check!

http://paraiso-lang.org/wiki/

Paraiso: an automated tuning framework for explicit solvers of partial differential equations

Takayuki Muranushi

Received 3 March 2012, in final form 26 April 2012
Published 7 June 2012
Computational Science & Discovery **5** (2012) 015003 (40pp)
doi:10.1088/1749-4699/5/1/015003

http://arxiv.org/abs/1204.4779

52 page paper

Takayuki Muranushi @nushio
Astrophysicist, Assistant professor at
The Hakubi Center, Kyoto University（2010-2015）

quick start guide

(I fixed it yesterday, working again!)

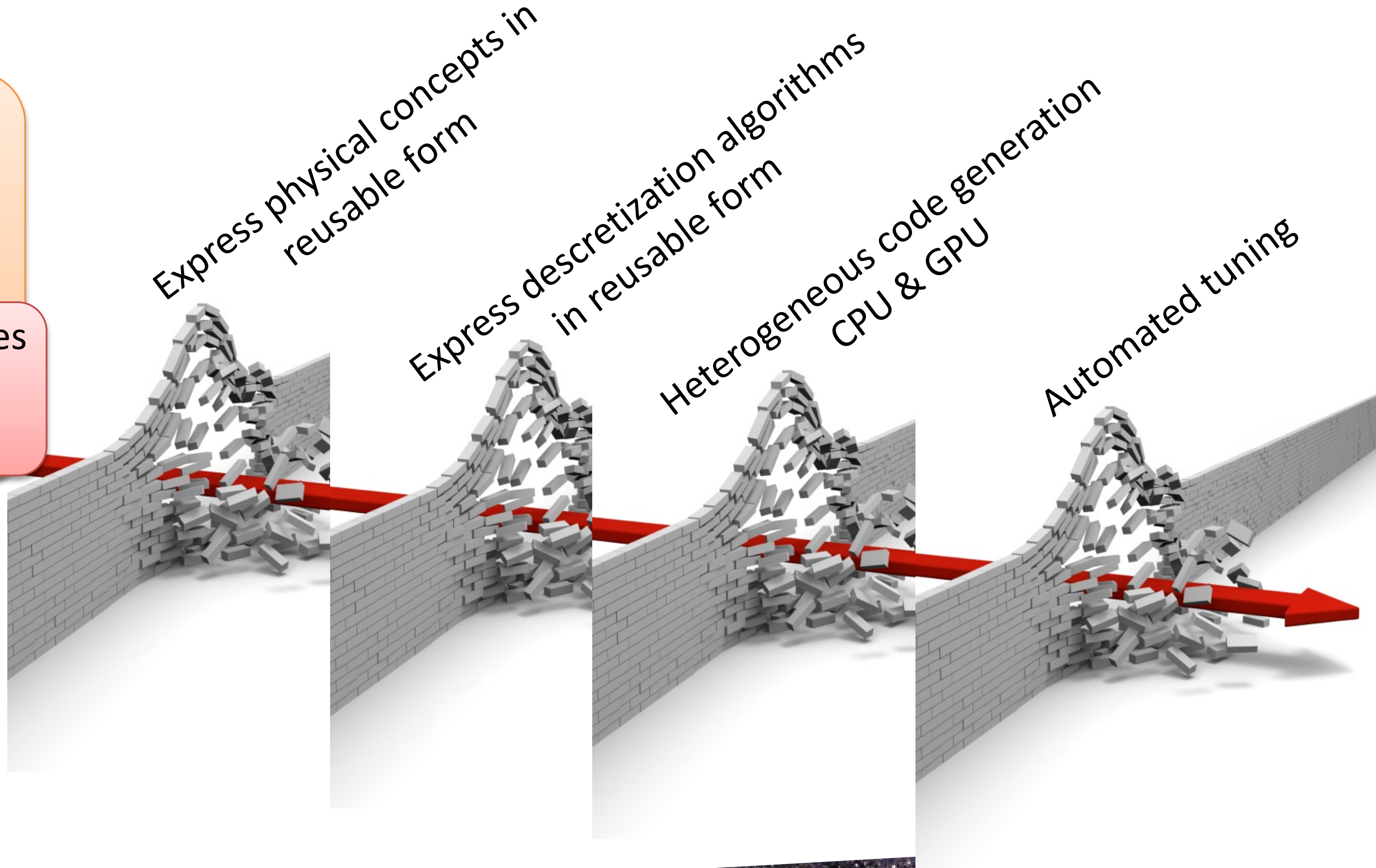
Install Haskell Platform and git, then type

```
> git clone git@github.com:nushio3/Paraiso.git
> cd Paraiso/
> cabal install
> cd examples/Life/        #Conway's game of life example
> make
> ls output/OM.txt
output/OM.txt              #OM dataflow graph
> ls dist/
Life.cpp  Life.hpp         #an OpenMP implementation
> ls dist-cuda/
Life.cu  Life.hpp          #a CUDA implementation
> ./main.out
```

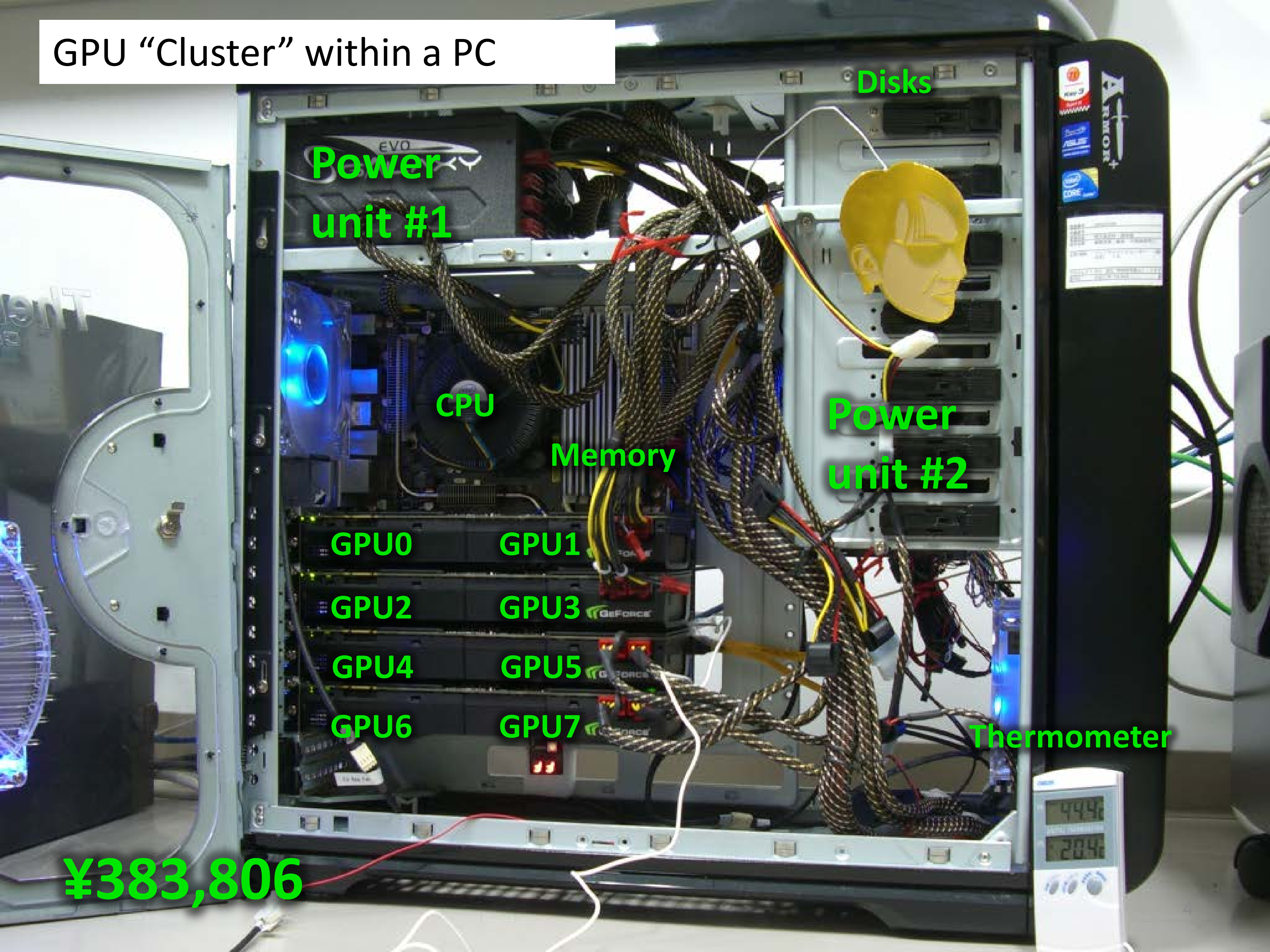
Took a specific domain,
took a very specific example,
broke through all walls

Partial differential equation solvers

Navier-Stokes Equation Solver

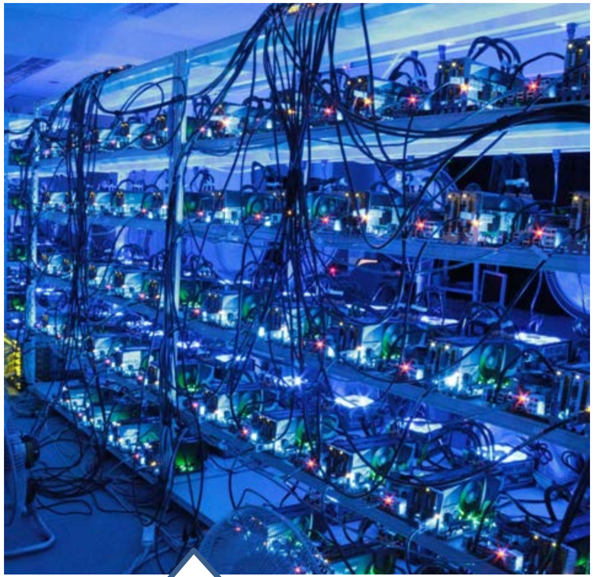


GPU “Cluster” within a PC



GPU Clusteres I have used so far

TenGU
Homebuilt, Kyoto-u



DEGIMA
Nagasaki Univ.

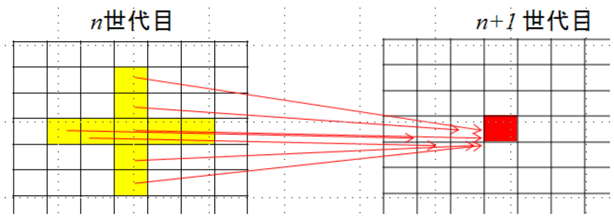
TSUBAME(1.2-2.0)
Tokyo institute of Tech.

