  !i
      speedRight = velocity right !i
  presStar <- bind $ max 0 $ (pressure left   + pressure right  ) / 2 -
              densMid * soundMid * (speedRight - speedLeft)
  shockLeft <- bind $ velocity left !i -
               soundSpeed left * hllcQ presStar (pressure left)
  shockRight <- bind $ velocity right !i +
               soundSpeed right * hllcQ presStar (pressure right)
  shockStar <- bind $ (pressure right - pressure left
                       + density left  * speedLeft  * (shockLeft  - speedLeft)
                       - density right * speedRight * (shockRight - speedRight)
)
               / (density left  * (shockLeft  - speedLeft ) -
                  density right * (shockRight - speedRight) )
  lesta <- starState shockStar shockLeft  left
  rista <- starState shockStar shockRight right
```

Don’t Repeat Yourself

- Builder Monad is a first class resident in Haskell
- You can (easily) write code generators, code generator generators, ...

- Fundamentalistic pursuit of DRY(don’t repeat yourself) principle

Re: Matthew Sottile’s challenge

combinability

A Hydrodynamic type class

```haskell
class Hydrable a where
  density  :: a -> BR
  velocity :: a -> Dim BR
  velocity x =
    compose (\i -> momentum x !i / density x)
  pressure :: a -> BR
  pressure x = (kGamma-1) * internalEnergy x
  momentum :: a -> Dim BR
  momentum x =
      compose (\i -> density x * velocity x !i)
  energy   :: a -> BR
  energy   x = kineticEnergy x + 1/(kGamma-1) * pressure x
  enthalpy :: a -> BR
  enthalpy x = energy x + pressure x
  densityFlux  :: a -> Dim BR
```

- Automated conversion of primitive <-> conserved variables
- Dead Code Elimination helps

Hydro is Applicative

```
instance Applicative Hydro where
  pure x = Hydro
    {densityHydro = x, velocityHydro = pure x, pressureHydro = x,
     momentumHydro = pure x, energyHydro = x, enthalpyHydro = x,
     densityFluxHydro = pure x, momentumFluxHydro = pure (pure x),
     energyFluxHydro = pure x, soundSpeedHydro = x,
     kineticEnergyHydro = x, internalEnergyHydro = x}
  hf <*> hx = Hydro
    {densityHydro        = densityHydro        hf $ densityHydro        hx,
     pressureHydro       = pressureHydro       hf $ pressureHydro       hx,
     energyHydro         = energyHydro         hf $ energyHydro         hx,
     enthalpyHydro       = enthalpyHydro       hf $ enthalpyHydro       hx,
     soundSpeedHydro     = soundSpeedHydro     hf $ soundSpeedHydro     hx,
     kineticEnergyHydro  = kineticEnergyHydro  hf $ kineticEnergyHydro  hx,
     internalEnergyHydro = internalEnergyHydro hf $ internalEnergyHydro hx,
     velocityHydro       = velocityHydro    hf <*> velocityHydro    hx,
     momentumHydro       = momentumHydro    hf <*> momentumHydro    hx,
     densityFluxHydro    = densityFluxHydro hf <*> densityFluxHydro hx,
     energyFluxHydro     = energyFluxHydro  hf <*> energyFluxHydro  hx,
     momentumFluxHydro   =
         compose(\i -> compose(\j -> (momentumFluxHydro hf!i!j)
                                     (momentumFluxHydro hx!i!j)))
```

- Now you can apply functions uniformly to all the Hydro components, as you need

Interpolation in time

```
cell2 <- proceedSingle 1 (dt/2) dR cell  cell
cell3 <- proceedSingle 2  dt    dR cell2 cell
```

- This piece of code takes a first-order integrator `proceedSingle` and constructs a second-order one
- This single code can handle any integrator that takes field with any numbers of degree of freedom
- arbitrary high dimensions

Interpolation in space

```haskell
interpolate :: Int -> Axis Dim -> Hydro BR -> B (Hydro BR, Hydro BR)
interpolate order i cell = do
  let shifti n =  shift $ compose (\j -> if i==j then n else 0)
  a0 <- mapM (bind . shifti ( 2)) cell
  a1 <- mapM (bind . shifti ( 1)) cell
  a2 <- mapM (bind . shifti ( 0)) cell
  a3 <- mapM (bind . shifti (-1)) cell
  intp <- sequence $ interpolateSingle order <$> a0 <*> a1 <*> a2 <*> a3
```

- This single code
- can handle any field with any numbers of degree of freedom
- any direction of arbitrary high dimensions

Select a characteristic from shock-tube fans

```
let selector a b c d =
        select (0 `lt` shockLeft) a $
        select (0 `lt` shockStar) b $
        select (0 `lt` shockRight) c d
mapM bind $ selector <$> left <*> lesta <*> rista <*> right
```

- This single code
- can handle every degree of freedom at once
- any direction of arbitrary high dimensions

Sum up fluxes of every directions

```
proceedSingle :: Int -> BR -> Dim BR -> Hydro BR -> Hydro BR -> B (Hydro BR)
proceedSingle order dt dR cellF cellS = do
  let calcWall i = do
        (lp,rp) <- interpolate order i cellF
        hllc i lp rp
  wall <- sequence $ compose calcWall
  foldl1 (.) (compose (\i -> (>>= addFlux dt dR wall i))) $ return cellS
```

- This single code
- can handle every degree of freedom at once
- any direction of arbitrary high dimensions
- Monads, folds, partial applications.... **hard to read even for me,** to tell you the truth
- But, this small code!

Re: Matthew Sottile's challenge

Array index as a first class object

$$(\mathbf{VR}[\mathbf{i} - \tfrac{1}{2}\mathbf{e}_a], \mathbf{VL}[\mathbf{i} + \tfrac{1}{2}\mathbf{e}_a]) = \\ \quad \mathrm{Interpolate}(\mathbf{V0}[\mathbf{i} - \mathbf{e}_a], \mathbf{V0}[\mathbf{i}], \mathbf{V0}[\mathbf{i} + \mathbf{e}_a]), \tag{20}$$

$$\mathbf{F}_a[\mathbf{i} + \tfrac{1}{2}\mathbf{e}_a] = \mathrm{HLLC}_a(\mathbf{VL}[\mathbf{i} + \tfrac{1}{2}\mathbf{e}_a], \mathbf{VR}[\mathbf{i} + \tfrac{1}{2}\mathbf{e}_a]). \tag{22}$$

$$\mathbf{U2} = \mathrm{AddFlux}(\Delta t, \mathbf{F}_a, \mathbf{U1})$$

$$\Leftrightarrow \mathbf{U2}[\mathbf{i}] = \mathbf{U1}[\mathbf{i}] + \sum_a \frac{\Delta t}{\Delta r_a} \left(\mathbf{F}_a[\mathbf{i} - \tfrac{1}{2}\mathbf{e}_a] - \mathbf{F}_a[\mathbf{i} + \tfrac{1}{2}\mathbf{e}_a]\right), \tag{23}$$

Don’t Repeat Yourself

- Paraiso lacks a string-based frontend
- instead, it uses **Builder Monad**s as a frontend. Being a first-class citizen, you can put them into tensor equations, define hydrodynamic behavior of them, write algorithms and transform them ... handle them in many and meta ways.

- DRY!!

--Advanced topic--

a common drawback encountered

when doing

declarative style

to generate codes (or circuits)

Duplicated Calculations!

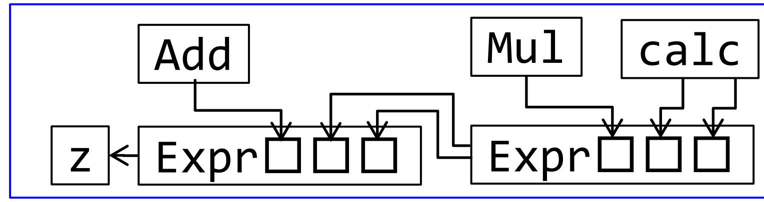
How the customer explained it

```
let x = calc
let y = x*x
let z = y+y
```

What the customer really needed

```
x = calc();
y = x*x;
z = y+y;
```

How Haskell internally represents it



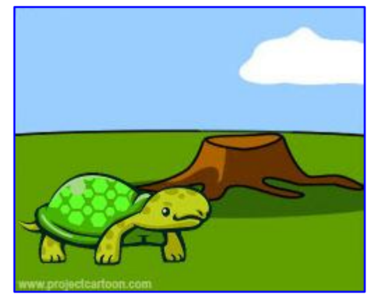
How Haskell semantically means it

```
z = Expr Add
  (Expr Mul calc calc)
  (Expr Mul calc calc)
```

What code generated

```
z =(calc()*calc())+
   (calc()*calc());
```

What speed you get



- Although the in-memory representation of Haskell avoids duplication, user cannot observe the sharing (Mainland & Morriset 2010).
- let-sharing and λ-sharing ... to recover sharing is Publishable Results at the International Conferences™ (Elliott et al. 2003, O'Donnell 1993, Bjesse et al. 1998, Claessen and Sands, 1999, Gill 2009.)

The Russians Used a Pencil

```
x <- bind $ someCalc
y <- bind $ x*x
z <- bind $ y+y
```

Paraiso generates this code

```
void Hello::Hello_sub_0 (const int & a1, int & a5)  {
int a1_0_0 = a1;
int a3_0_0 = (a1_0_0) * (a1_0_0);
(a5) = ((a3_0_0) + (a3_0_0));
}
```

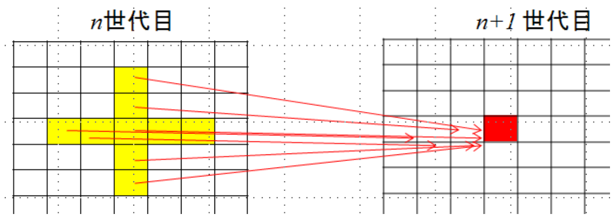
- I use monad! (Undergraduate™)
- Each term is bound to a node index in the graph in the State monad, the indices get duplicated, but calculation doesn't. The **bind** keyword does this indexing.
- Then do I need to be careful not to bind unused values?

→ NO! *dead code elimination* takes care of them

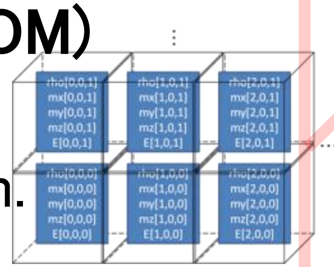
The Backend

equation
you want to solve $\frac{\partial U}{\partial t} + \nabla \cdot \boldsymbol{F} = 0$

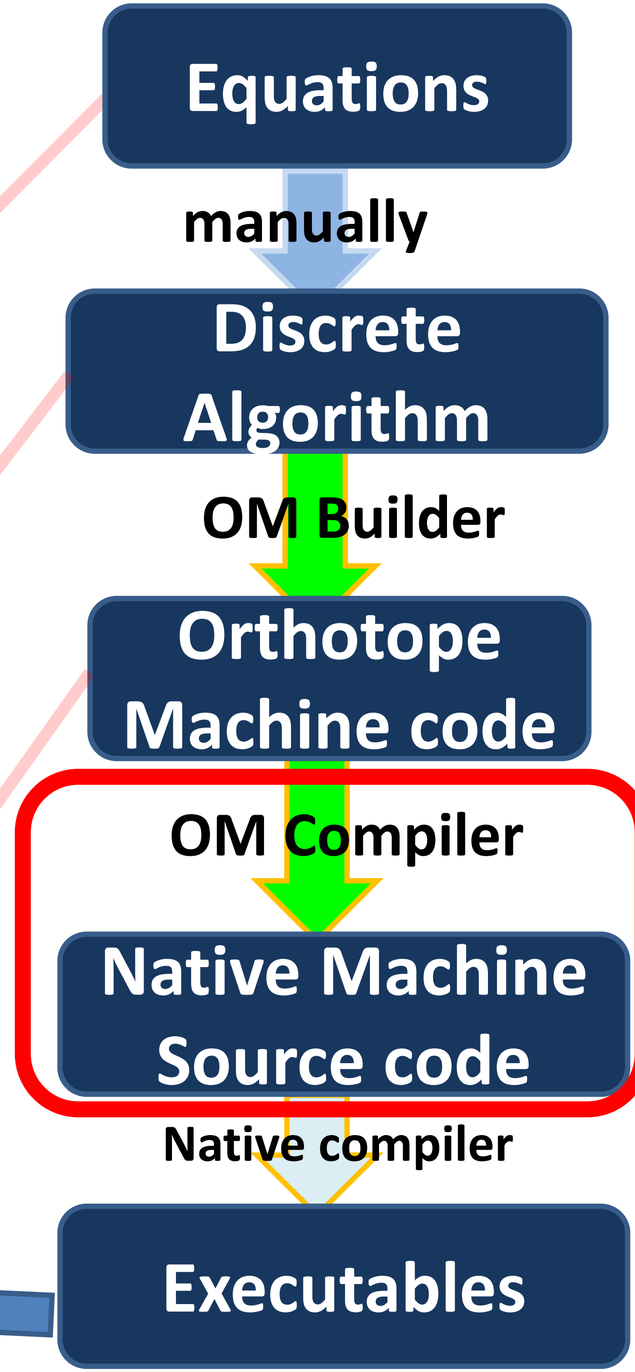
solution algorithm described in
OM Builder Monad



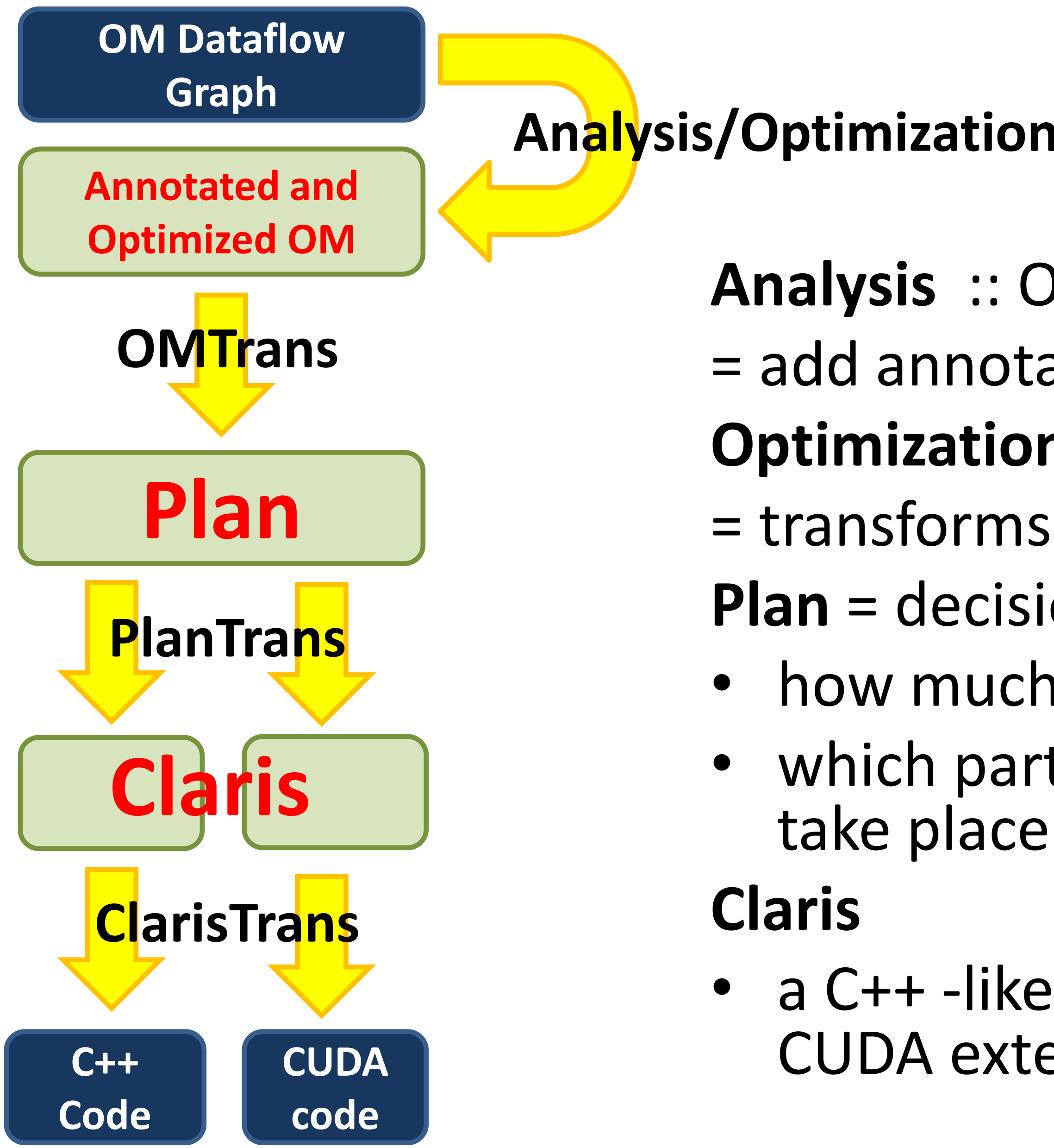
Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays



result



code generator



Analysis :: OM -> OM
= add annotations
Optimization :: OM -> OM
= transforms graph
Plan = decisions made upon

- how much memory to allocate
- which part of calculation to take place in same subroutine

Claris

- a C++ -like syntax tree with CUDA extension.

an omnibus interface for analysis and optimization

```
type Annotation = [Dynamic]
```

```
add :: Typeable a => a -> Annotation -> Annotation
```

Add an annotation to a collection.

Analyzers annotate the graph nodes with values of their favorite types

```
gmap :: (Graph v g a -> Graph v g a) -> OM v g a -> OM v g a
```

map the graph optimization to each dataflow graph of the kernel

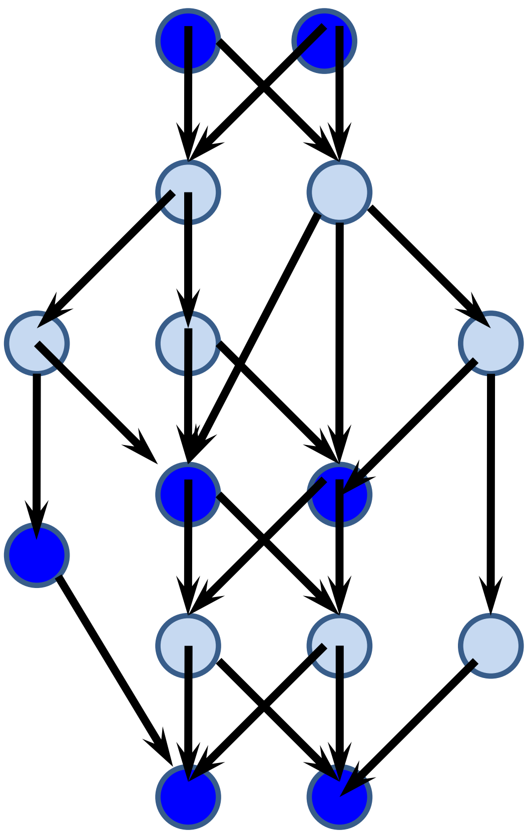
```
boundaryAnalysis :: Graph v g Annotation -> Graph v g Annotation
```

Optimizers read what type they recognize and transform graphs

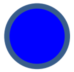
```
optimize :: Ready v g => Level -> OM v g Annotation -> OM v g Annotation
```

just one example:
an annotation for memory allocation

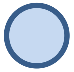
```
data Allocation
  = Existing -- ^ This entity is already allocated as a static variable.
  | Manifest -- ^ Allocate additional memory for this entity.
  | Delayed  -- ^ Do not allocate, re-compute it whenever if needed.
  deriving (Eq, Show, Typeable)
```



- some of the dataflow graph nodes are marked 'Manifest.'



Manifest nodes are stored in memory.



Delayed nodes are re-computed as needed.

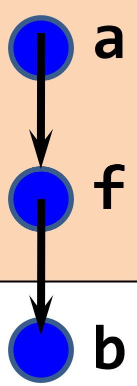
Names inherited from Repa (hackage.haskell.org/package/repa) as Matsuoka-san pointed out

Which one better?

no one but benchmark knows

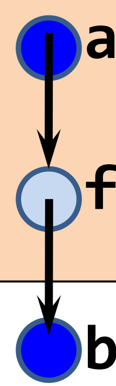
Less computation

```
for(;;){
   f[i] = calc_f(a[i], a[i+1]);
}
for (;;){
   b[i] += f[i] – f[i-1];
}
```



Less storage consumption & bandwidth

```
for(;;){
   f0 = calc_f(a[i-1], a[i]);
   f1 = calc_f(a[i], a[i+1]);
   b[i] +=  f1 – f0;
}
```



write grouping

Kernel

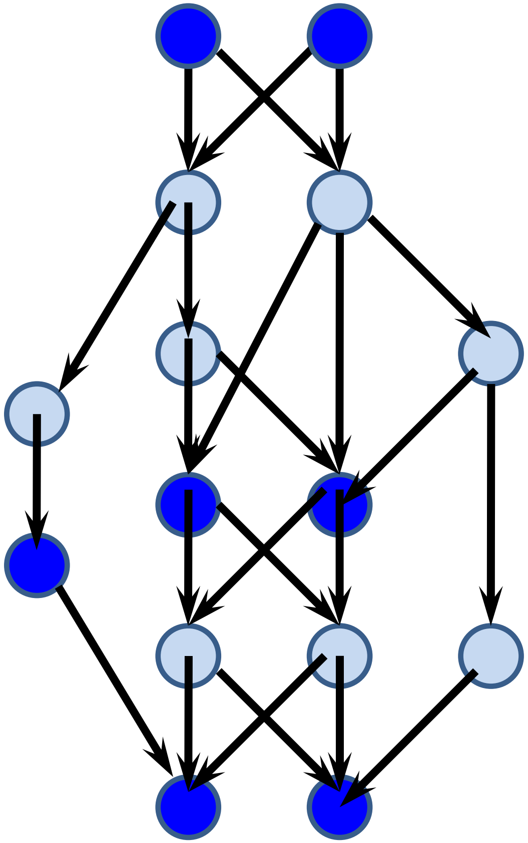
- a user-defined function that does desired task
- calls several Subkernel

Subkernel

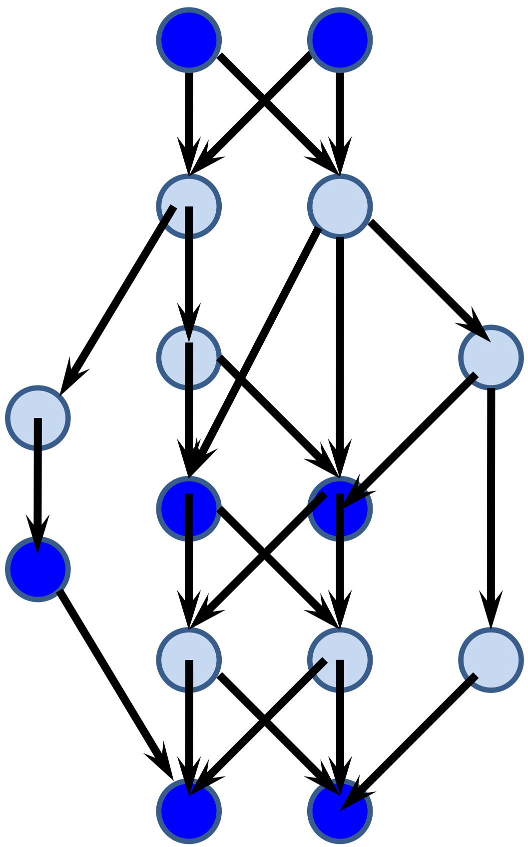
- a set of calculation executed in a loop
- = Fortran subroutine
- = CUDA __global__ kernel