1440^3 simulation of interstellar medium turbulence, visualised in 40-face display array in collab. with Oyamada Lab. Kyoto University

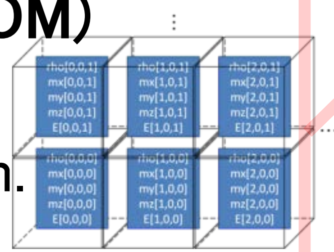
Paraiso Toolchain

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

Equations

manually

Discrete Algorithm

OM Builder

Orthotope Machine code

OM Compiler

Native Machine Source code

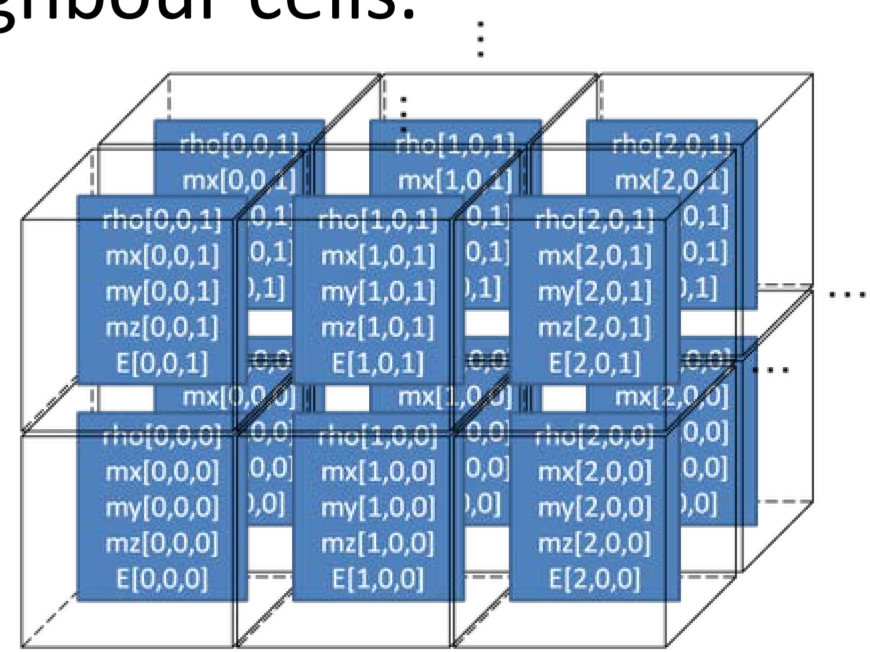
Native compiler

Executables

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

```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

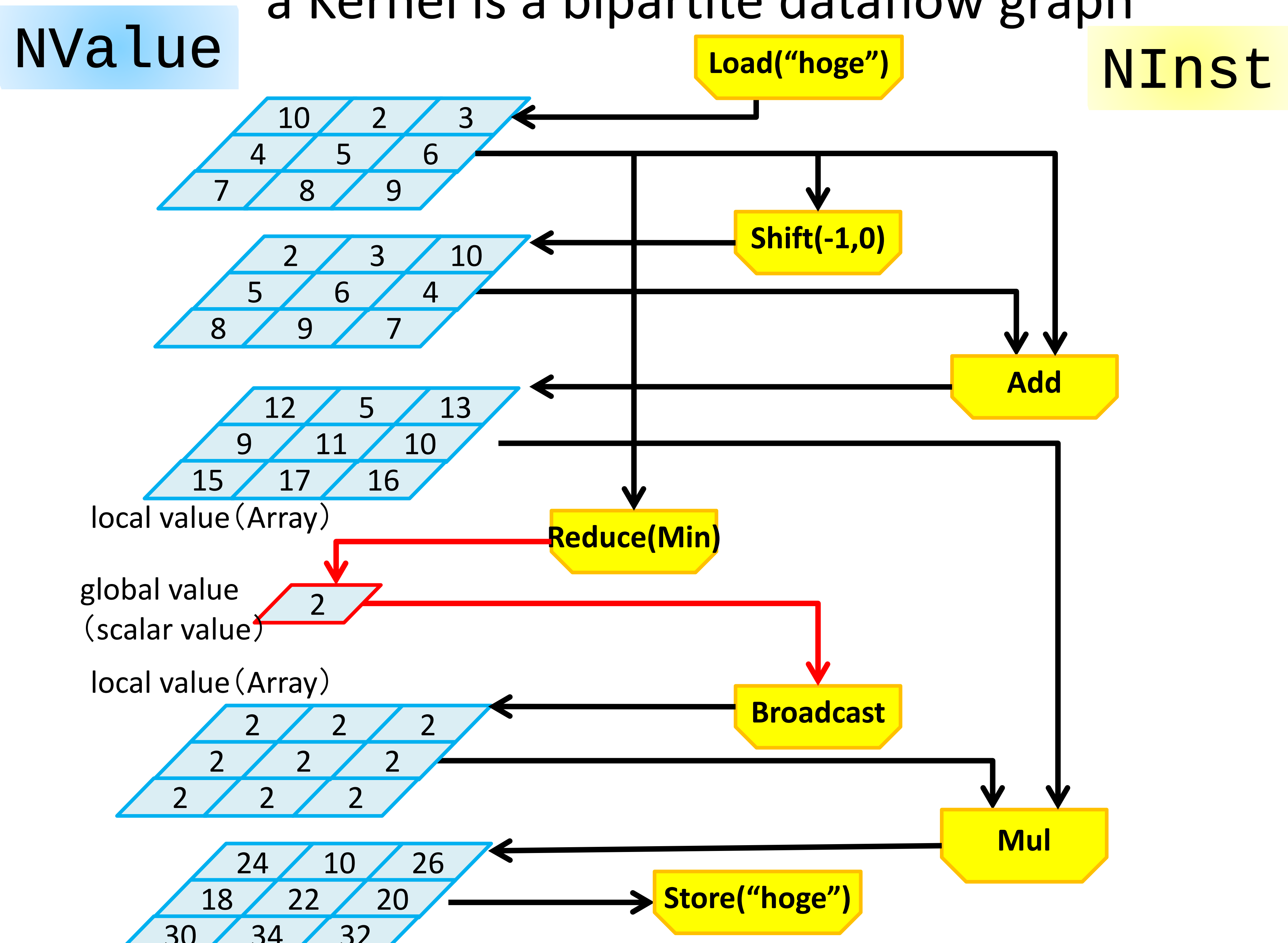
`Broadcast`
  scalar to array

`Shift`
  move each cell to neighbourhood

`Arith`
  various mathematical operations

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

a Kernel is a bipartite dataflow graph



Teach Haskell a hydrodynamics and tensor calculus and let him generate the dataflow graph

```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
  densityFlux  x = momentum x
  momentumFlux :: a -> Dim (Dim BR)
  momentumFlux x =
      compose (\i -> compose (\j ->
          momentum x !i * velocity x !j + pressure x * delta i j))
  energyFlux   :: a -> Dim BR
```

The frontend generate a dataflow graph on arrays that has 3958 nodes.

c.f. https://raw.githubusercontent.com/nushio3/Paraiso/master/examples-old/Hydro-exampled/output/OM.txt

That represents a solver of Navier-Stokes equation.

$$\rho_t + \nabla \cdot (\rho \mathbf{V}) = 0 \ ,$$

$$\frac{\partial}{\partial t}(\rho \mathbf{V}) + \nabla \cdot [\rho \mathbf{V} \otimes \mathbf{V} + pI - \Pi] = \rho \mathbf{g} \ ,$$

$$E_t + \nabla \cdot [(E + p)\mathbf{V} - \mathbf{V} \cdot \Pi + \mathbf{Q}] = \rho(\mathbf{V} \cdot \mathbf{g}) \ .$$

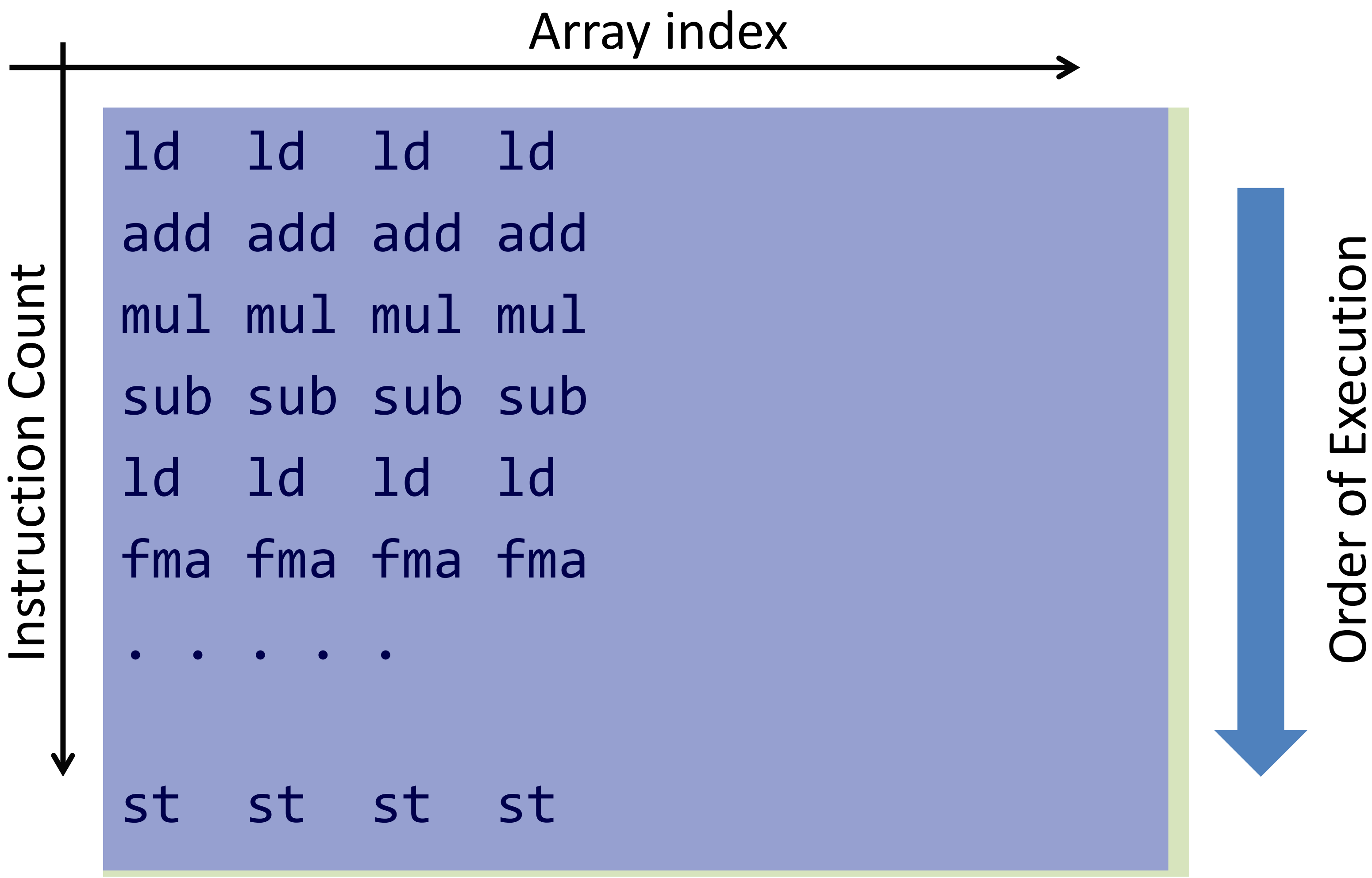
- of which 1908 nodes are fusion candidate
- 23186314741403598975944790941378166501633903963546171079785389729146769112962898895289498878984644779339098839938471655122333685680678398260291269160624836444577017233503954535729241917880311363490383137914861274921255128950712734788397408670521950919714209832229269791771351811195343521433399062351344722156320922220134647507093436286672888539484845152980307877955920545907395325548222694867051456609645215932758935244244579084816176470059329340736642337222850662358951938698298215645717772808920891115086440342006478637177469672403326343875446350241918444483542305006944256 different implementations are possible just for fusion

Tuning Target

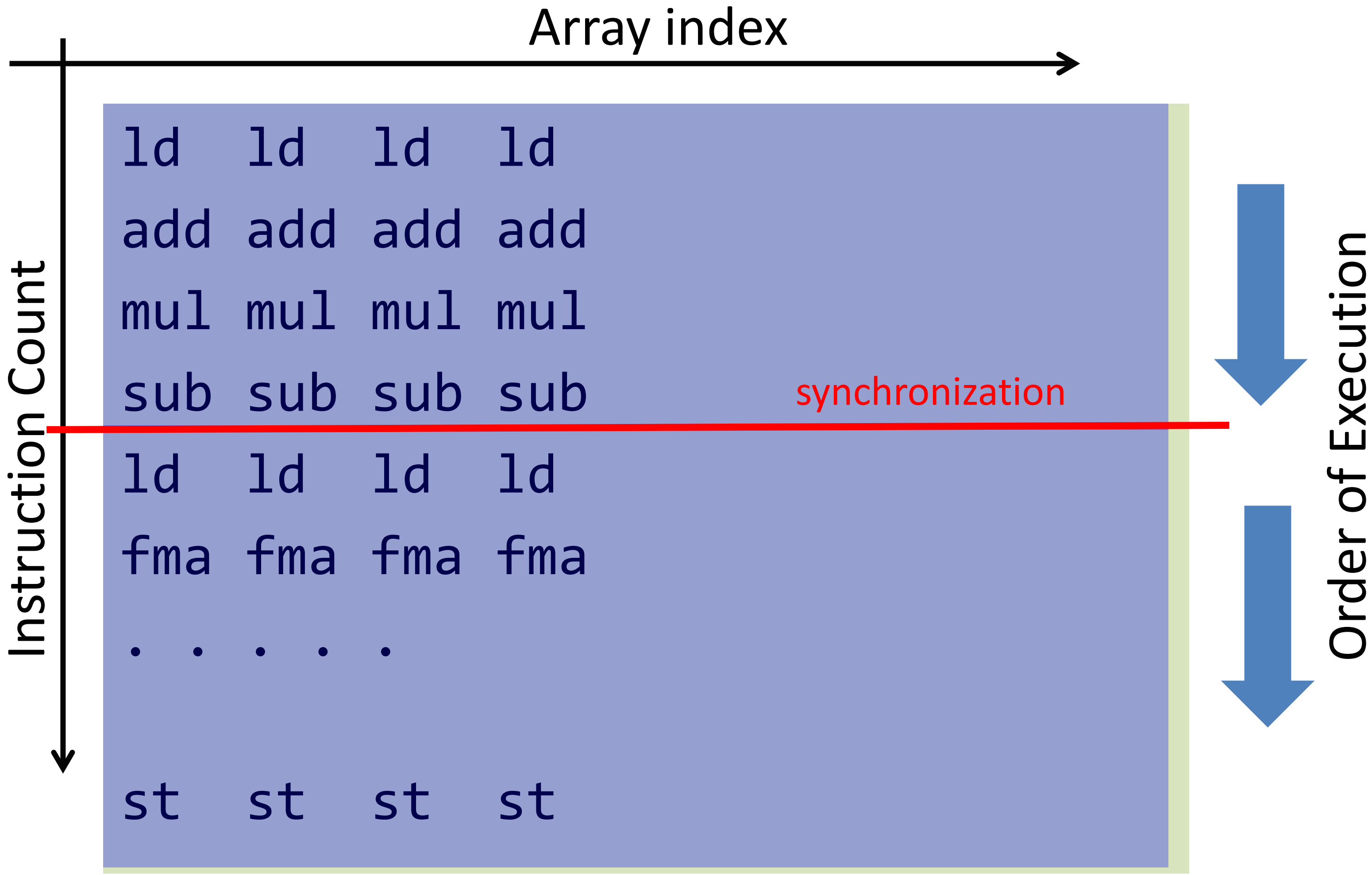
- C : cuda configuration `<<<NT,NB>>>`
- M : Manifest/Delay

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

Choice for syncronization



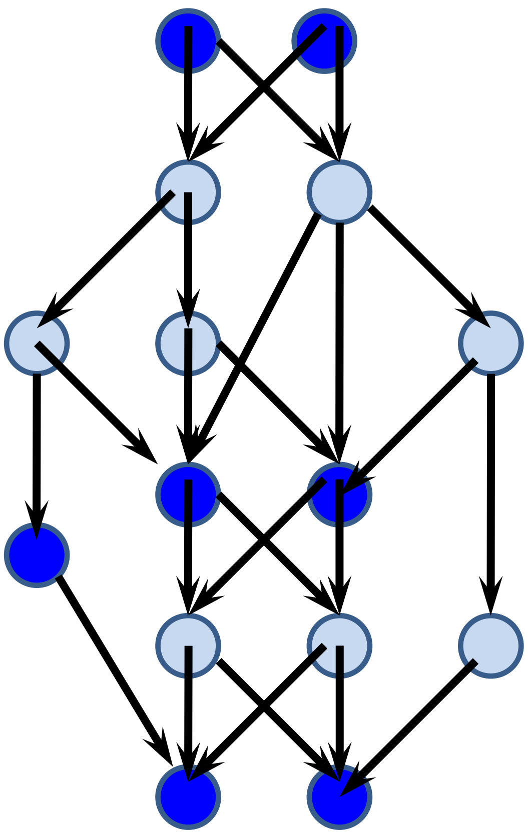
Choice for syncronization changes reuse pattern in the cache



Manifest/Delay selection

Names inherited from Repa (hackage.haskell.org/package/repa)

```
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)
```



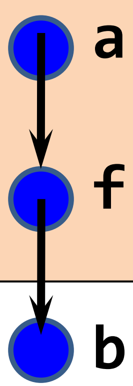
- some of the dataflow graph nodes are marked ‘Manifest.’
- Manifest nodes are stored in memory.
- Delayed nodes are re-computed as needed.

Fusion: which one better?

no one but benchmark knows

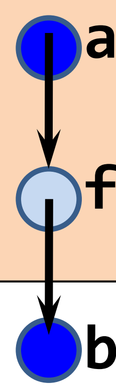
Less computation

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

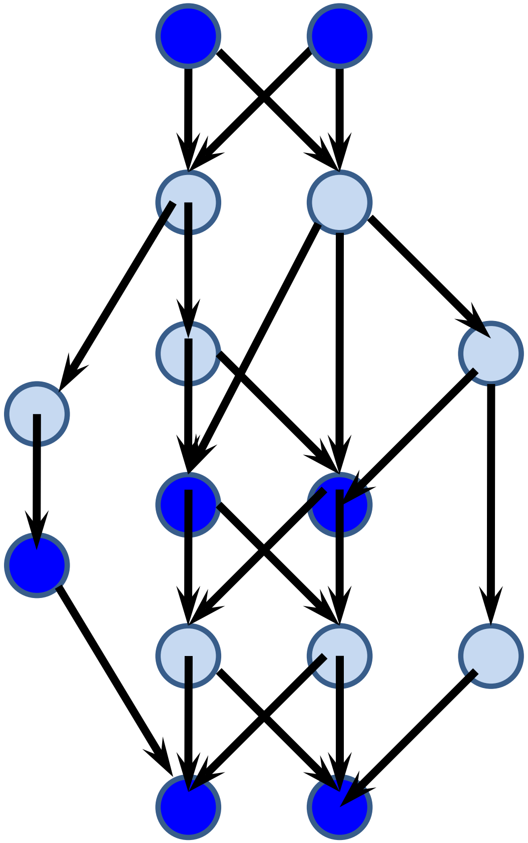


Less storage consumption & bandwidth