```
void Life::proceed ()  { // example of a kernel calling subkernels
  Life_sub_2(static_2_cell, manifest_1_67);
  Life_sub_3(static_1_generation, manifest_1_67, manifest_1_69,
manifest_1_74);
  (static_0_population) = (manifest_1_69);
  (static_1_generation) = (manifest_1_74);
  (static_2_cell) = (manifest_1_67);
}
```

a Kernel

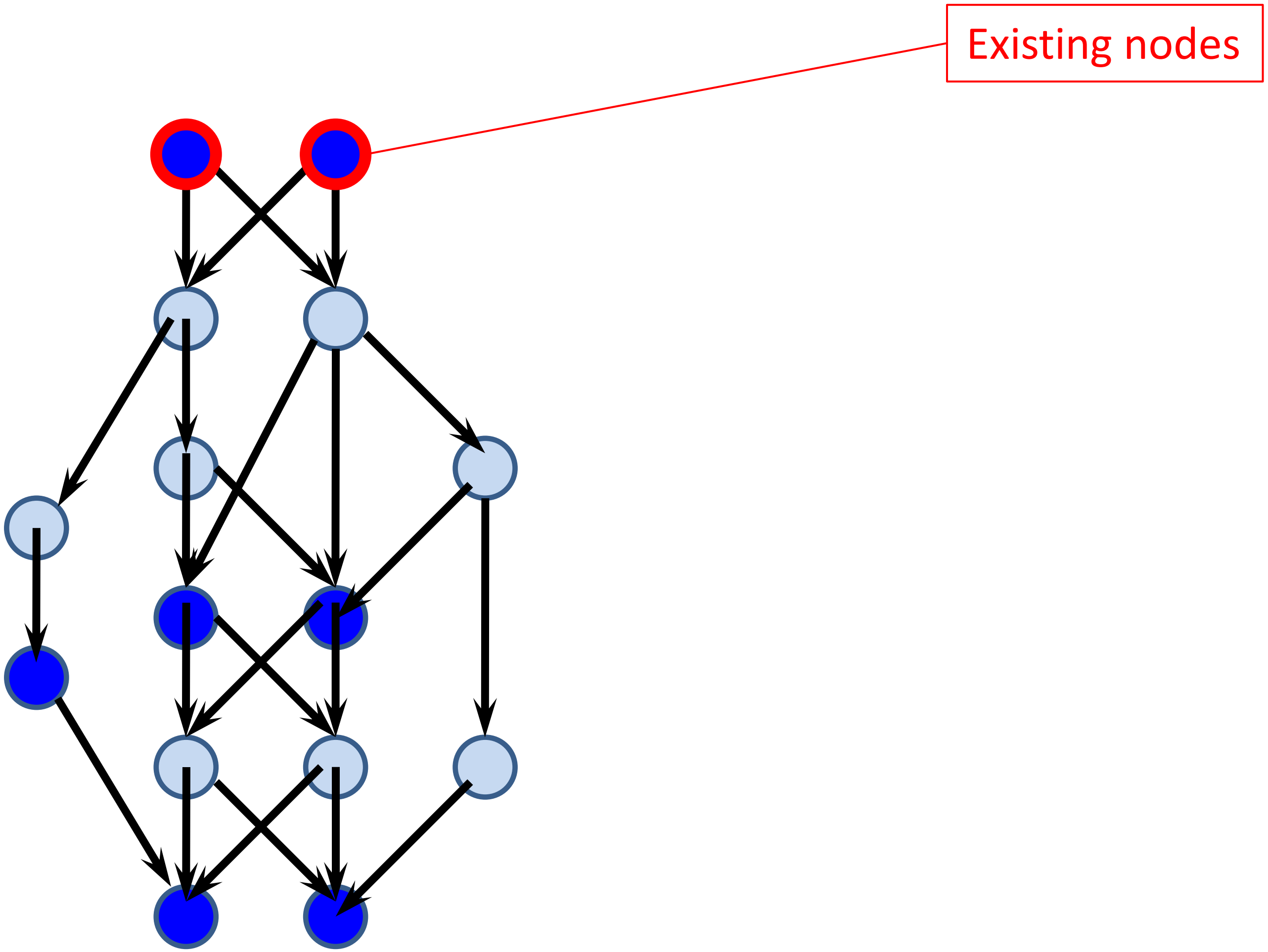


write grouping
= a Kernel -> subkernels

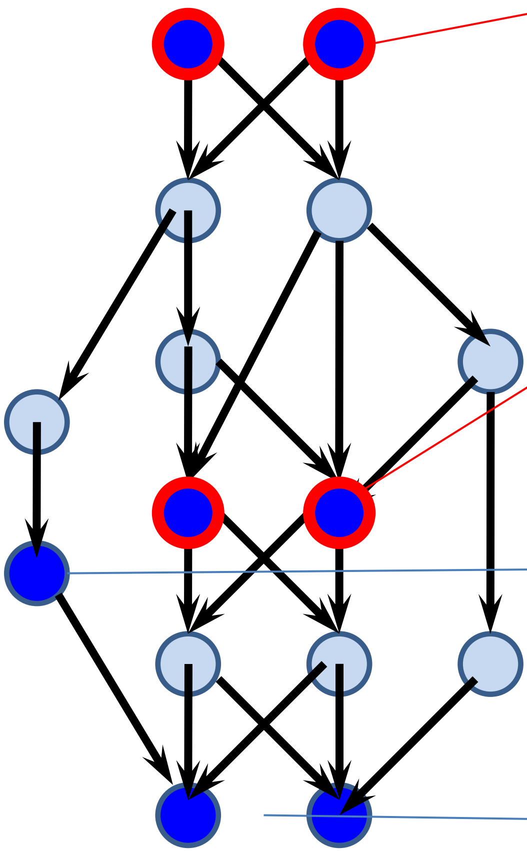


- all node written by one subkernel must have the same array size
- nodes written by one subkernel must not depend on each other
- greedy

a Kernel

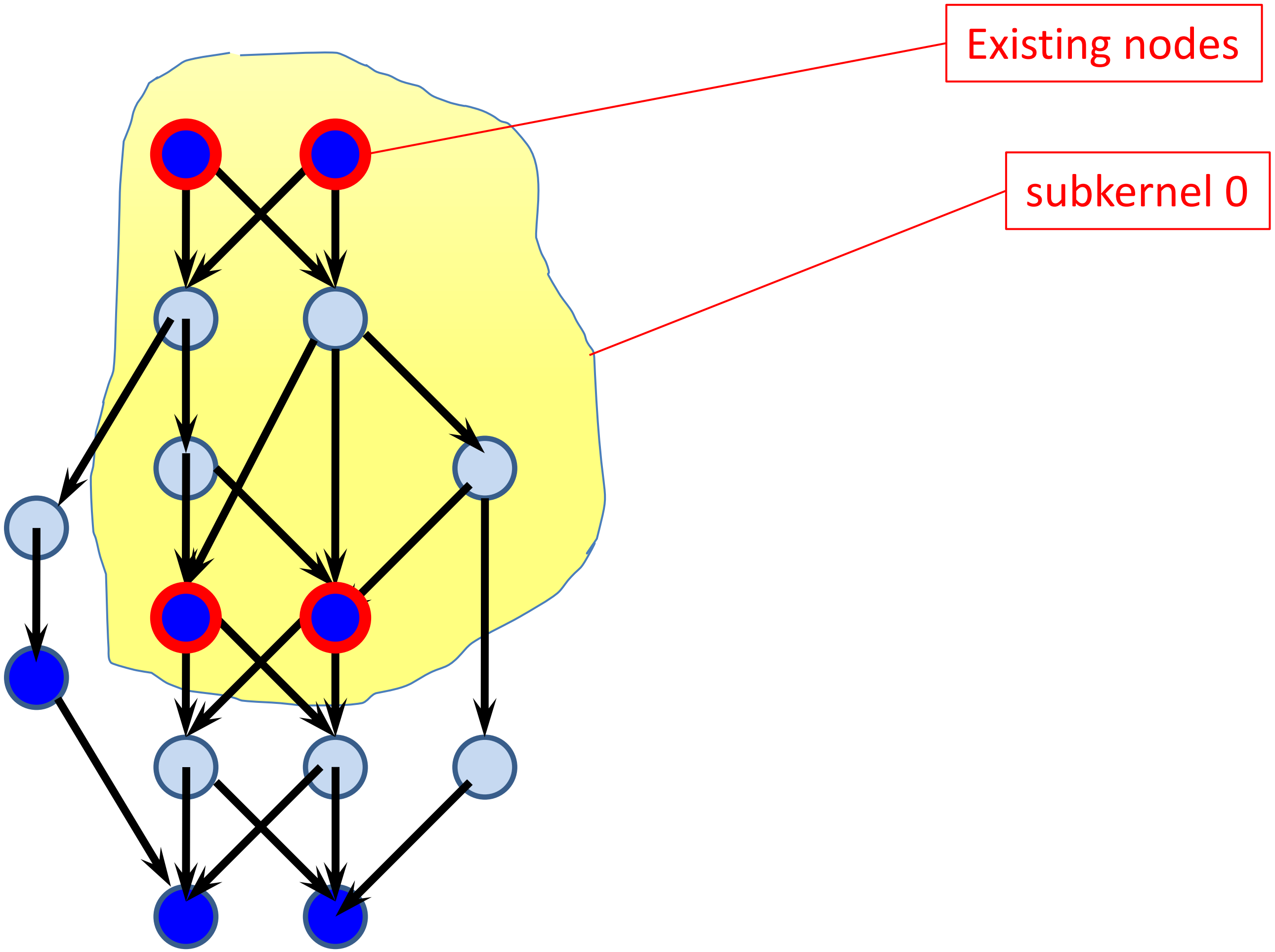


a Kernel

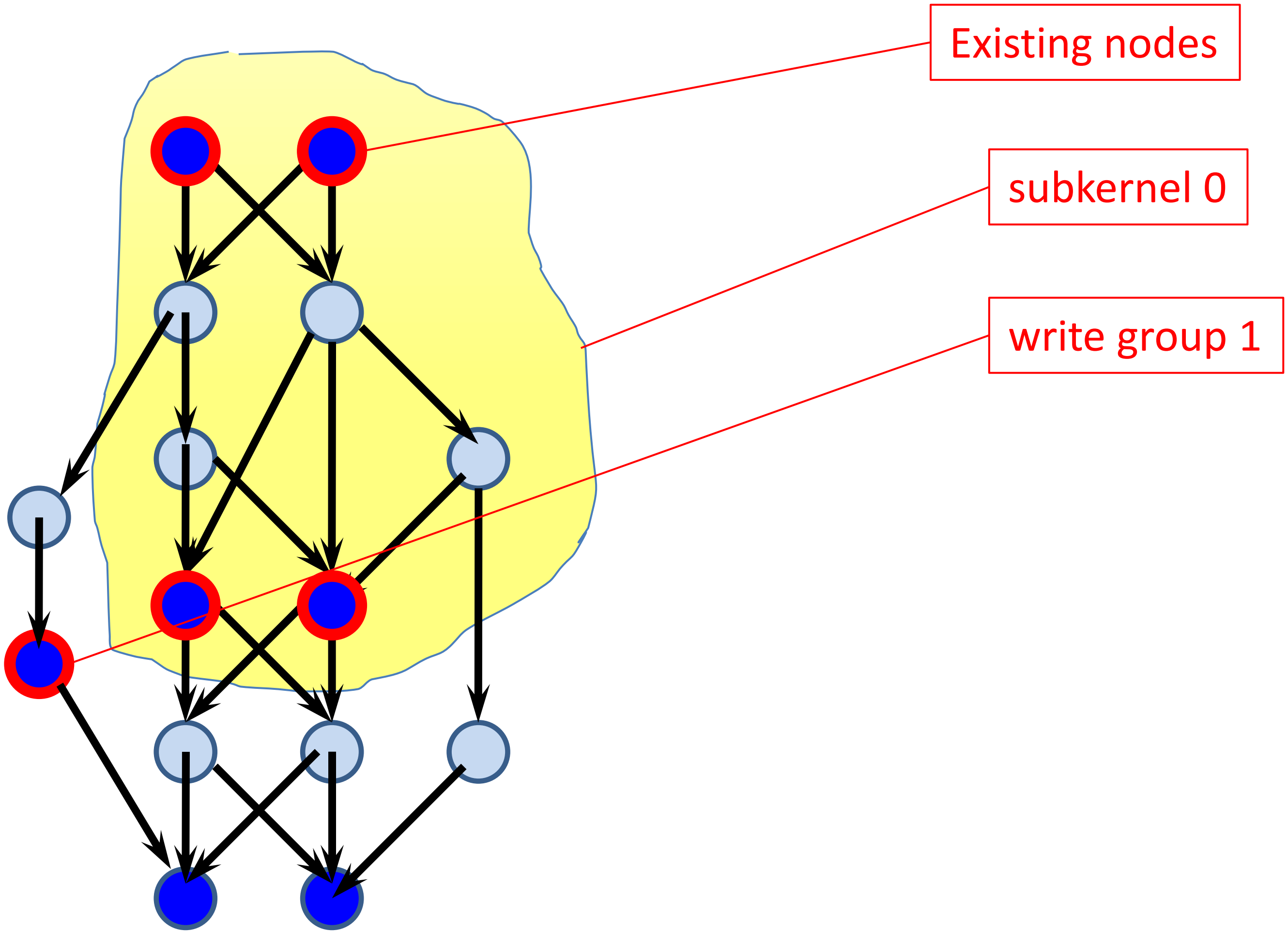


Existing nodes
write group 0
(･∀･) ready for update!
(･A･) ready for update, but let us assume this node had a different array size, so cannot be updated in the same loop as above
(･A･) dependency! not yet ready for update

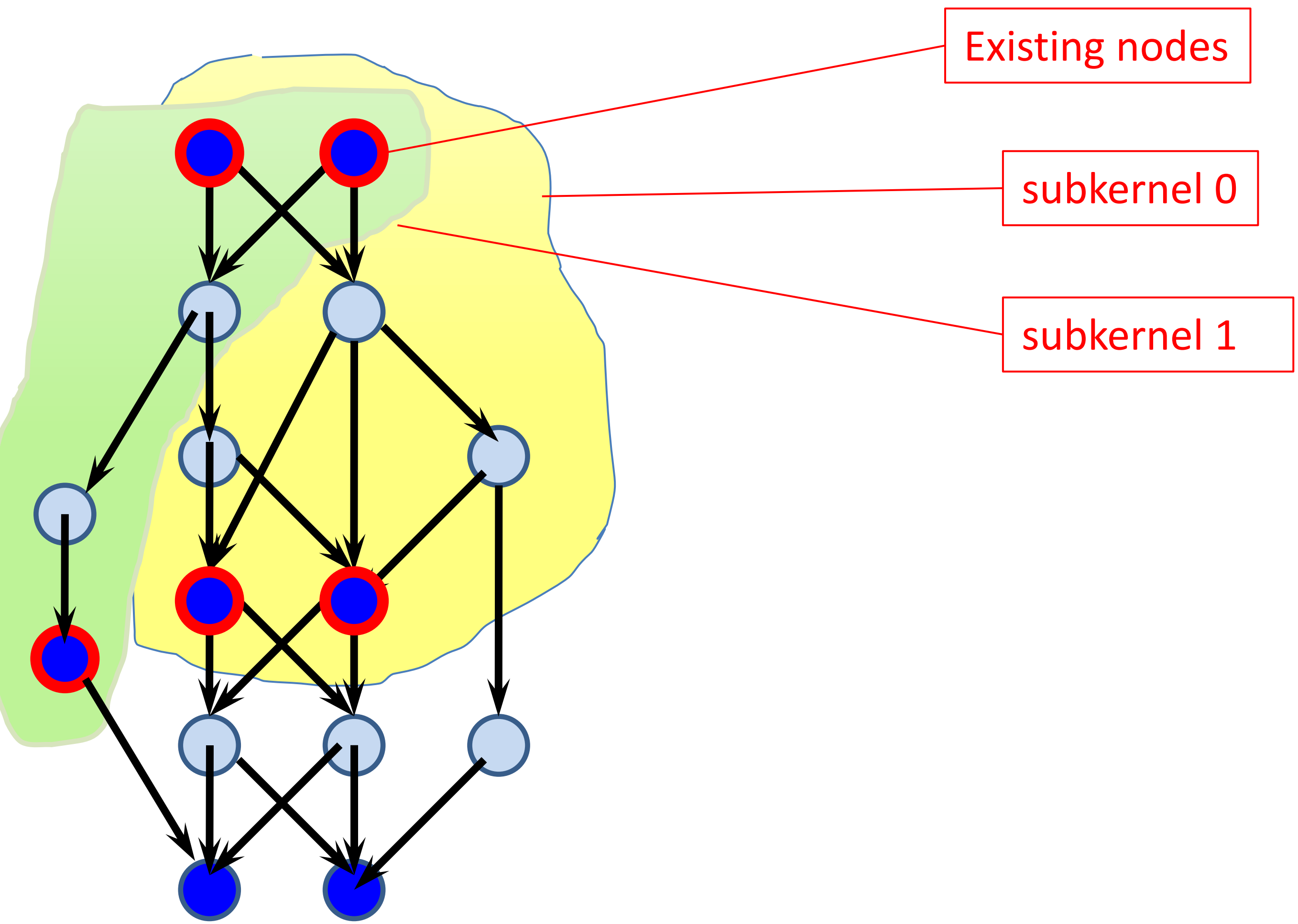
a Kernel



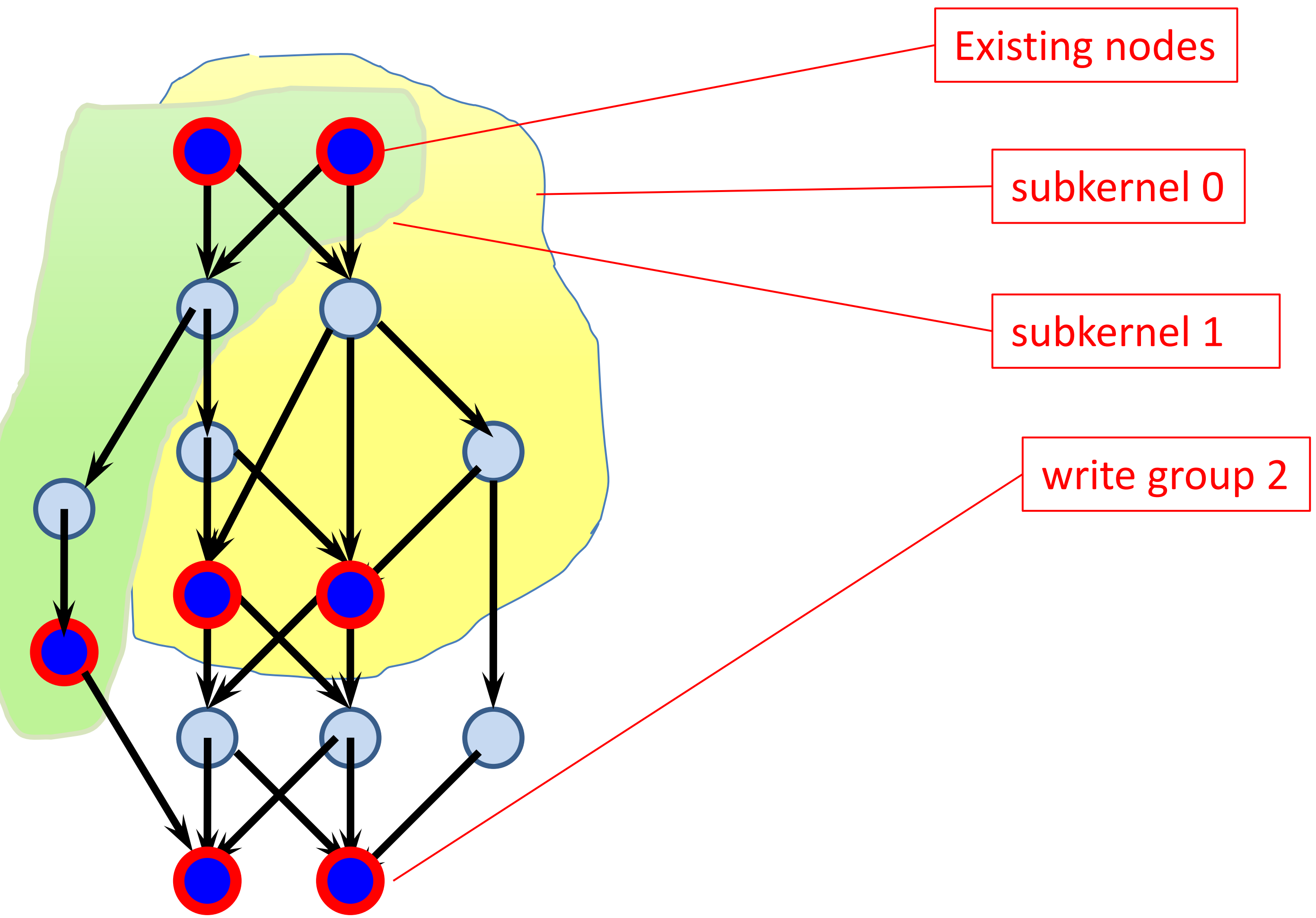
a Kernel



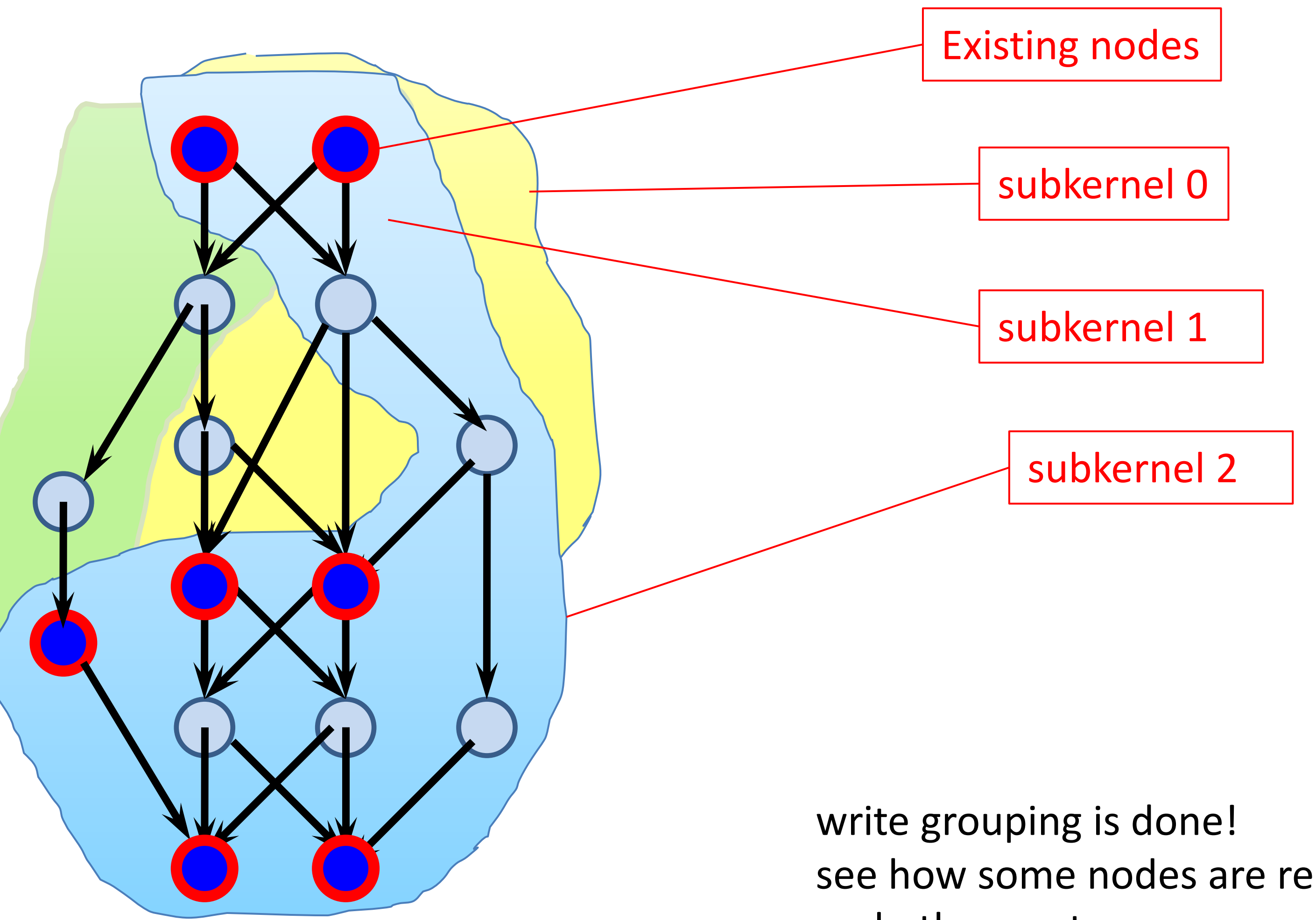
a Kernel



a Kernel



a Kernel



write grouping is done!
see how some nodes are re-calculated
and others not.

e.g. Hydrodynamics written in Paraiso

- # of nodes in graph＝3958
- # of nodes we can choose layout = 1908
- # of possible implementations

$\rightarrow 2^{1908}$

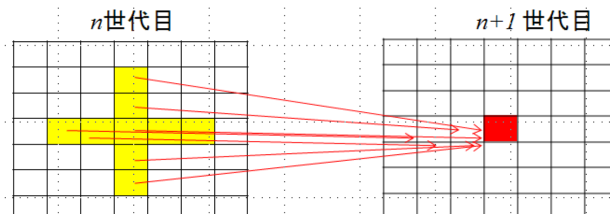
=2318631474140359897594479094137816650163390396354617107978538972914676911296
28988952894988789846447793390988399384716551223336856806783982602912691606248
36444577017233503954535729241917880311363490383137914861274921255128950712734
78839740867052195091971420983222926979177135181119534352143339906235134472215
63209222201346475070934362866728885394848451529803078779559205459073953255482
22694867051456609645215932758935244244579084816176470059329340736642337222850
66235895193869829821564571777280892089111508644034200647863717746967240332634
3875446350241918444483542305006944256

The Performance

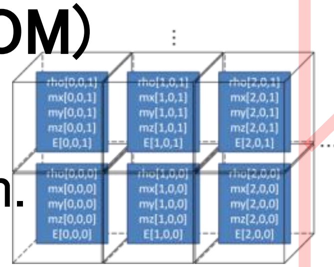
equation
you want to solve

$$\frac{\partial U}{\partial t} + \nabla \cdot \boldsymbol{F} = 0$$

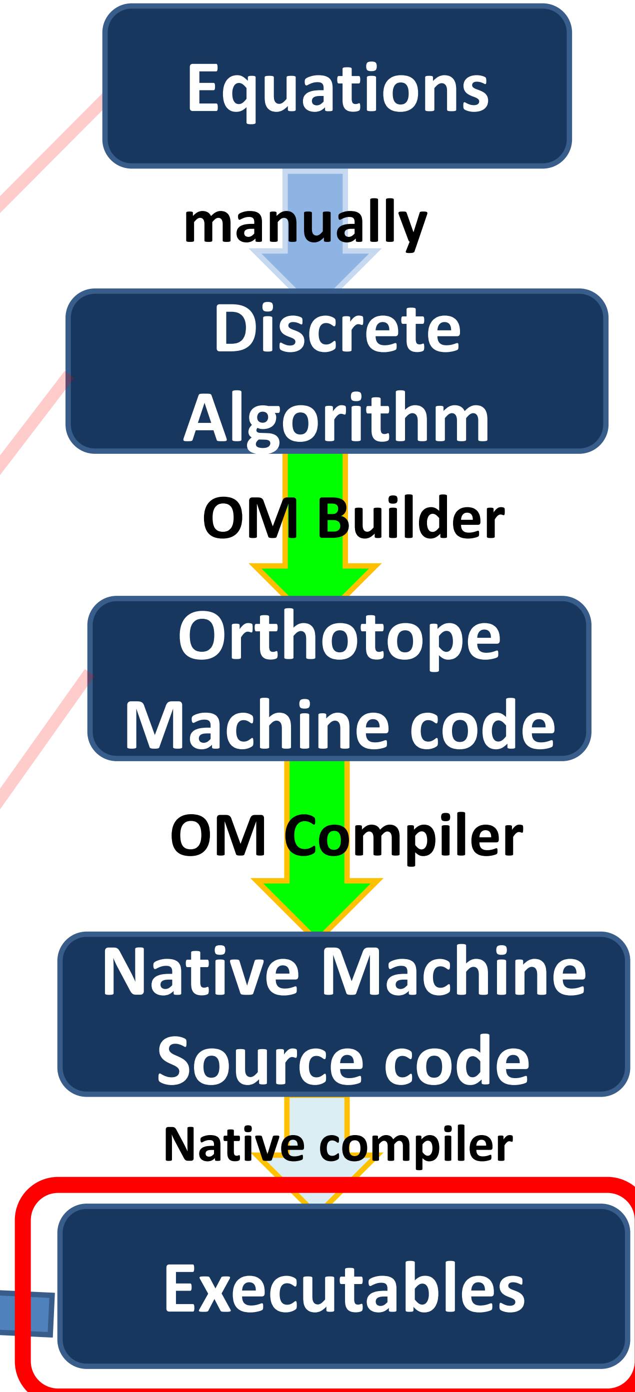
solution algorithm described in
OM Builder Monad



Orthotope Machine (OM)
Virtual machine that operates on multi-dim. arrays



result



2^{1908} different implementation of each 10’000 lines of code, generated from

Paraiso

- A framework for writing any hyperbolic partial differential equations solver
- 4299 lines

Hydro.hs

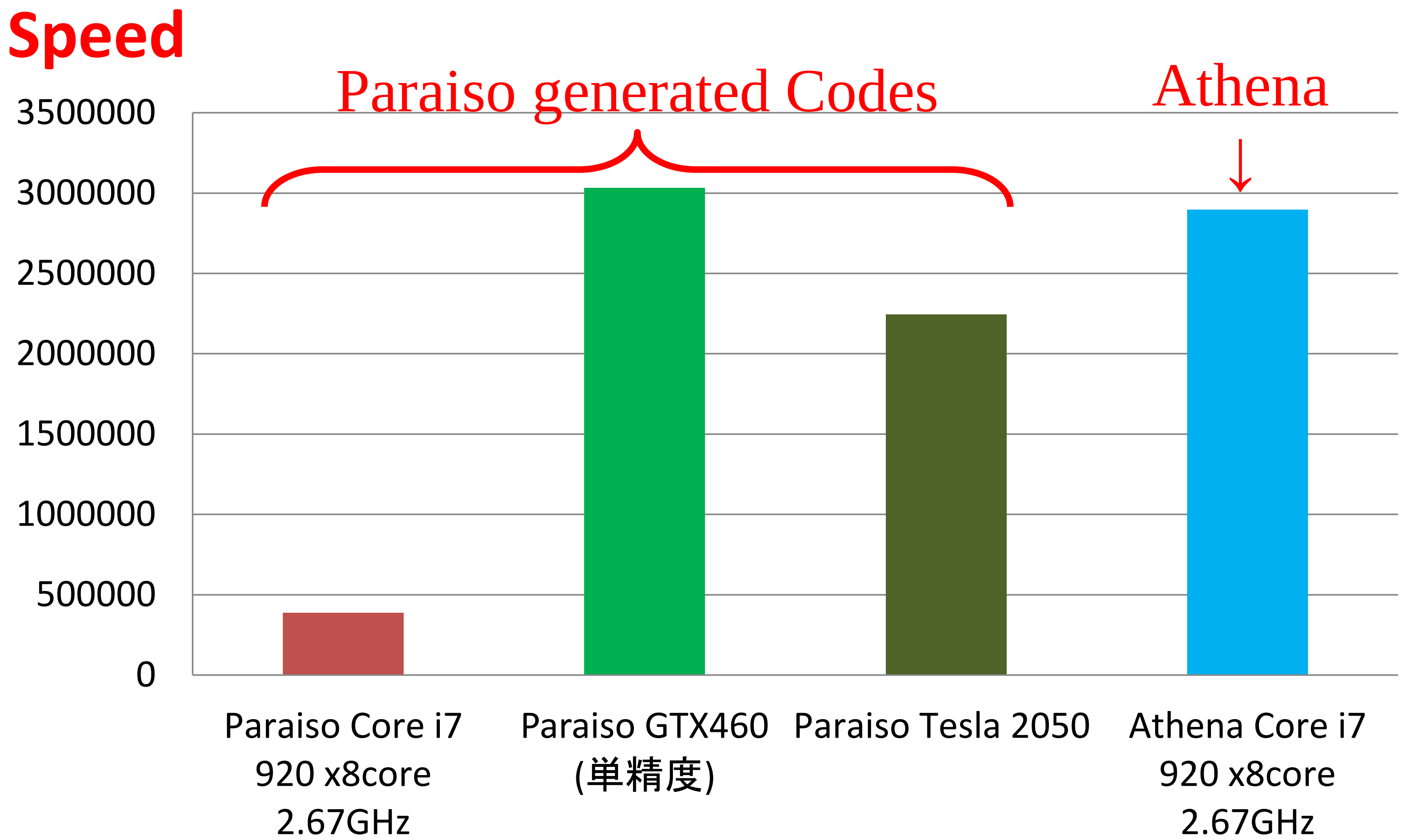
HydroMain.hs

- a Navier-Stokes equations solver written in Paraiso
- 464 lines

Movie

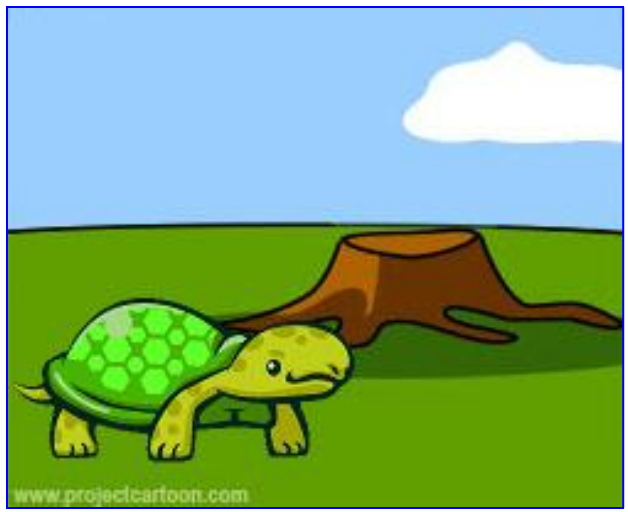
- `movie-2-jet.avi`
- 1024^2 Resolution
- A shockwave formed by supersonic jet

Benchmark Results



Athena: An open-source plasma simulator widely used in our field. I’m 10 times slower than them! What a shame!

What speed you get




Land of the Rising Sun, JAPAN
We won't give in!
Thank you for your prayers, words, and competitive compassion.

Why not see how 2^{1908}-1 other implementation performs?

```
interpolateSingle :: Int -> BR -> BR -> BR -> BR -> B (BR,BR)
interpolateSingle order x0 x1 x2 x3 =
  if order == 1
  then do
    return (x1, x2)
  else if order == 2
       then do
         d01 <- bind $ x1-x0
         d12 <- bind $ x2-x1
         d23 <- bind $ x3-x2
         let absmaller a b = select ((a*b) `le` 0) 0 $ select (abs a `lt` abs b) a b
         d1 <- bind $ absmaller d01 d12
         d2 <- bind $ absmaller d12 d23
         l <- bind $ x1 + d1/2
         r <- bind $ x2 - d2/2
         return ( Anot.add Alloc.Manifest <?> l,  Anot.add Alloc.Manifest <?> r)
       else error $ show order ++ "th order spatial interpolation is not yet implemented"
```

```
(<?>) :: (TRealm r, Typeable c) => (a -> a) -> Builder v g a (Value r c) -> Builder v g a (Value r c)
```

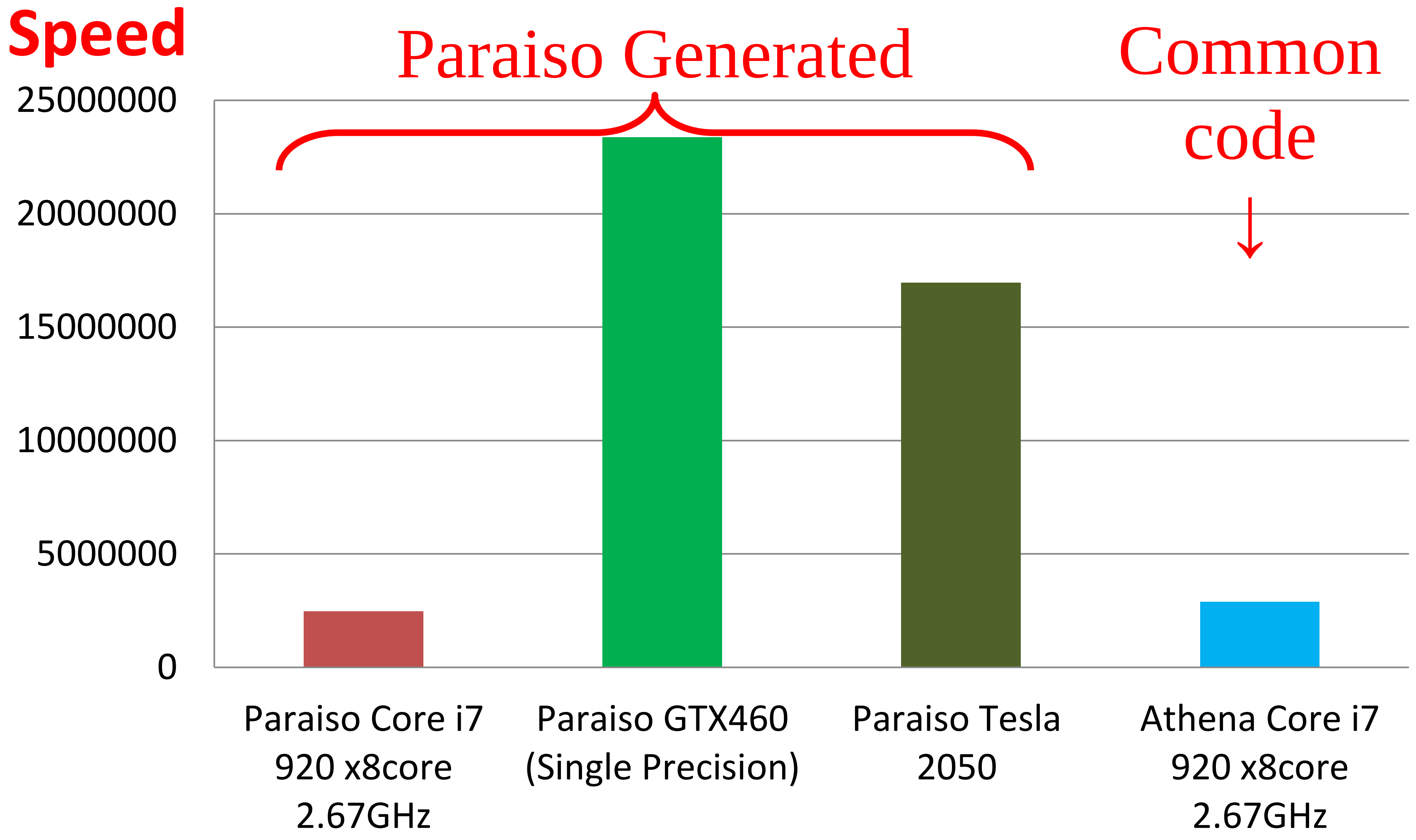
(Anot.add AnyAnnotation <?>) has an identity type on **Builder**;
you can freely add any annotation at almost anywhere in builder combinator equation.

I also add annotations here...