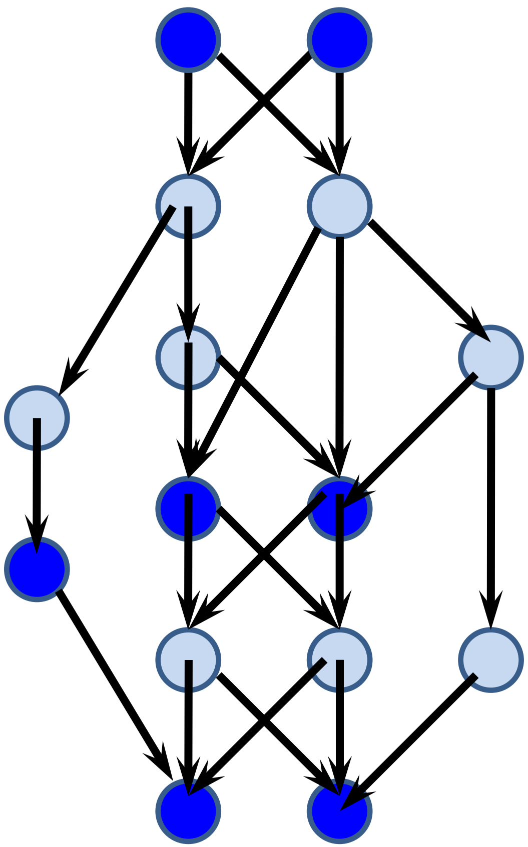
```
for(;;){
   f0 = calc_f(a[i-1], a[i]);
   f1 = calc_f(a[i], a[i+1]);
   b[i] +=  f1 – f0;
}
```



write grouping:
once manifest/delay is fixed

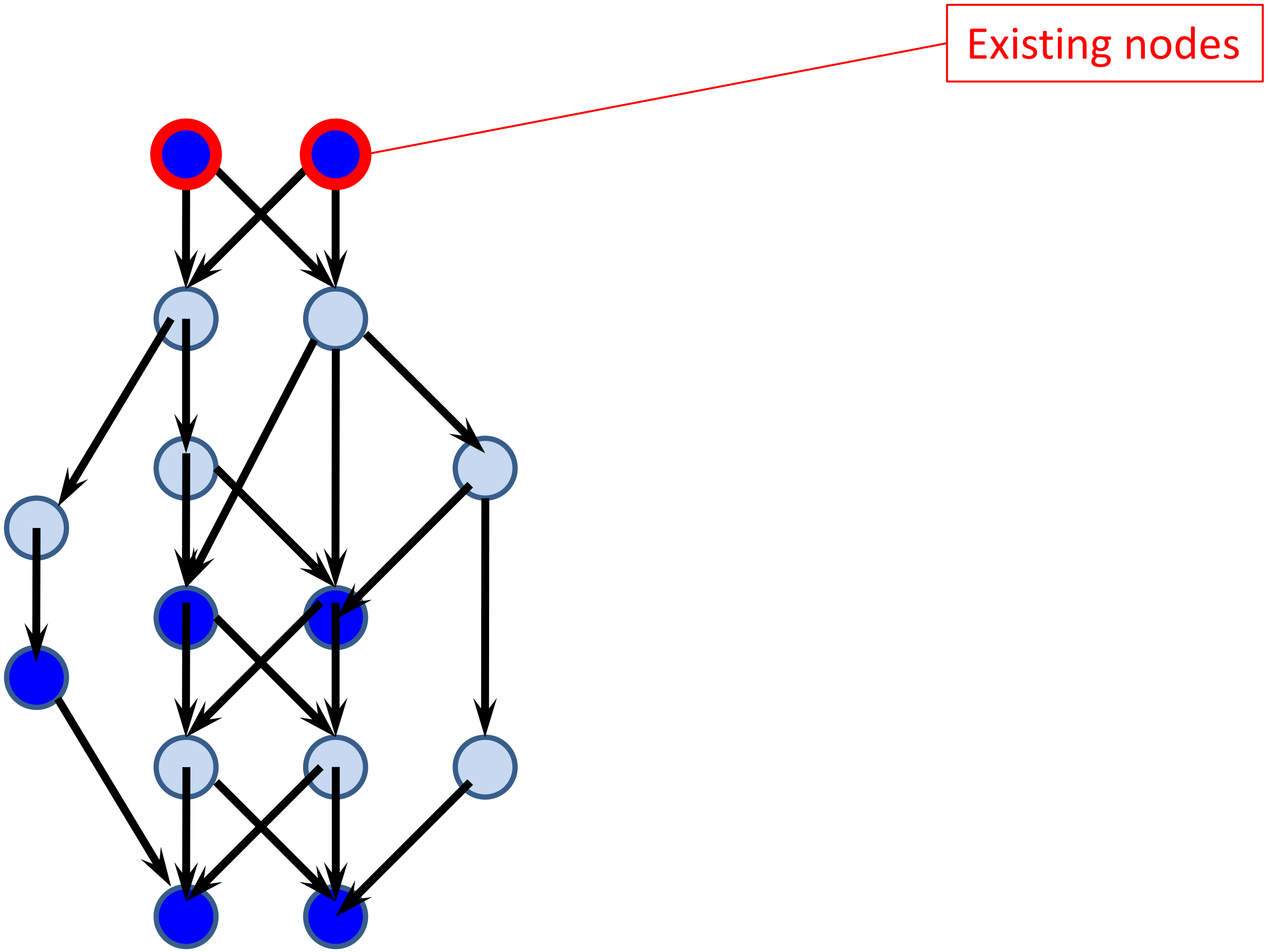


write grouping
= a Kernel -> subkernels

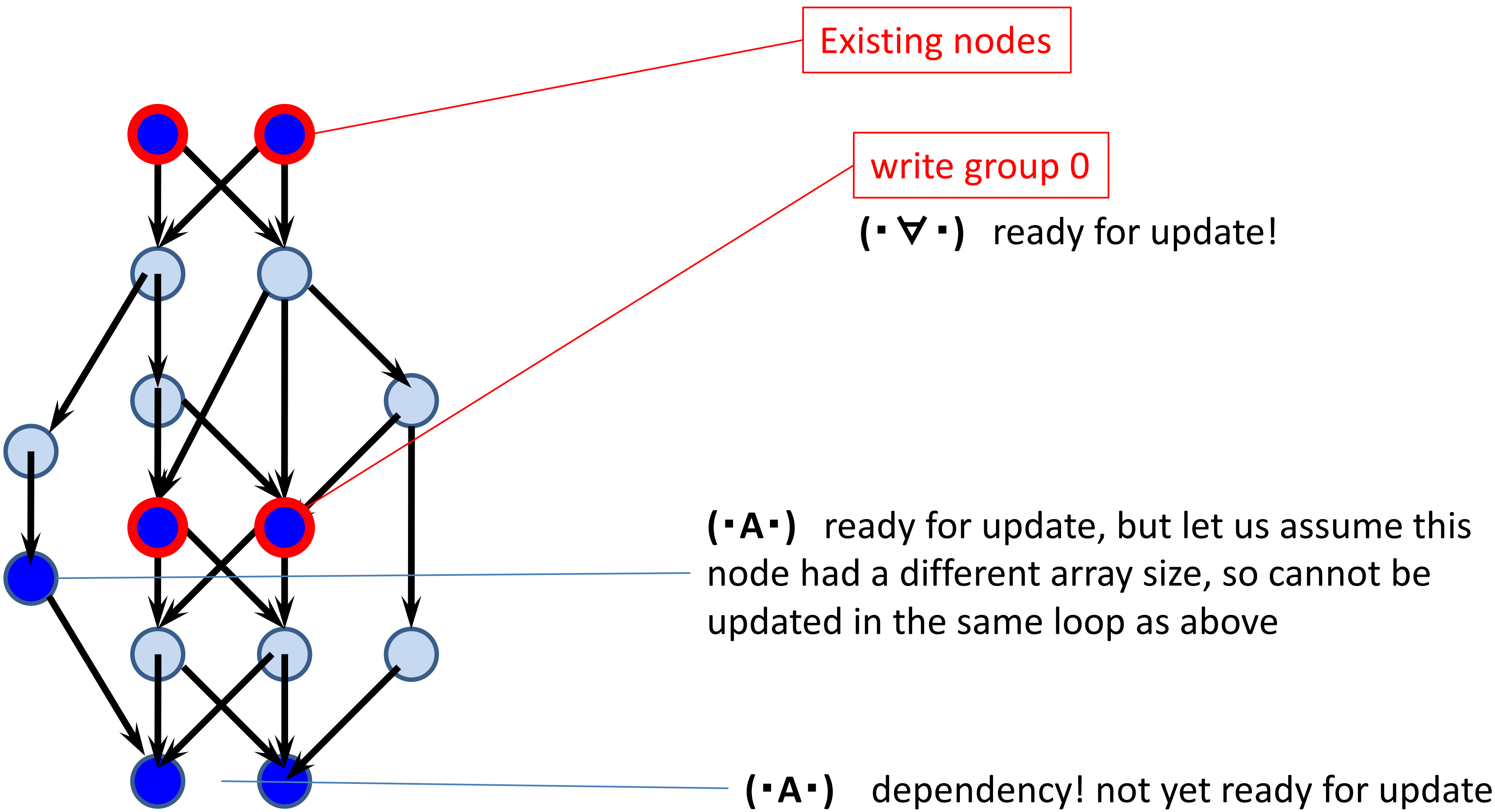


- all node written by one subkernel must have the same array size
- nodes written by one subkernel must not depend on each other
- greedy

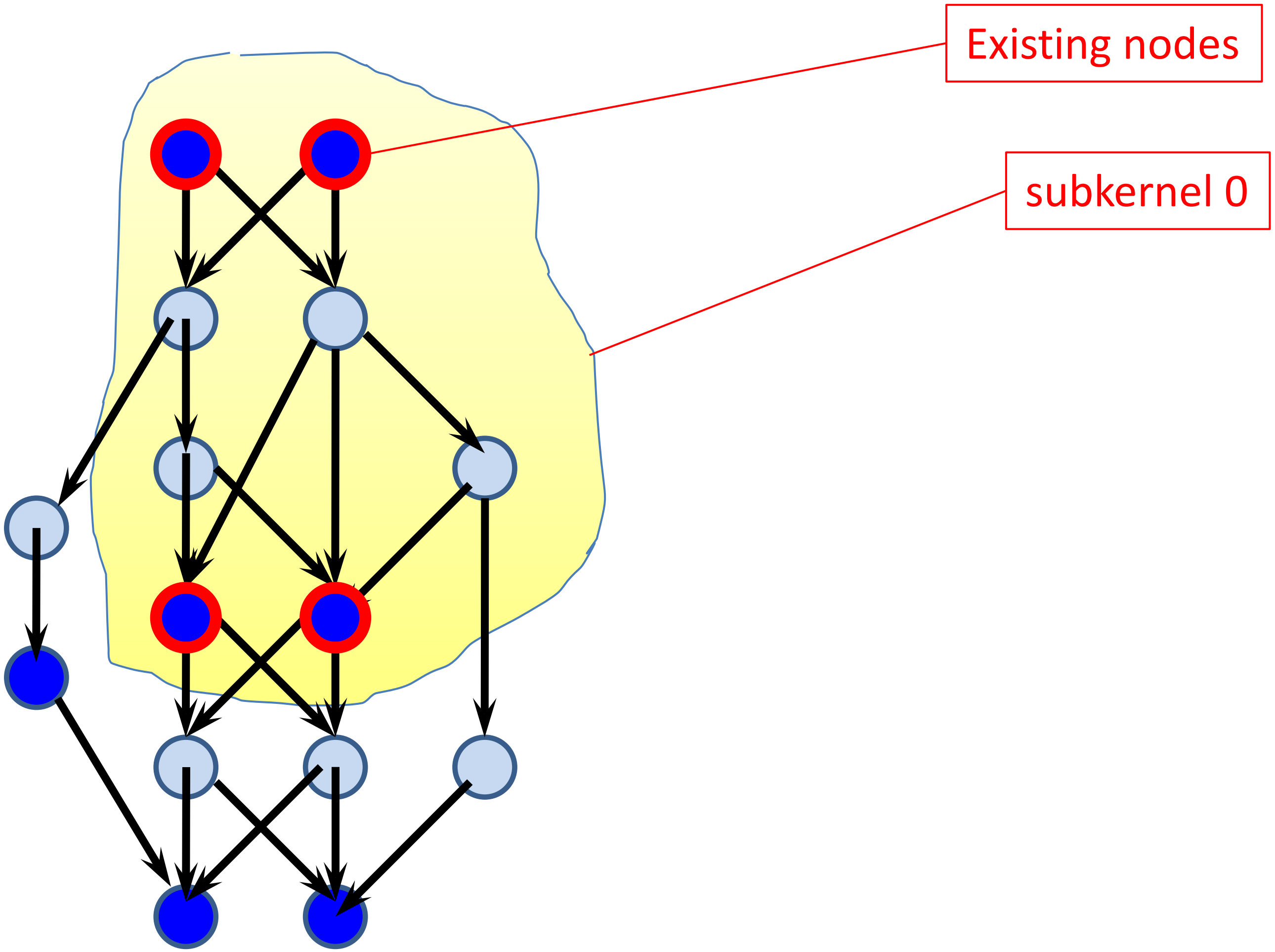