```haskell
hllc :: Axis Dim -> Hydro BR -> Hydro BR -> B (Hydro BR)
hllc i left right = do
  densMid  <- bind $ (density left    + density right   ) / 2
  soundMid <- bind $ (soundSpeed left + soundSpeed right) / 2
  let
      speedLeft  = velocity left  !i
      speedRight = velocity right !i
  presStar <- bind $ max 0 $ (pressure left   + pressure right  ) / 2 -
              densMid * soundMid * (speedRight - speedLeft)
  shockLeft <- bind $ velocity left !i -
               soundSpeed left * hllcQ presStar (pressure left)
  shockRight <- bind $ velocity right !i +
               soundSpeed right * hllcQ presStar (pressure right)
  shockStar <- bind $ (pressure right - pressure left
                       + density left  * speedLeft  * (shockLeft  - speedLeft)
                       - density right * speedRight * (shockRight - speedRight) )
               / (density left  * (shockLeft  - speedLeft ) -
                  density right * (shockRight - speedRight) )
  lesta <- starState shockStar shockLeft  left
  rista <- starState shockStar shockRight right
  let selector a b c d =
        (Anot.add Alloc.Manifest <?> ) $
        select (0 `lt` shockLeft) a $
        select (0 `lt` shockStar) b $
        select (0 `lt` shockRight) c d
  mapM bind $ selector <$> left <*> lesta <*> rista <*> right
    where
```

Manifest Strategy	Hardware	size of .cu file	number of CUDA kernels	memory consumption	speed (mesh/s)
none		13108 lines	7	52 x N	3.03×10^6
HLLC + interpolate	GTX 460	3417 lines	15	84 x N	22.38×10^6
HLLC only	GTX 460	2978 lines	11	68 x N	23.37×10^6
interpolate only	GTX 460	17462 lines	12	68 x N	0.68×10^6
HLLC only	Tesla M2050	2978 lines	11	68 x N	16.97×10^6
HLLC only	Core i7 x8	2978 lines		68 x N	2.48×10^6
Athena	Core i7 x8				2.90×10^6

Benchmark rev.2

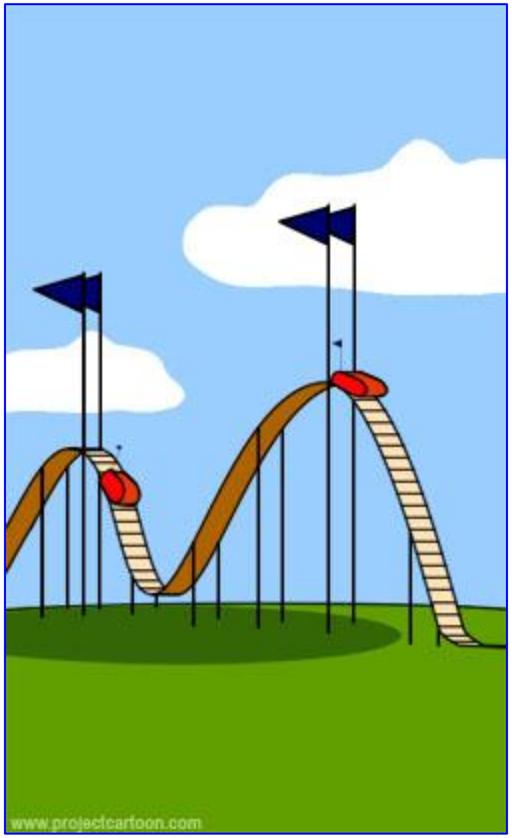


By adding two lines of annotation

- We made several tens of nodes Manifest

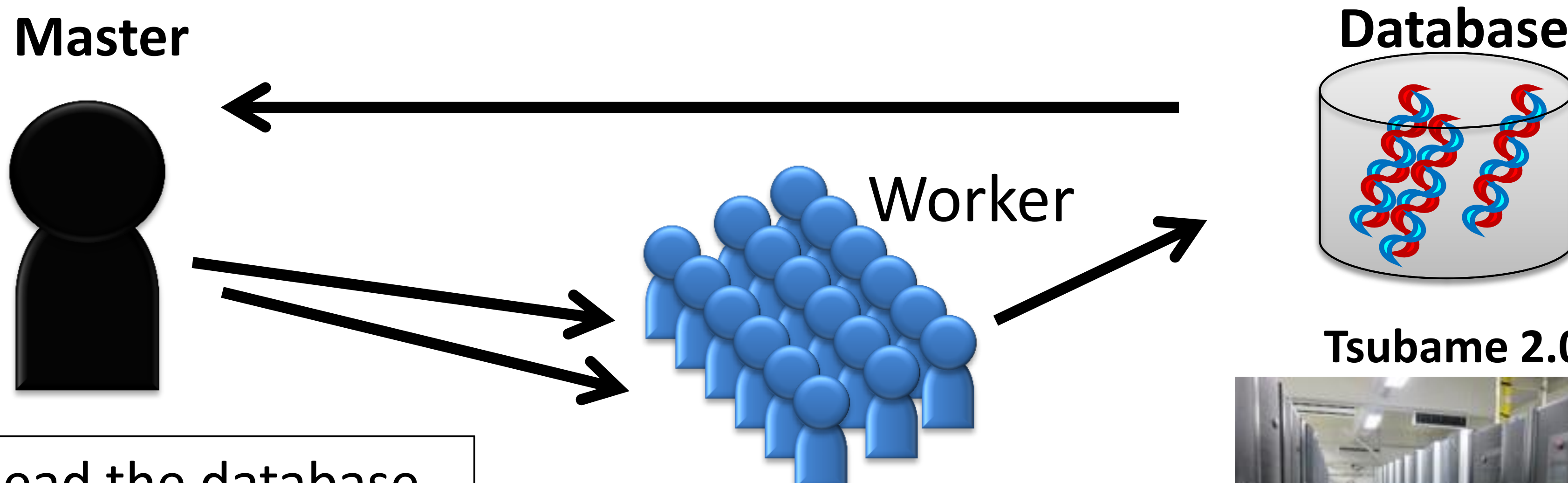
  (not just two; applicative functors and traversables work as leverage)
- Our generated codes is ¼ in line number
- Our code makes double more CUDA kernel call per generation
- Our code uses slightly more memory
- and 7 times faster than it used to be!

What speed you get rev.2



3-2. Automated Tuning with Genetic Algorithms

Automated Tuning System



Read the database, create new genomes and launch workers

Given a genome, generate an individual code, measure its speed, and write it into the database.

Tsubame 2.0

Automated tuning testbed

Three things to optimize:

- C : cuda configuration `<<<NT,NB>>>`
- M : Manifest/Delay

  (Manifest : to store intermediate data on memory
  Delayed: not to store and recompute as needed)
- S : `__syncthreads()`

- more you can add, if you want

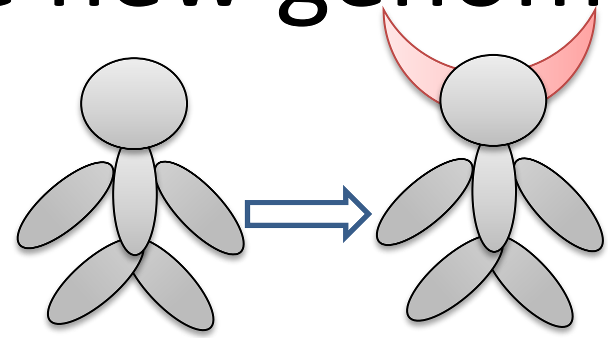
Three ways to create new genomes

- mutation (1 parent)

ATATATAAATTATATATATAAAAAAAAAAAAT

↓

ATATAGCAATTATATCTATAAAAAGTGAAAAT



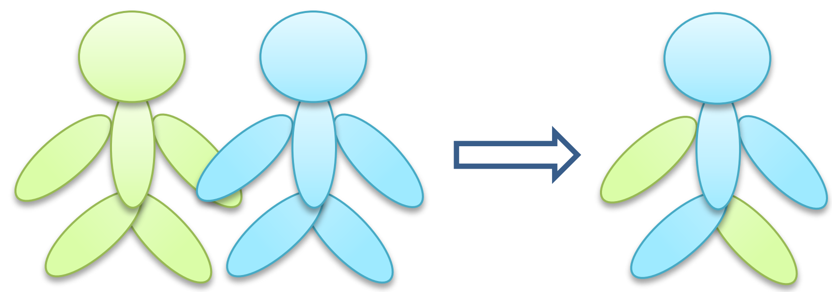
- crossover (2 parents)

ATATATAAATTATATATATAAAAAAAAAAAAT

GGCCGCGCCCGCGCGCCCGCGCGCCCGGCGG

↓

ATATGCGAATTATATATACGCGCGCCCGGCGT



- triangulation (3 parents)

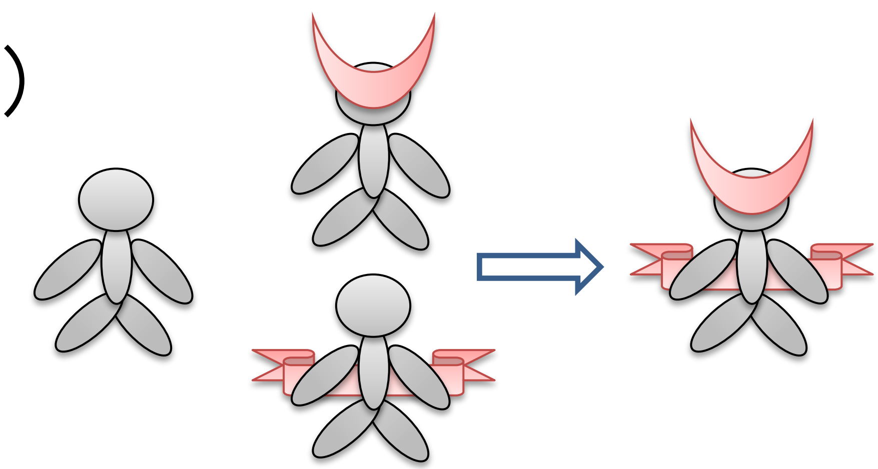
ATATATAAATTATATATATAAAAAAAAAAAAT

ATATATAAATTATATATATAAAAAAGTTAAAT

ATATAGCAATTATATCTATAAAAAAAAAAAAT

↓

ATATAGCAATTATATCTATAAAAAAGTTAAAT



Individuals annotated by hand

ID	config	(1)	(2)	lines	subKernel	memory
Izanagi	32×32	D	D	13128	7	$52 \times N$
Izanami	448×256	D	D	13128	7	$52 \times N$
Iwatsuchibiko	448×256	M	D	17494	12	$68 \times N$
Shinatsuhiko	448×256	D	M	3010	11	$68 \times N$
Hayaakitsuhime	448×256	M	M	3462	15	$84 \times N$

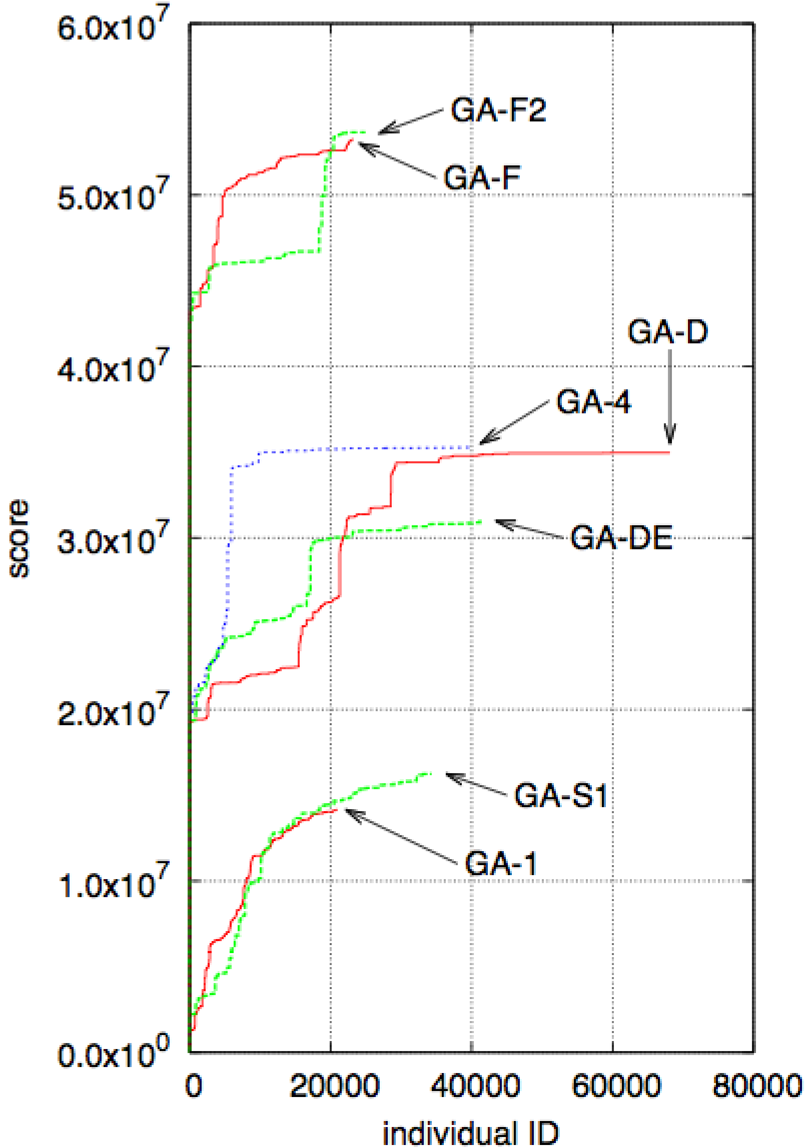
ID	score (SP)	score (DP)
Izanagi	1.551 ± 0.0005	1.138 ± 0.000
Izanami	5.838 ± 0.004	3.091 ± 0.002
Iwatsuchibiko	5.015 ± 0.002	2.491 ± 0.001
Shinatsuhiko	42.682 ± 0.083	19.831 ± 0.021
Hayaakitsuhime	34.100 ± 0.110	15.632 ± 0.024

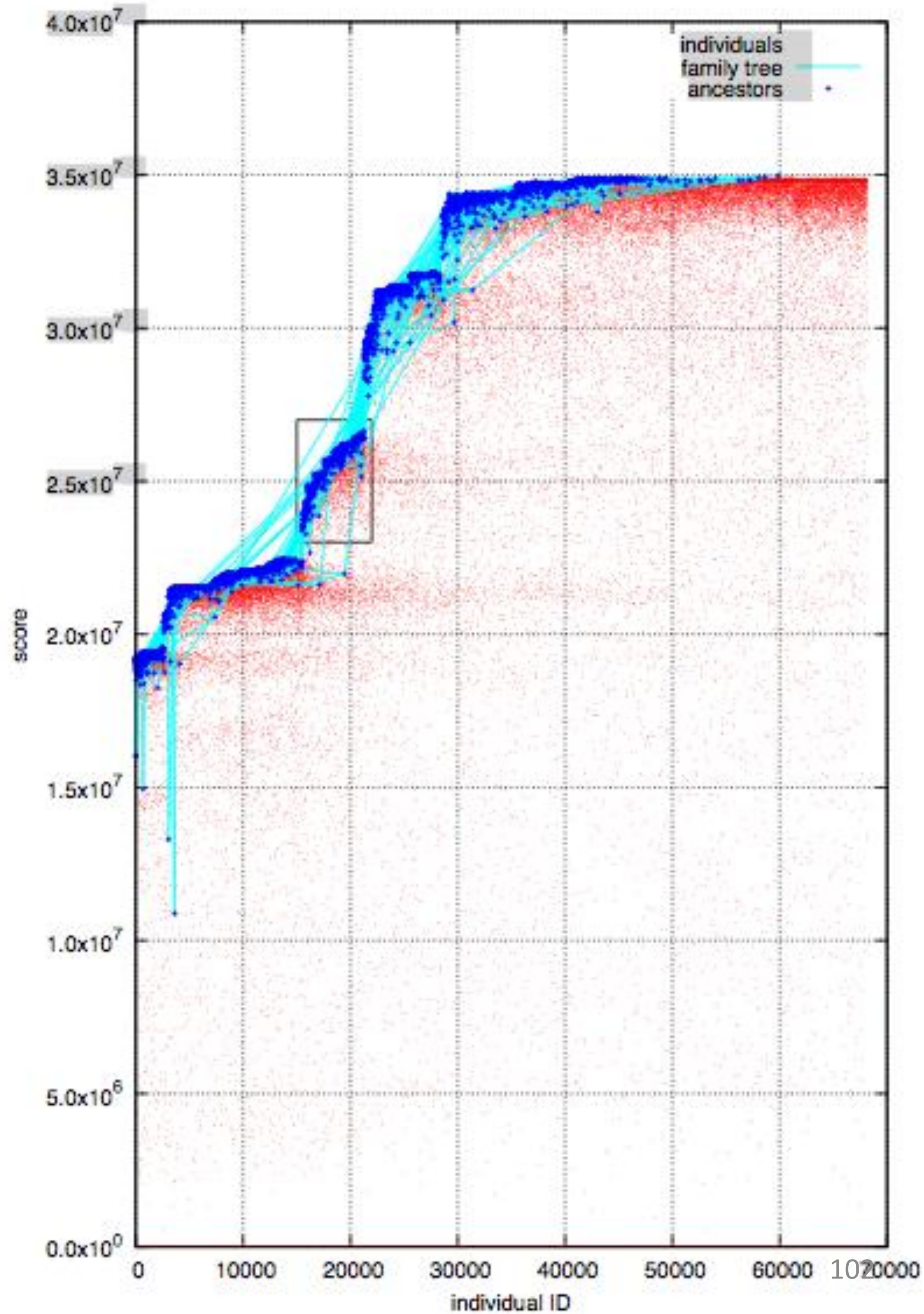
Indibiduals generated by GA

RunID	prec.	initial score	wct	best ID/total	highscore
GA-1	DP	1.138 ± 0.000	3870	20756 / 20885	14.158 ± 0.002
GA-S1	DP	1.138 ± 0.000	4120	33958 / 34328	16.247 ± 0.002
GA-DE	DP	19.253 ± 0.044	7928	41250 / 41386	31.015 ± 0.032
GA-D	DP	19.253 ± 0.044	8770	59841 / 68138	34.968 ± 0.043
GA-4	DP	19.253 ± 0.044	5811	39991 / 40262	35.303 ± 0.035
GA-F	SP	42.682 ± 0.083	2740	23019 / 23062	53.300 ± 0.078
GA-F2	SP	42.682 ± 0.083	4811	22242 / 24887	53.656 ± 0.078
GA-3D	SP	24.638 ± 0.001	5702	38146 / 39200	45.443 ± 0.116

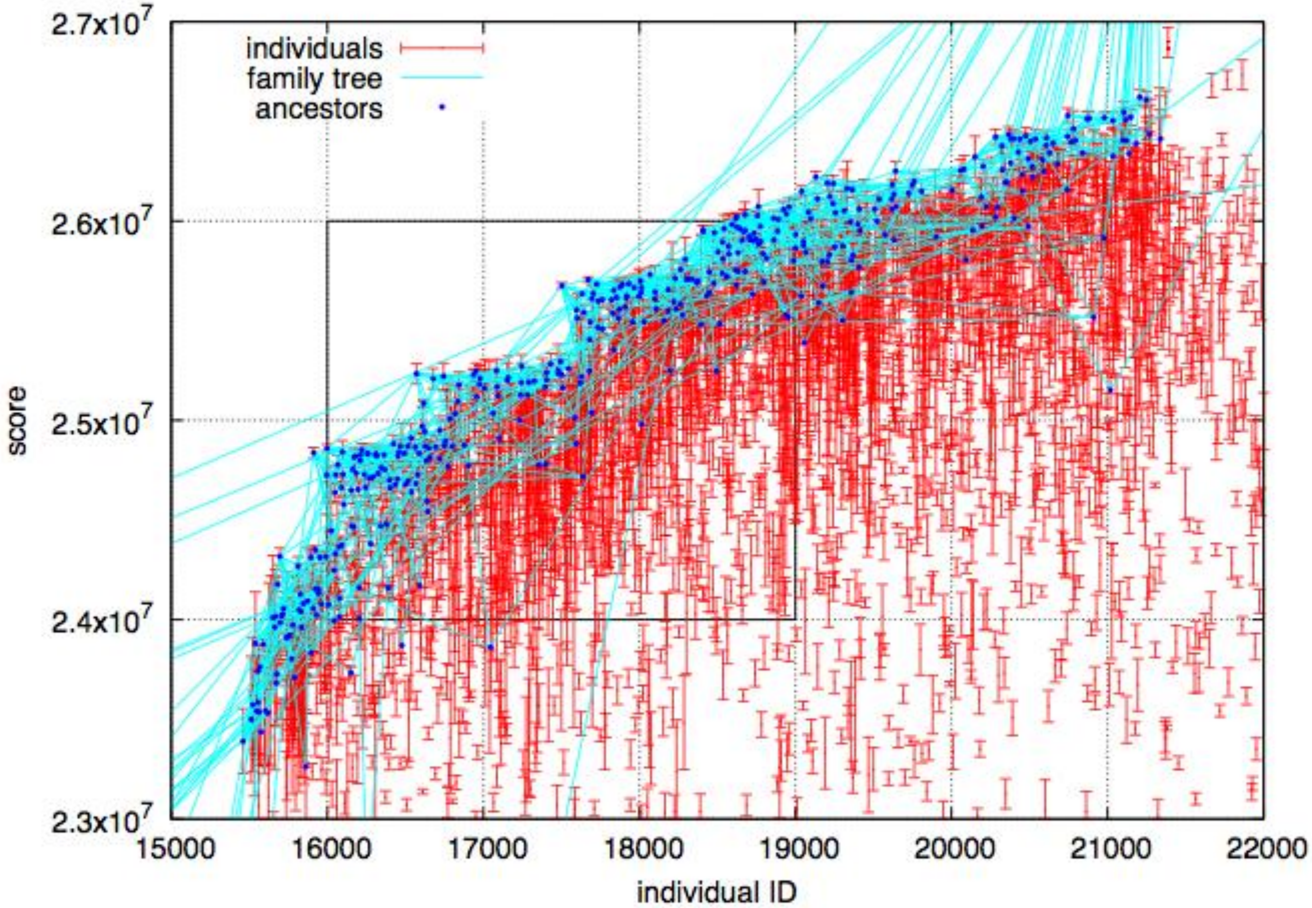
Table 3. The statistics of auto-tuning experiments. The columns are RunID, precision, the score of initial individual, the wall-clock time for the experiment (in minutes), the ID of the best individual and the number of individuals generated, the highscore (in Mcups). Experiments GA-1 and GA-S1 started with *Izanagi*, others started with *Shinatsuhiko*. GA-3D started with *Shinatsuhiko*, and solved 3D problems.

evolution tracks



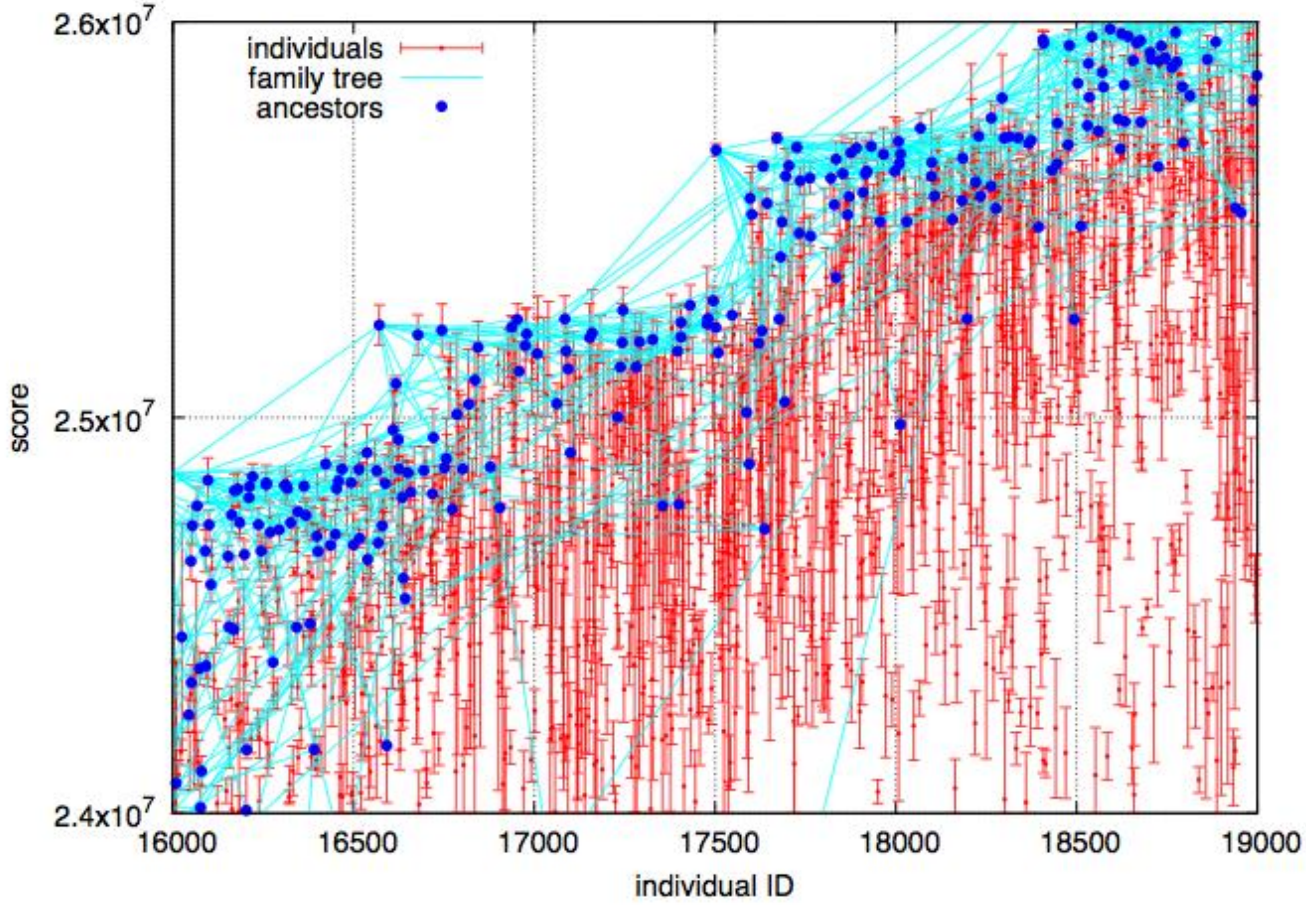


zoom-in (1)



individuals
family tree
ancestors
score
individual ID
2.7x10^7
2.6x10^7
2.5x10^7
2.4x10^7
2.3x10^7
15000
16000
17000
18000
19000
20000
21000
22000

zoom-in (2)



2.6x10^7
2.5x10^7
2.4x10^7
score
individuals
family tree
ancestors
16000
16500
17000
17500
18000
18500
19000
individual ID

How are three methods of birth (mutation, crossover, triangulation) working and interacting?

try switching off the method of birth

RunID	prec.	initial score	wct	best ID/total	highscore
GB-333-0	DP	19.253 ± 0.044	TODO	TODO / TODO	TODO
GB-333-1	DP	19.253 ± 0.044	TODO	TODO / TODO	TODO
GB-333-2	DP	19.253 ± 0.044	TODO	TODO / TODO	TODO
GB-370-0	D				
GB-370-1	D				
GB-370-2	D				
GB-307-0	D				
GB-307-1	D				
GB-307-2	D				

	mutation	crossover	triangulation
GB-333	1/3	1/3	1/3
GB-370	1/3	2/3	0
GB-307	1/3	0	2/3

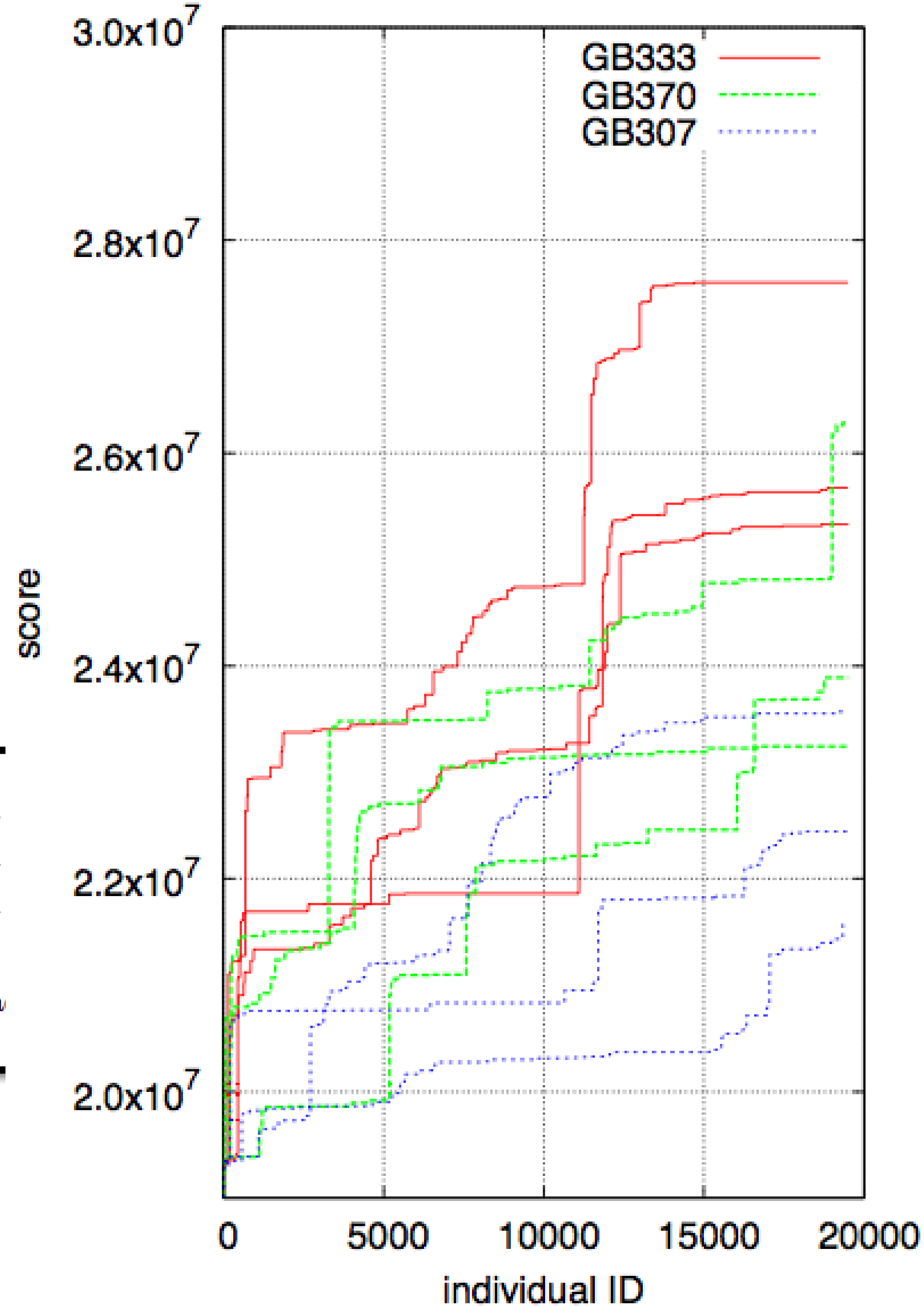
Table 4. The probability of the master node attempting each method of experiment series GB-*.

Table 5. The statistics of *GB* experiment series. The columns are RunID, precision, the score of initial individual, the wall-clock time for the experiment (in minutes), the ID of the best individual and the number of individuals generated, the highscore (in Mcups). Experiments started with *Shinatsuhiko*.

both crossover and triangulation are important!

	mutation	crossover	triangulation
GB-333	1/3	1/3	1/3
GB-370	1/3	2/3	0
GB-307	1/3	0	2/3

able 4. The probability of the master node attempting ea
periment series GB-*.



Which part of family tree are the methods of birth contributing?

$d(I)$	mutation	crossover	triangulation	total
0	785(0.023)	1099(0.071)	1680(0.087)	3565(0.052)
1	16113(0.482)	6208(0.403)	9699(0.503)	32020(0.470)
2	11510(0.344)	6946(0.451)	7490(0.389)	25946(0.381)
3	4509(0.135)	1139(0.074)	408(0.021)	6056(0.089)
4	472(0.014)	13(0.001)	1(0.000)	486(0.007)
5	21(0.001)	0(0.000)	0(0.000)	21(0.000)
6	2(0.000)	0(0.000)	0(0.000)	2(0.000)
sum	33412(1.000)	15405(1.000)	19278(1.000)	68096(1.000)

Table 13. Contribution distance analysis for experiment GA-D.

Crossovers and triangulations contributes more directly in generating the champion's family tree, and triangulations contributes the more.

How do children's scores compare with their parents'?

mutation			crossover				triangulation			
33420(1.000)			15412(1.000)				19261(1.000)			
[≪]	[≃]	[≫]	[≪]	[≤]	[≃]	[≫]	[≪]	[≤]	[≃]	[≫]
30112	2510	788	4110	5694	4657	944	3899	8372	6382	625
(0.901	0.075	0.024)	(0.267	0.370	0.302	0.061)	(0.202	0.434	0.331	0.032)
420	313	52	122	204	648	125	90	370	1134	86
(0.013	0.009	0.002)	(0.008	0.013	0.042	0.008)	(0.005	0.019	0.059	0.004)
0.014	0.125	0.066	0.030	0.036	0.139	0.132	0.023	0.044	0.178	0.138

Table 20. Tombi-Taka analysis for Experiment GA-D.

statistic significance of these statements analysed ...

$f_1(I)$	$f_2(I)$	$f_B(I)$	GA-1	GA-S1	GA-DE	GA-D	GA-4	GA-F	GA-F2
$n(\mathbb{P}(I)) = 1$	$d(I) = 0$	True	1678.37$\ominus$	176.82$\ominus$	195.35$\ominus$	1101.14$\ominus$	228.43$\ominus$	233.27$\ominus$	646.90$\ominus$
$n(\mathbb{P}(I)) = 2$	$d(I) = 0$	True	352.27$\oplus$	0.00$\ominus$	40.03$\oplus$	144.68$\oplus$	22.07$\oplus$	31.02$\oplus$	32.89$\oplus$
$n(\mathbb{P}(I)) = 3$	$d(I) = 0$	True	736.92$\oplus$	215.63$\oplus$	92.78$\oplus$	656.16$\oplus$	136.57$\oplus$	145.75$\oplus$	552.01$\oplus$
$I \in [\gg]$	$d(I) = 0$	True	3086.23$\oplus$	193.47$\oplus$	11.85$\oplus$	172.65$\oplus$	109.81$\oplus$	50.53$\oplus$	97.78$\oplus$
$I \in [\simeq]$	$d(I) = 0$	True	2384.30$\oplus$	1766.64$\oplus$	1429.42$\oplus$	3566.00$\oplus$	1745.43$\oplus$	1513.70$\oplus$	1698.94$\oplus$
$I \in [\leq]$	$d(I) = 0$	True	8.22$\ominus$	291.79$\ominus$	0.08$\ominus$	47.13$\ominus$	50.13$\ominus$	16.82$\ominus$	0.05$\ominus$
$I \in [\ll]$	$d(I) = 0$	True	6373.92$\ominus$	1482.62$\ominus$	1314.95$\ominus$	2233.94$\ominus$	1283.65$\ominus$	923.74$\ominus$	1162.42$\ominus$
$n(\mathbb{P}(I)) = 2$	$d(I) = 0$	$n(\mathbb{P}(I)) \geq 2$	0.50$\ominus$	62.38$\ominus$	1.06$\ominus$	29.02$\ominus$	12.05$\ominus$	7.15$\ominus$	40.75$\ominus$
$n(\mathbb{P}(I)) = 3$	$d(I) = 0$	$n(\mathbb{P}(I)) \geq 2$	0.50$\oplus$	62.38$\oplus$	1.06$\oplus$	29.02$\oplus$	12.05$\oplus$	7.15$\oplus$	40.75$\oplus$
$n(\mathbb{P}(I)) = 2$	$I \in [\gg]$	$I \in \mathbb{E}$	410.39$\oplus$	31.79$\oplus$	13.00$\oplus$	179.86$\oplus$	64.28$\oplus$	18.81$\oplus$	144.85$\oplus$
$n(\mathbb{P}(I)) = 3$	$I \in [\gg]$	$I \in \mathbb{E}$	410.39$\ominus$	31.79$\ominus$	13.00$\ominus$	179.86$\ominus$	64.28$\ominus$	18.81$\ominus$	144.85$\ominus$
$n(\mathbb{P}(I)) = 2$	$I \in [\gg]$	$d(I) = 0$	69.73$\oplus$	17.64$\ominus$	3.72$\oplus$	37.14$\oplus$	10.15$\oplus$	0.26$\oplus$	17.27$\oplus$
$n(\mathbb{P}(I)) = 2$	$I \in [\simeq]$	$d(I) = 0$	177.77$\ominus$	0.11$\oplus$	1.20$\ominus$	0.03$\oplus$	0.72$\ominus$	4.60$\oplus$	1.43$\oplus$
$n(\mathbb{P}(I)) = 3$	$I \in [\gg]$	$d(I) = 0$	340.94$\ominus$	19.05$\ominus$	2.49$\ominus$	23.71$\ominus$	9.29$\ominus$	3.77$\ominus$	10.90$\ominus$
$n(\mathbb{P}(I)) = 3$	$I \in [\simeq]$	$d(I) = 0$	368.68$\oplus$	22.80$\oplus$	7.69$\oplus$	100.03$\oplus$	20.56$\oplus$	5.72$\oplus$	42.80$\oplus$
$I \equiv 7 \bmod 10$	$d(I) = 0$	True	1.73$\ominus$	1.38$\ominus$	0.41$\ominus$	1.00$\ominus$	0.98$\oplus$	0.00$\ominus$	0.02$\ominus$
$I \equiv 13 \bmod 100$	$d(I) = 0$	True	1.05$\oplus$	0.69$\oplus$	0.06$\oplus$	0.08$\ominus$	0.43$\ominus$	0.08$\oplus$	1.11$\oplus$

Table 8. Chi-squared test of statistical independence of predicates. For each pair of experiment (columns) and three predicates $f_1(I), f_2(I), f_B(I)$, the table shows the X^2 statistics of the null hypothesis "predicates $f_1(I)$ and $f_2(I)$ are statistically independent for the poulation of individuals that satisfy predicate $f_B(I)$. " Here, $\mathbb{E} \equiv \{I | n(\mathbb{P}(I)) \geq 2\} \cap ([\gg] \cup [\simeq])$. $\oplus$ denotes the positive correration and $\ominus$ denotes the negative correration.