a Kernel



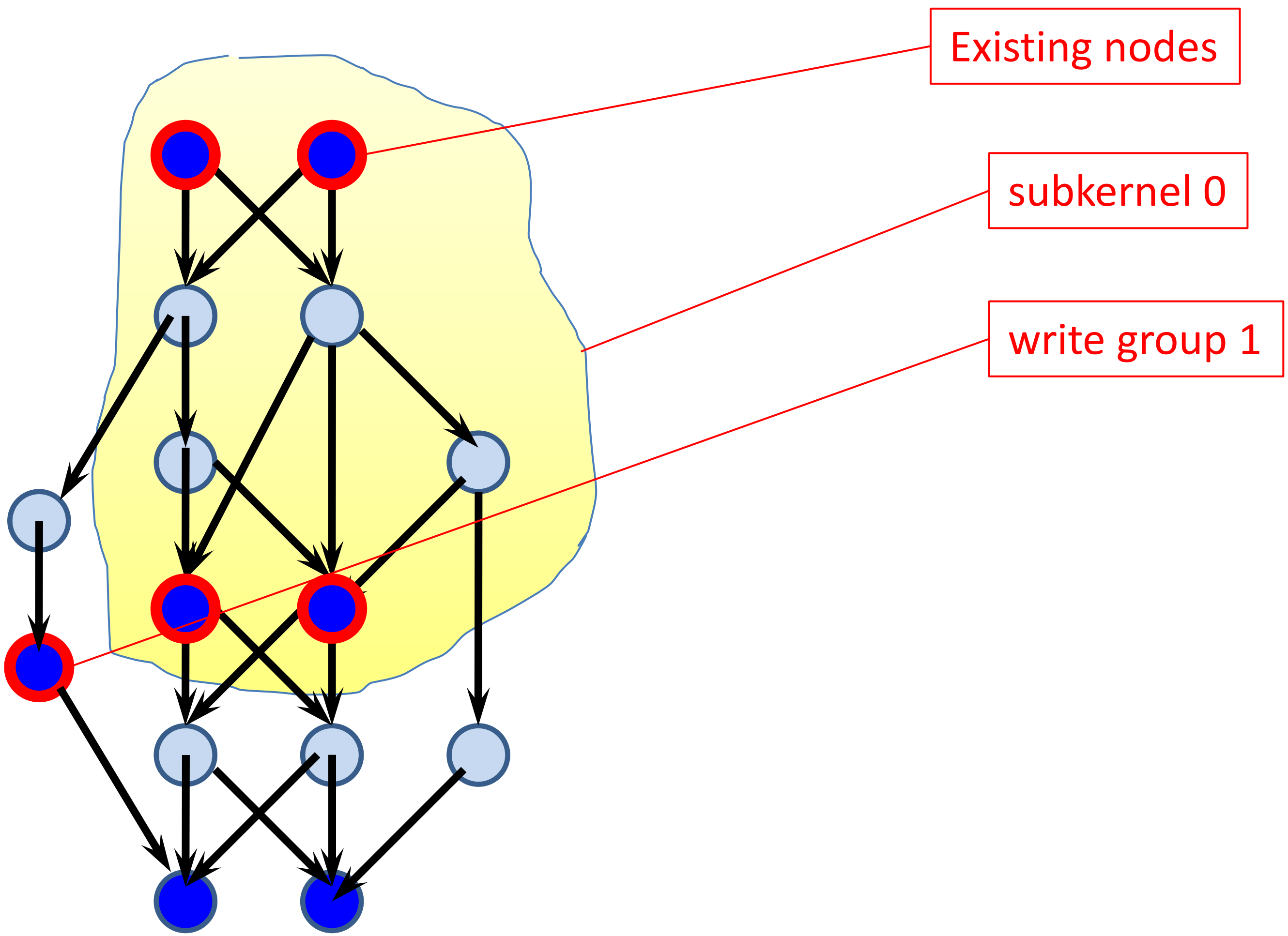
a Kernel


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

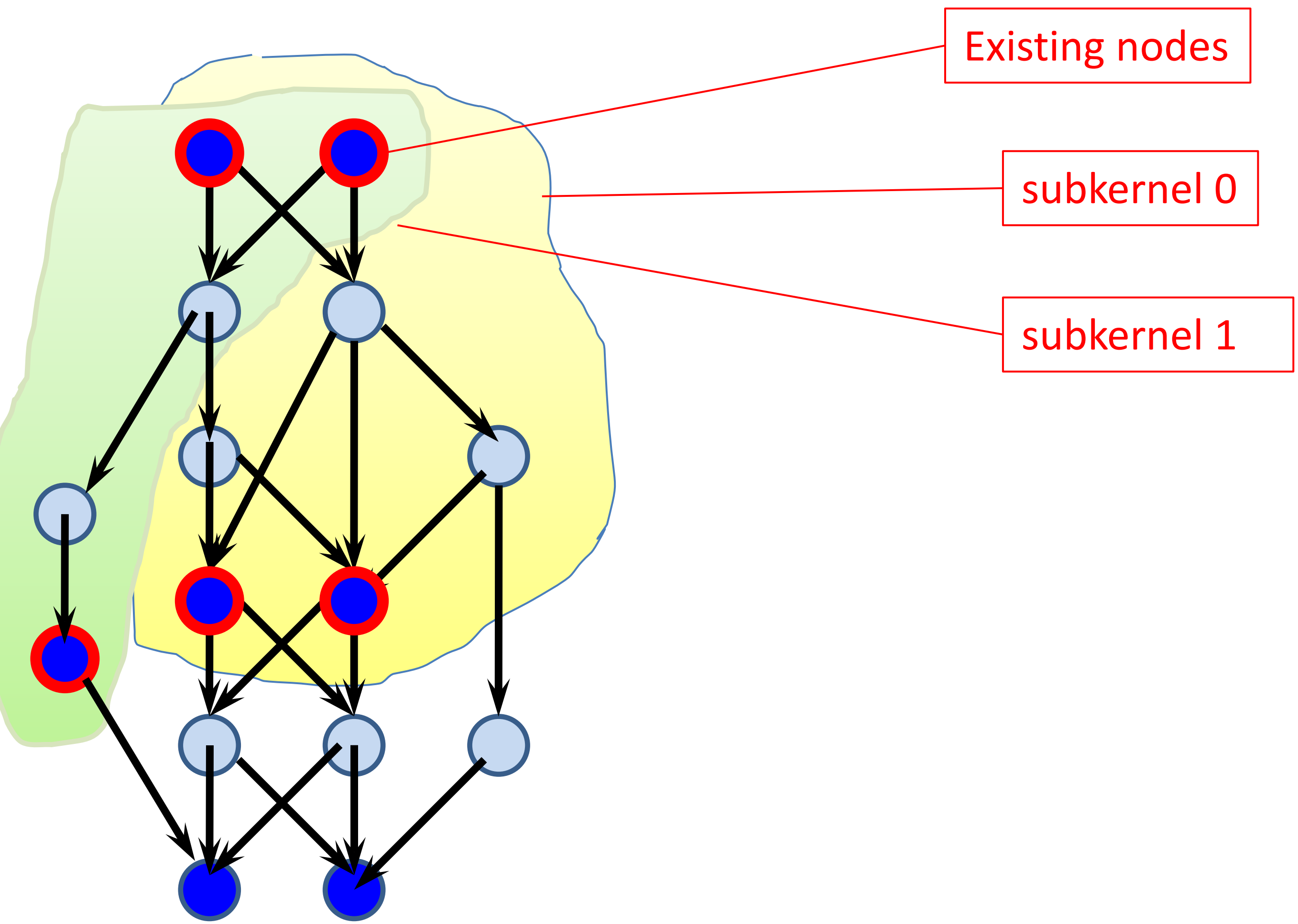
a Kernel



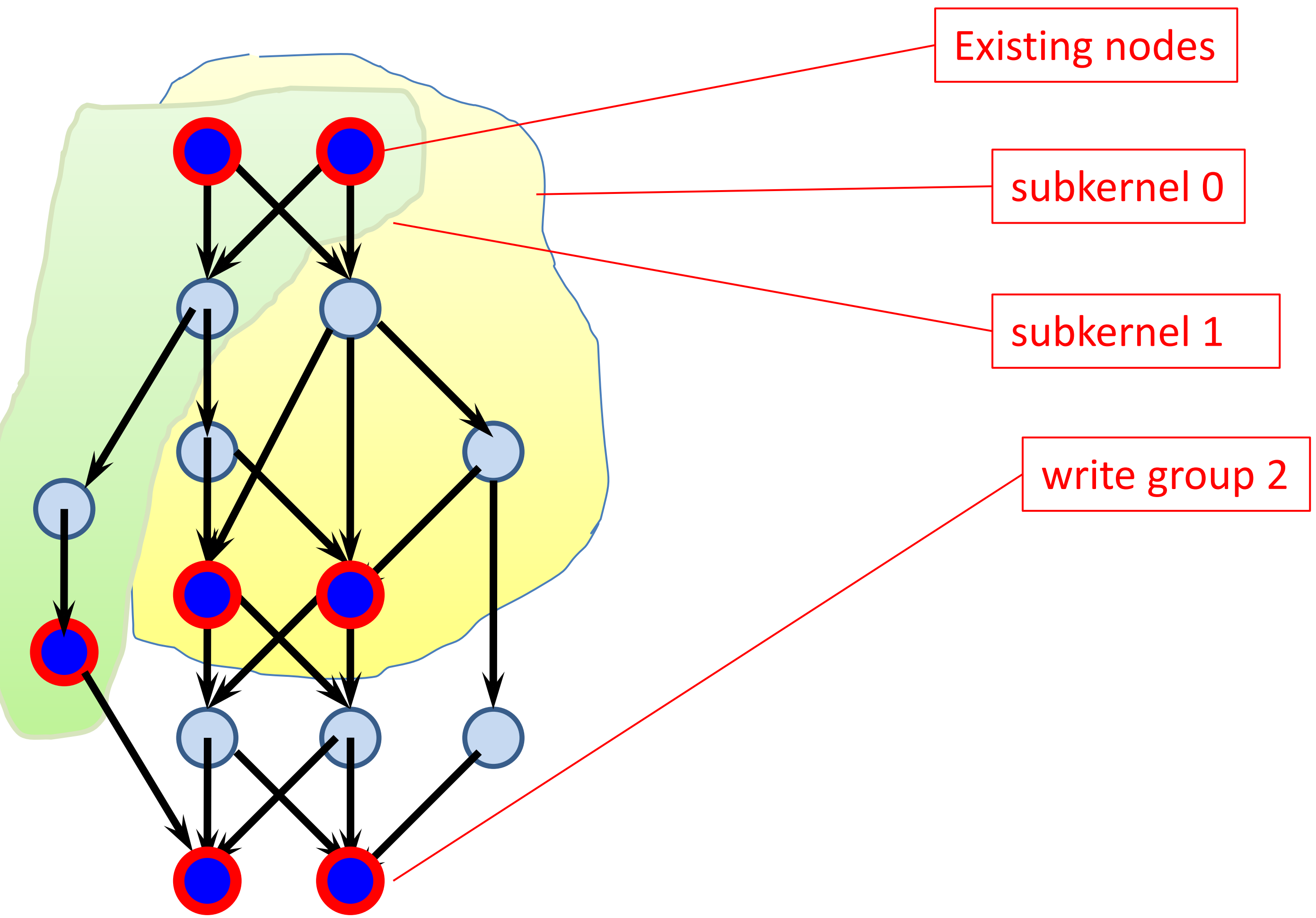
a Kernel



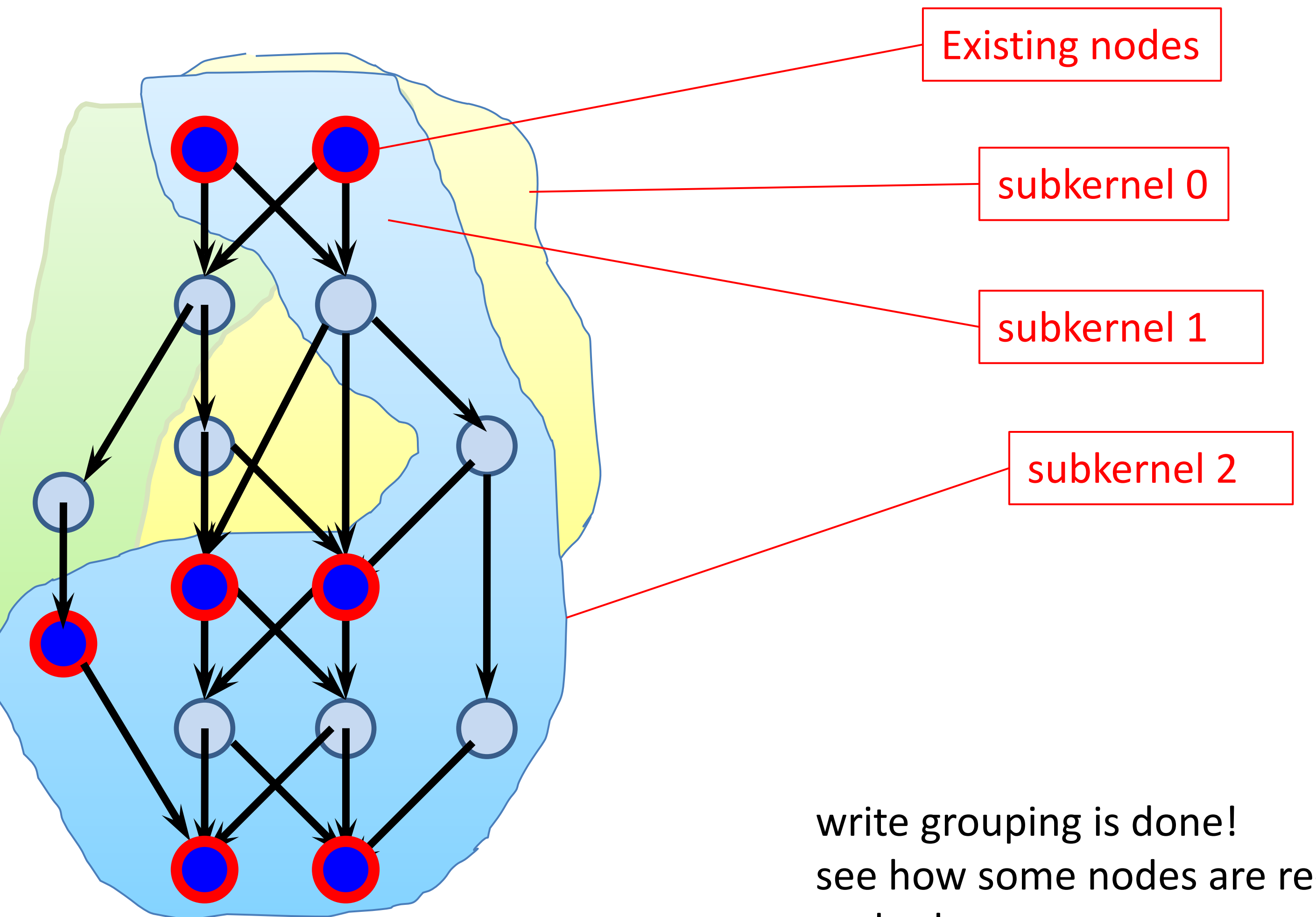
a Kernel



a Kernel

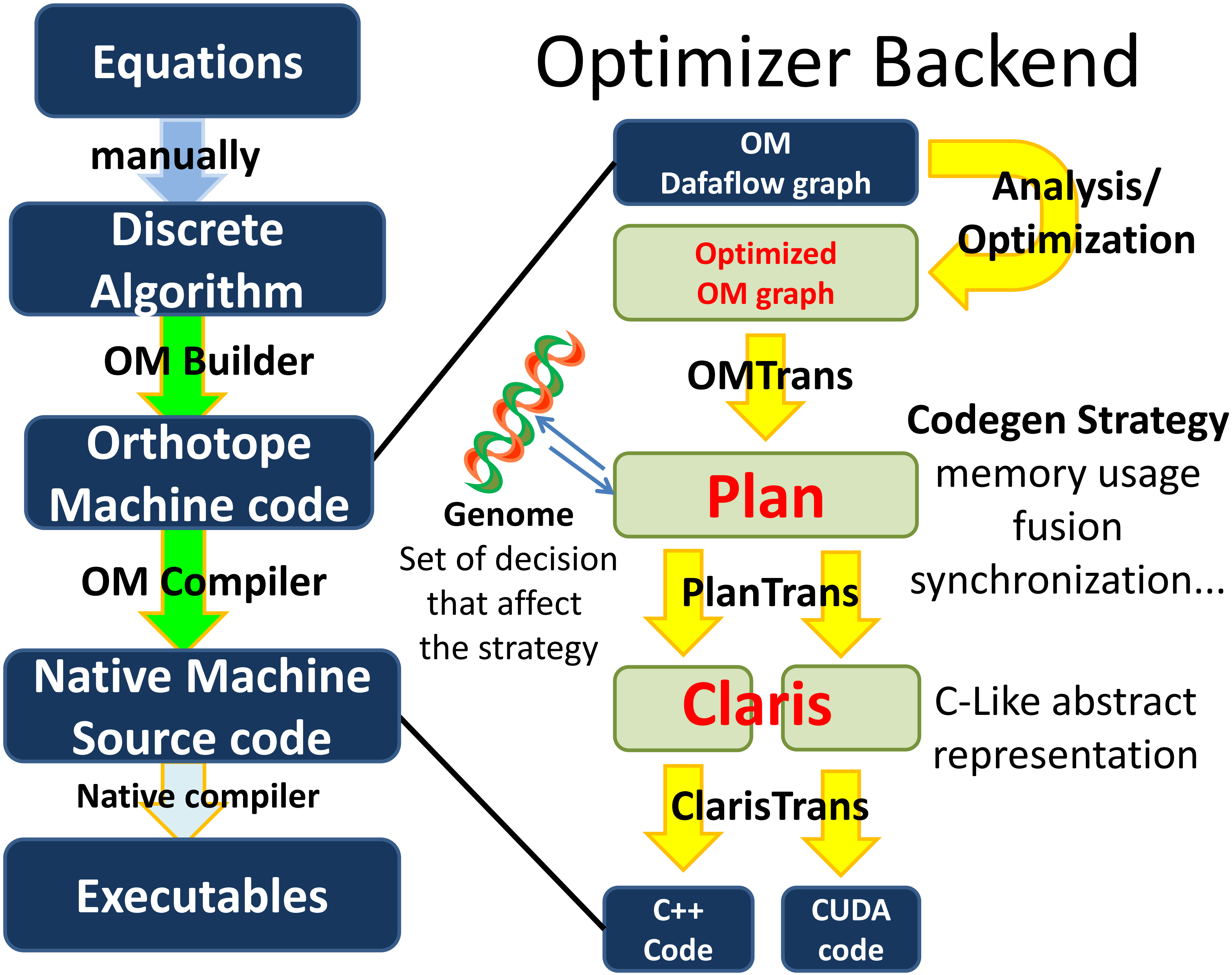


a Kernel



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

Optimizer Backend



Example of making a manual decision for a choice