Family trees as Markov chains

RunID	0th order	1st order	$2 \to 2$	$3 \to 3$	$22 \to 2$	$33 \to 3$
GA-1	2263.22	266.28	⊖118.86	⊕1655.46	⊕32.54	⊕71.54
GA-S1	1387.93	70.51	⊖23.98	⊕1075.96	⊖5.19	⊕7.84
GA-DE	546.42	43.31	⊕3.34	⊕427.88	⊖9.85	⊕3.68
GA-D	1038.15	88.20	⊖42.78	⊕811.09	⊕3.90	⊕1.34
GA-4	755.63	39.91	⊖7.98	⊕580.33	⊖2.09	⊖2.60
GA-F	422.08	22.24	⊖2.07	⊕333.57	⊕0.96	⊖0.25
GA-F2	490.90	86.34	⊖23.63	⊕381.72	⊕16.29	⊕6.09
GB-333-0	666.18	47.52	⊖12.34	⊕511.62	⊕1.36	⊖2.52
GB-333-1	930.33	25.26	⊖48.06	⊕727.01	⊖0.86	⊖0.90
GB-333-2	1208.20	68.11	⊖39.34	⊕937.37	⊕0.34	⊖7.59

Table 7. Chi-squared test of the family tree being lower-order Markov processes. The each column of the table shows the X^2 statistics of the null hypothesis the family tree being a n-th order Markov process and having no longer correlation.

Summary : three methods of birth

- Mutations : not efficient in making good species, but the only way of introducing new genomes
- Crossover : good at making large jumps
- Triangulations : good at accumulating small improvements

Current implementation of Paraiso has three things to tune:

- C: cuda configuration `<<<NT,NB>>>`
- M:Manifest/Delay
- S: `__syncthreads()`

- how are their contribuitions?

C: cuda configuration `<<<NT,NB>>>`

M:Manifest/Delay S: `__syncthreads()`

ID	C	M	S	score(Mcups)	relative score	logscale
Izanagi	0	0	0	1.137 ± 0.003	0.000 ± 0.000	0.000 ± 0.001
	0	0	1	1.122 ± 0.000	-0.001 ± 0.000	-0.005 ± 0.000
	0	1	0	5.400 ± 0.006	0.300 ± 0.000	0.599 ± 0.000
	0	1	1	5.300 ± 0.006	0.293 ± 0.000	0.591 ± 0.000
	1	0	0	3.073 ± 0.002	0.136 ± 0.000	0.382 ± 0.000
	1	0	1	2.946 ± 0.000	0.127 ± 0.000	0.366 ± 0.000
	1	1	0	15.829 ± 0.027	1.033 ± 0.002	1.012 ± 0.001
GA-S1.33958	1	1	1	15.354 ± 0.020	1.000 ± 0.001	1.000 ± 0.001

Table 6. The score of the individuals created by artificial crossover between the initial individual I_0 and the best scoring individual $I_\top$. The second to fourth columns indicate which component was taken from which individual. Columns C,M,S correspond to CUDA kernel execution configuration, Manifest/Delay choice, synchronization timing, respectively. For each individual I the fifth column shows $\mu(I) \pm \sigma(I)$, the sixth column shows $\dfrac{\mu(I)}{\mu(I_\top) - \mu(I_0)} \pm \dfrac{\sigma(I)}{\mu(I_\top) - \mu(I_0)}$, and the seventh column shows $\dfrac{\log \mu(I) - \log \mu(I_0)}{\log \mu(I_\top) - \log \mu(I_0)} \pm \dfrac{\sigma(I)}{(\log \mu(I_\top) - \log \mu(I_0))\mu(I)}$.

C: cuda configuration <<<NT,NB>>>
M:Manifest/Delay S: __syncthreads()

ID	C	M	S	score(Mcups)	relative score	logscale
Shinatsuhiko	0	0	0	19.808 ± 0.033	0.000 ± 0.002	0.000 ± 0.003
	0	0	1	19.817 ± 0.030	0.001 ± 0.002	0.001 ± 0.003
	0	1	0	32.821 ± 0.058	0.848 ± 0.004	0.880 ± 0.003
	0	1	1	32.694 ± 0.057	0.839 ± 0.004	0.873 ± 0.003
	1	0	0	19.773 ± 0.050	-0.002 ± 0.003	-0.003 ± 0.004
	1	0	1	19.859 ± 0.058	0.003 ± 0.004	0.005 ± 0.005
	1	1	0	32.994 ± 0.273	0.859 ± 0.018	0.889 ± 0.014
GA-4.33991	1	1	1	35.160 ± 0.082	1.000 ± 0.005	1.000 ± 0.004

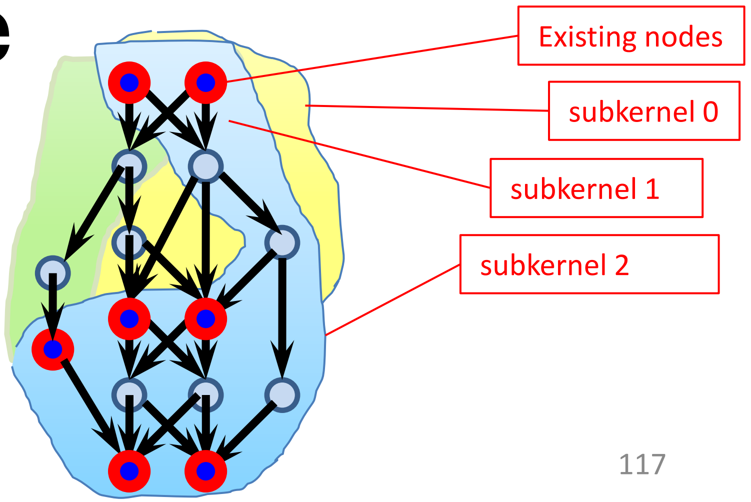
Table 6. The score of the individuals created by artificial crossover between the initial individual I_0 and the best scoring individual $I_\top$. The second to fourth columns indicate which component was taken from which individual. Columns C,M,S correspond to CUDA kernel execution configuration, Manifest/Delay choice, synchronization timing, respectively. For each individual I the fifth column shows $\mu(I) \pm \sigma(I)$, the sixth column shows $\dfrac{\mu(I)}{\mu(I_\top) - \mu(I_0)} \pm \dfrac{\sigma(I)}{\mu(I_\top) - \mu(I_0)}$, and the seventh column shows $\dfrac{\log \mu(I) - \log \mu(I_0)}{\log \mu(I_\top) - \log \mu(I_0)} \pm \dfrac{\sigma(I)}{(\log \mu(I_\top) - \log \mu(I_0))\mu(I)}$.

# of three things to tune:

- C: cuda configuration `<<<NT,NB>>>`
- M:Manifest/Delay
- S: `__syncthreads()`

- Manifest/Delay is the major source of speedup
- Config and Sync are nevertheless important, without them we lose at least 10–20% each.

- So Paraiso's GA is not just about optimizing a few parameters: it's really searching for better memory layouts, and by doing so found 2x faster solutions than those a human being (me) can think of.



automatically tuned codes v.s. hand-optimized codes by others

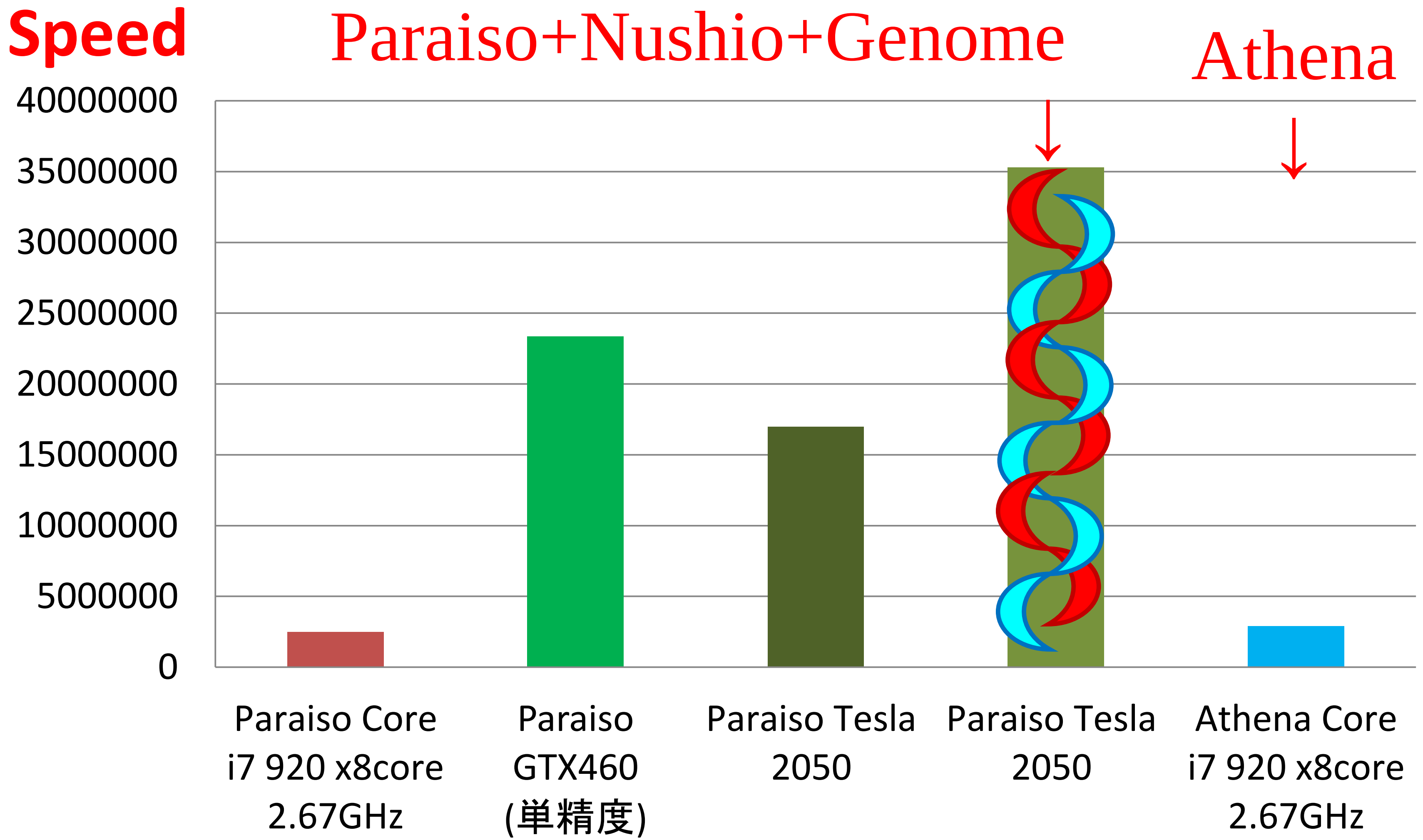
The automated tuning system can generate and benchmark approximately $10'000$ individual per day. $20 - 100$ workers were running at the same time. It takes a few days to tune up *Izanami* to speed comparable to *Shinatsuhiko*, or speed up *Shinatsuhiko* by another factor of 2. The best speed obtained was 35.3Mcups for double precision, and 53.7Mcups for single precision. Our autotuning experiments on 3D solvers mark 42.4Mcups SP. These are competitive performances to hand-tuned codes for single GPUs; e.g. Schive et. al. [29] reports 30Mcups per C2050 card (single precision, note that their code is 3D. Asuncióna et.al. [30] reports 6.8Mcups per GTX580 card (single precision, 2D).

All you need to change your 2D code to 3D code

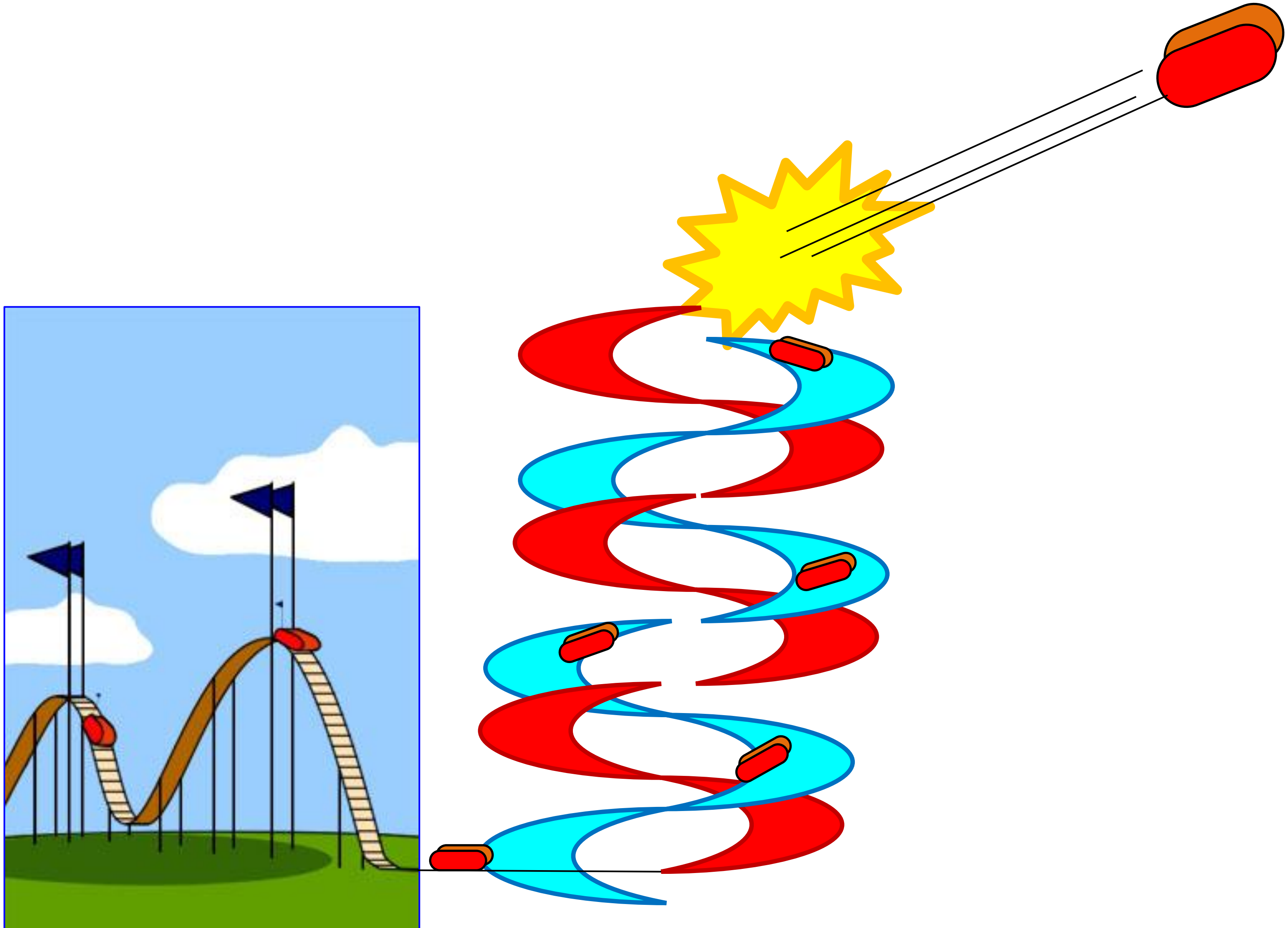
```haskell
-------------------------------------------
-- Binder monad utilities
-------------------------------------------

type Real = Double
type Dim = Vec2
type B a = Builder Dim Int Annotation a
type BR = B (Value TLocal Real)
type BGR = B (Value TGlobal Real)

bind :: B a -> B (B a)
bind = fmap return
```

```haskell
-------------------------------------------
-- Binder monad utilities
-------------------------------------------

type Real = Double
type Dim = Vec3
type B a = Builder Dim Int Annotation a
type BR = B (Value TLocal Real)
type BGR = B (Value TGlobal Real)

bind :: B a -> B (B a)
bind = fmap return
```

Benchmark rev.3



What speed you get rev.3



Current State of Paraiso (1/2)

- Can write explicit solvers of PDE using abstract, mathematical, combinable and reusable notations.
- Can generate OpenMP and CUDA program for multicore CPUs as well as GPUs
- On 8-core CPU, the speed of OpenMP version almost matches that of hand-written codes widely used.
- **CUDA version** is **10x** faster than them, and comes for free.

Current State of Paraiso (2/2)

- By adding just 1-2 lines of Annotation **by hand**, we can make radical changes on memory usage/computation structure of the code, resulting in radical change in performance of **6x-10x**.
- **Automated tuning** gives yet another **2x** speedup.

Future of Paraiso

This is not a victory; this is where the real fight begins.

- Distributed computation via MPI.
- Other native language backends ... OpenCL, Fortran and **Physis**!

to be continued...