```haskell
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"
```

```haskell
(<?>) :: (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.

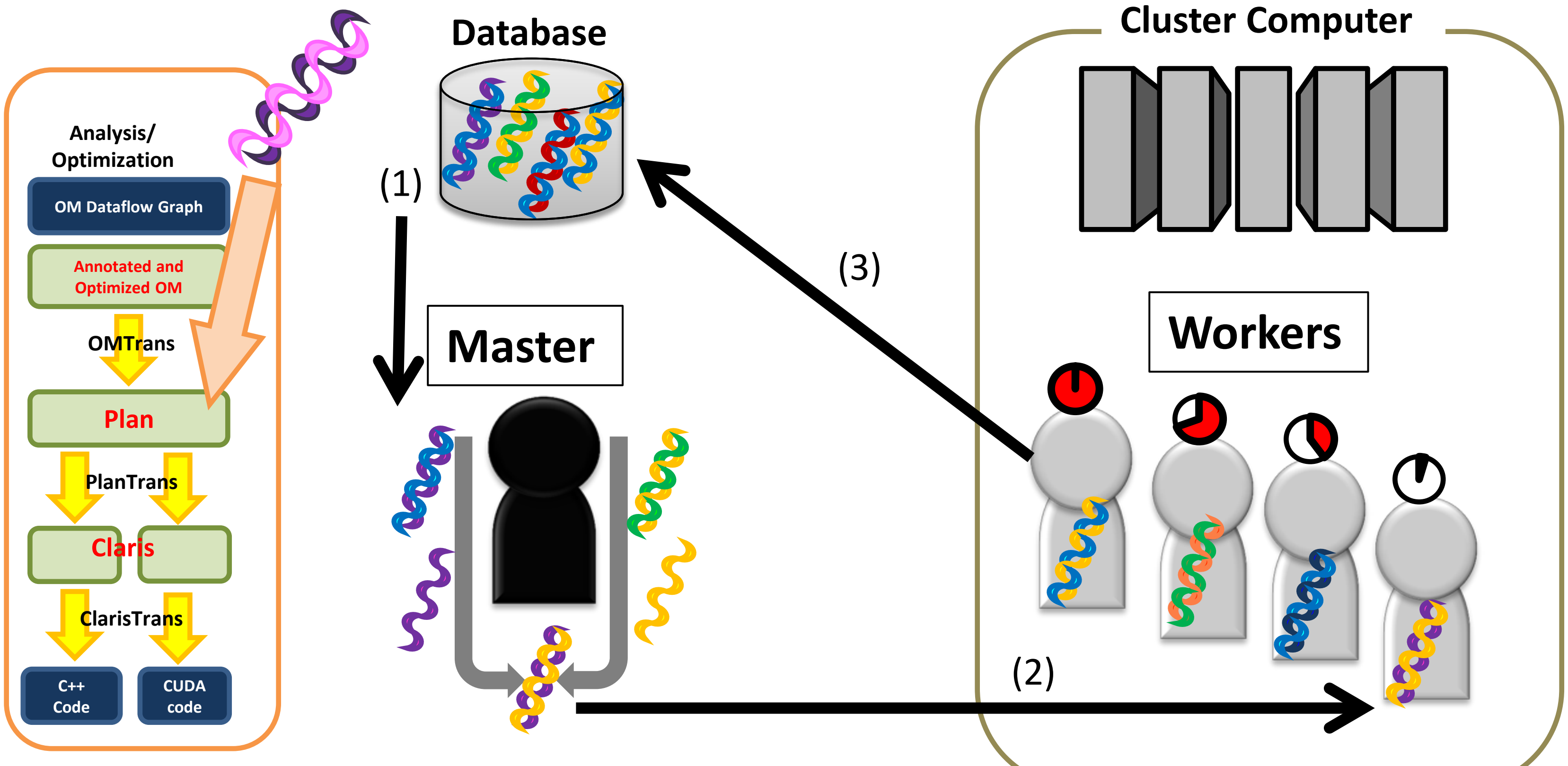
I also add annotations here...

```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
```

Strategy Annotation	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

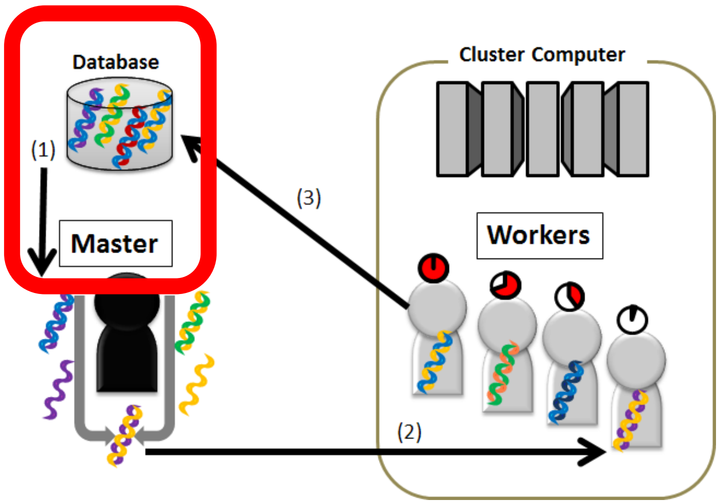
Automated Tuning

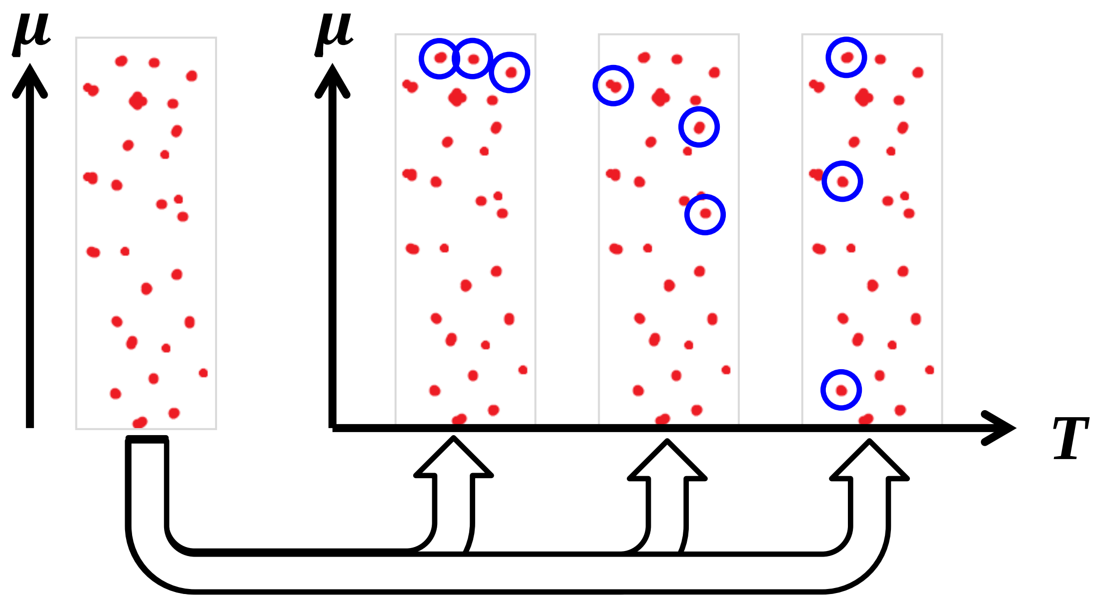
(Genetic Temperature-parallel Annealing)



0. The genome and benchmark results are stored in the database
1. The Master performs “finite temperature draws” from the database and creates a new genome, launches a worker
2. Worker generate and benchmark the code w.r.t the genome
3. The result is written to the database

Finite temperature draw(n, T)





- The probability for drawing an individual I with score $\mu(I)$ is proportional to $\exp\left(\frac{\mu(I_{\text{top}})-\mu(I)}{T}\right)$
- T is randomly chosen per draw

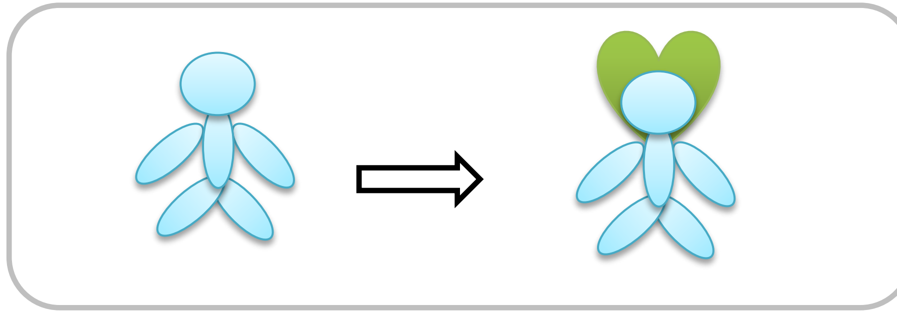
Three kind of children creation

mutation (1Parent)

ATATATAAATTATATATATAAAAAAAAAAAT

↓

ATATA**GC**AATTATAT**C**TATAAAAA**GTG**AAAAT



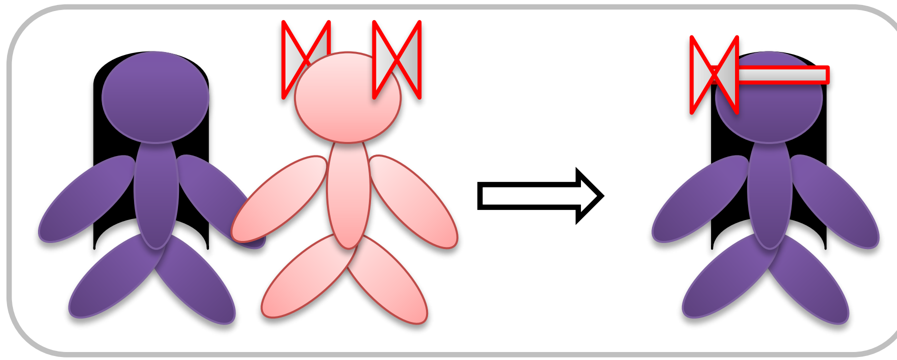
crossover (2Parents)

ATATATAAATTATATATATAAAAAAAAAAAT

GGCCGCGCCCGCGCGCCCGCGCGCCCGGCGG

↓

ATAT**GCG**AATTATATATA**CGCGCGCCCGGCG**T



triangulation (3Parents)

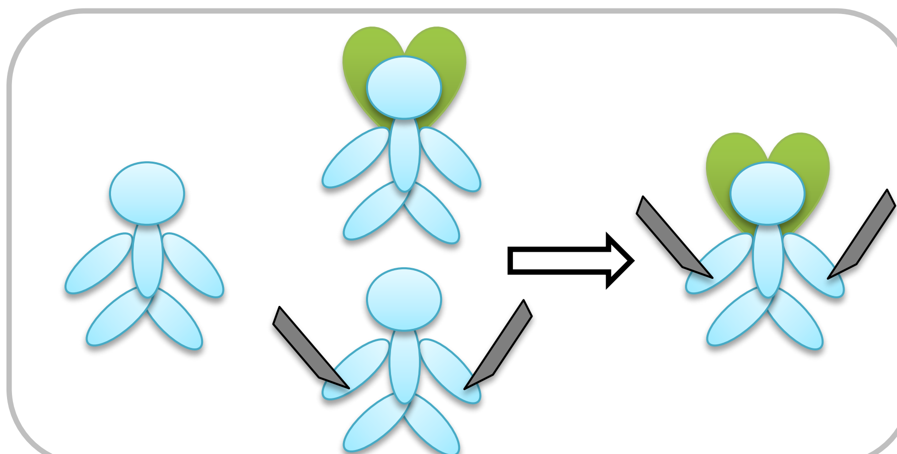
ATATATAAATTATATCTATAAAAAAAAAAAT

ATATATAAATTATAT**C**TATAAAAAA**GTT**AAAT

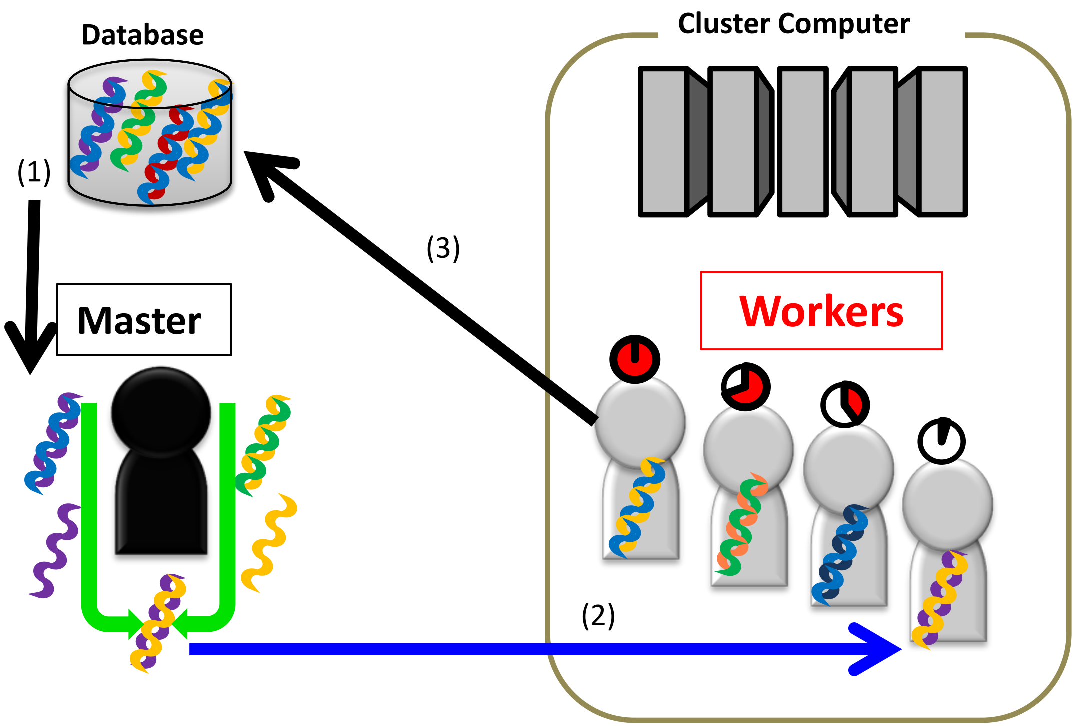
ATATA**GC**AATTATAT**C**TATAAAAAAAAAAAT

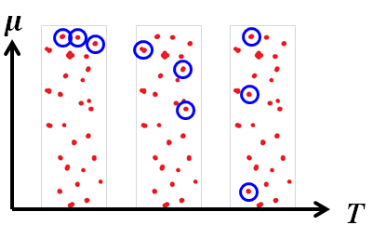
↓

ATATA**GC**AATTATAT**C**TATAAAAAA**GTT**AAAT



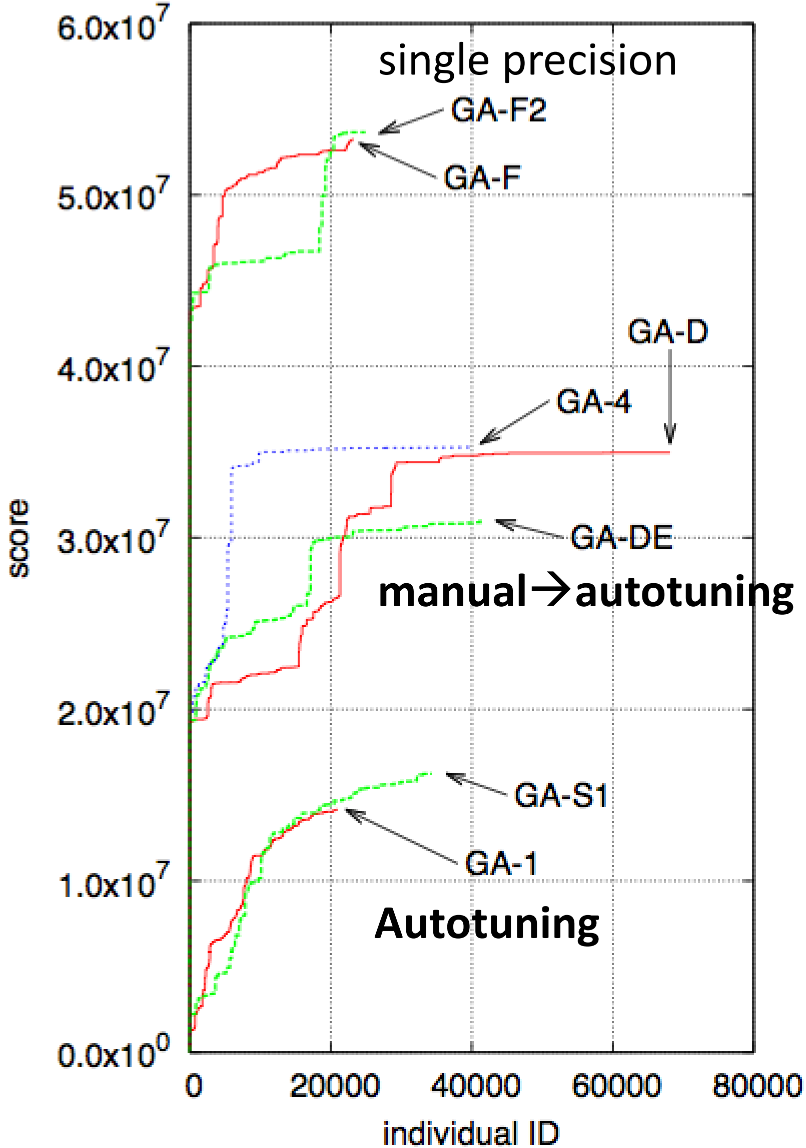
probabilistic & parallel temperature annealing + genetic algorithm

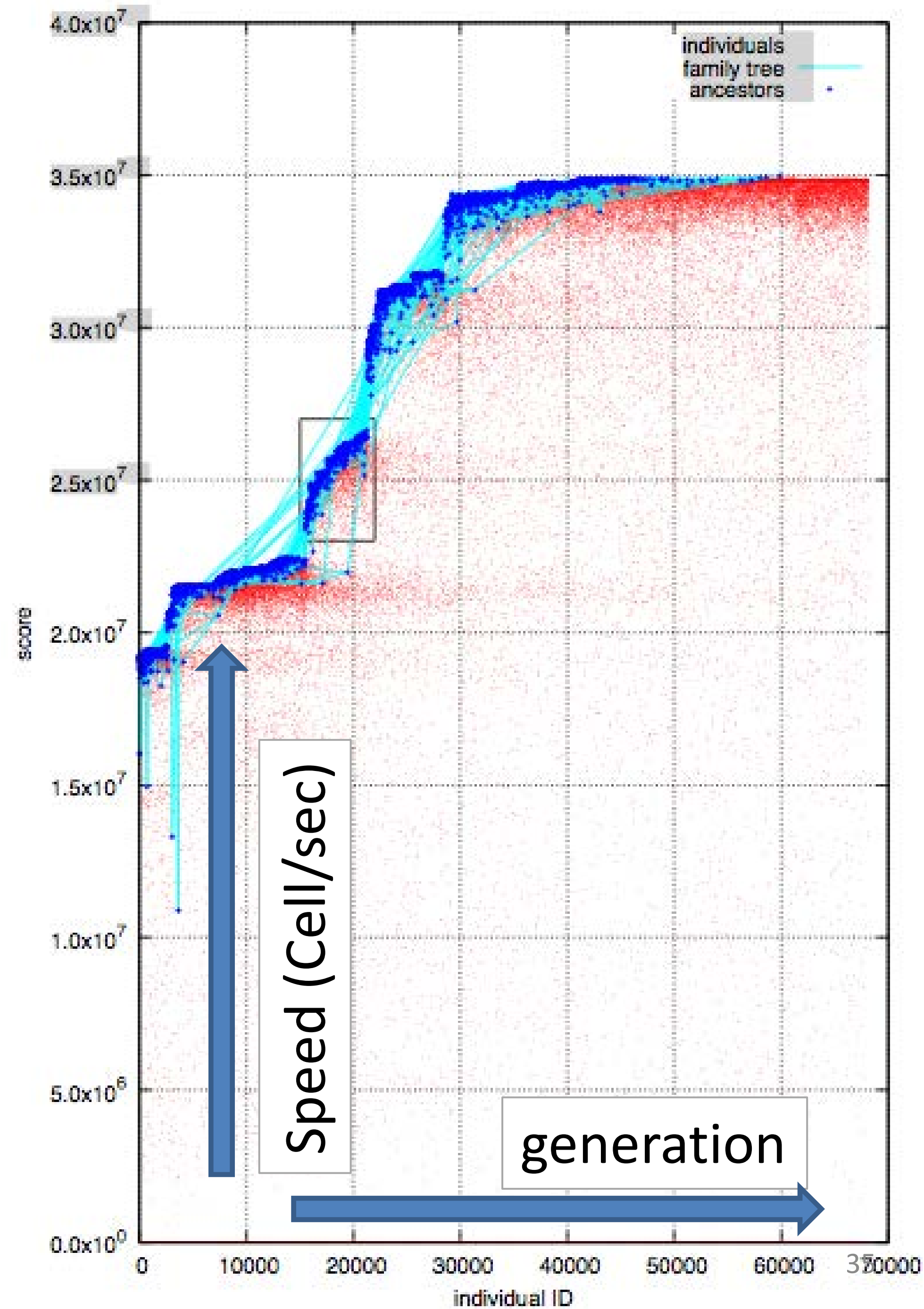




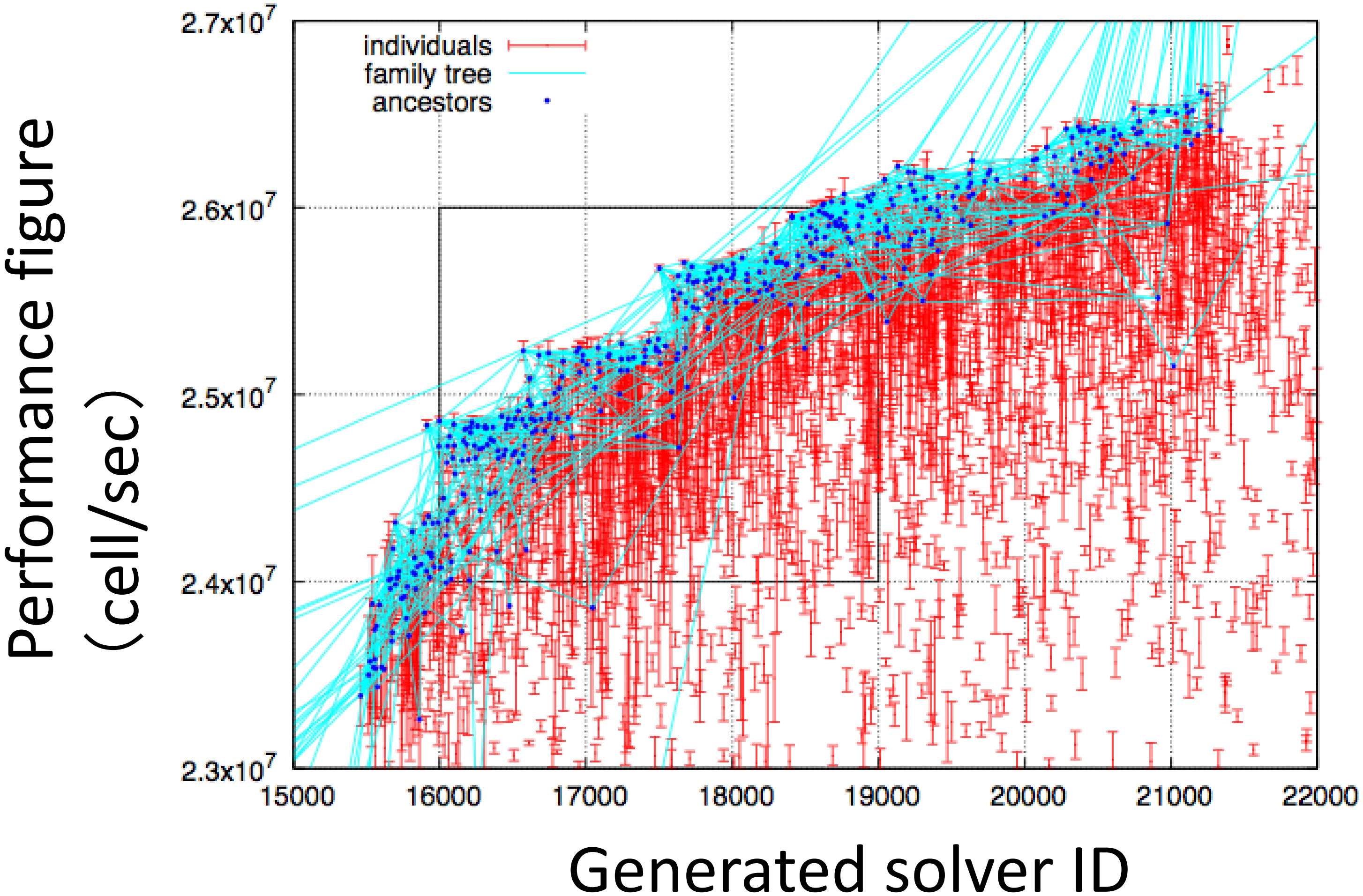
parallel temperature	No annealing schedule management
probabilistic temperature	Utilize dynamic computer resource
Genetic algorithm	merge independent updates

Track of history

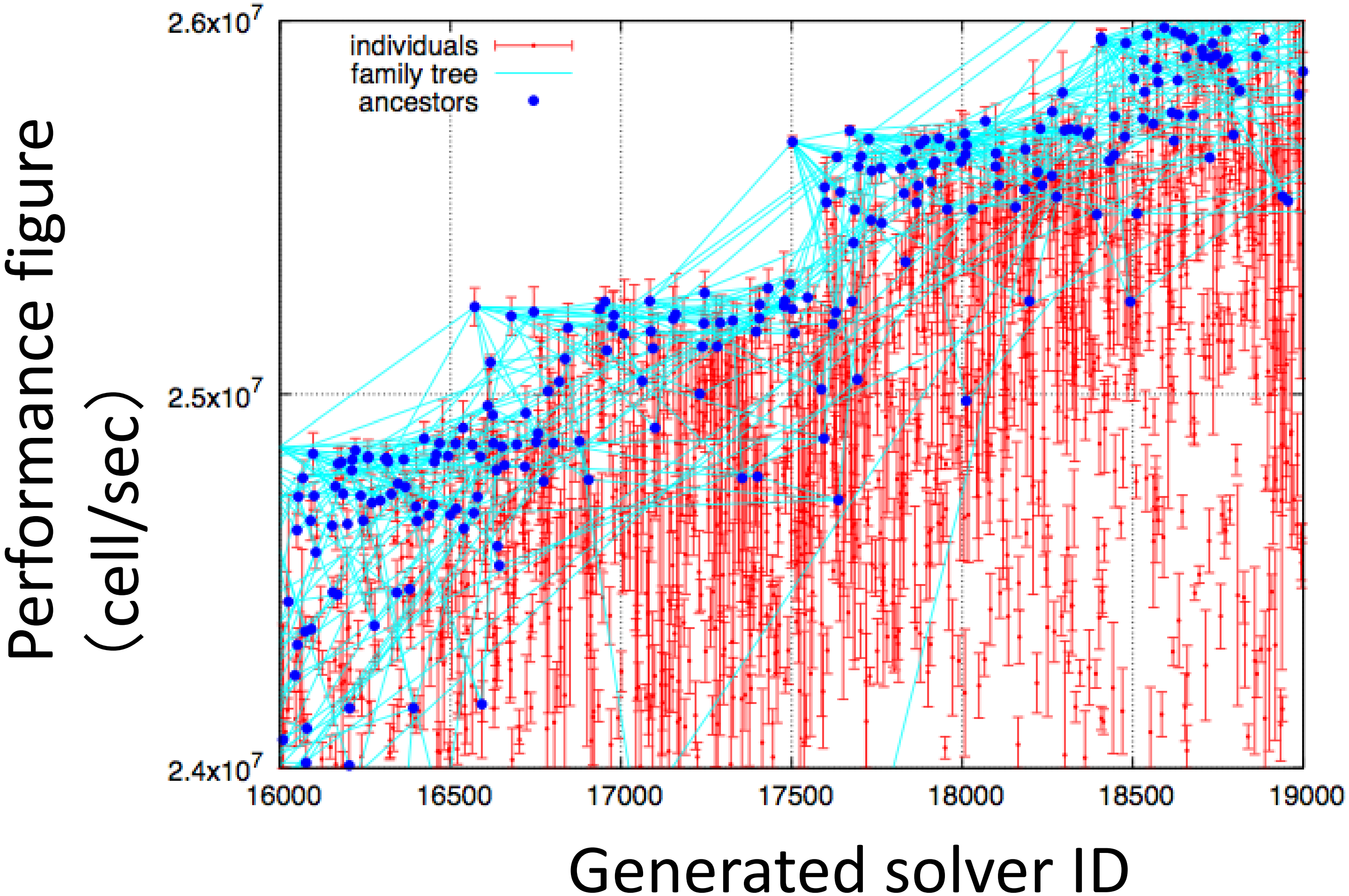


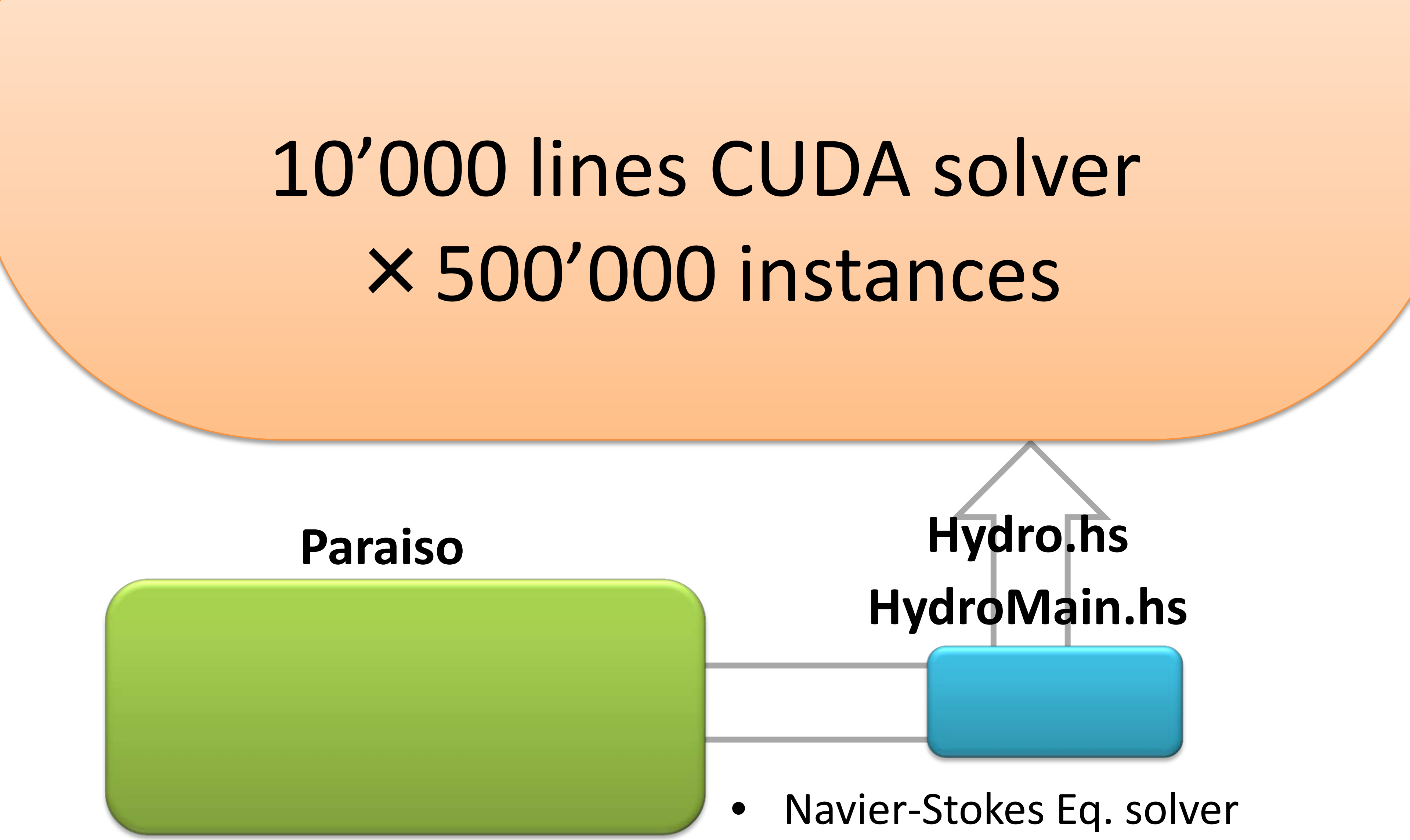


zoom-in (1)


Performance figure
(cell/sec)
2.7x10^7
2.6x10^7
2.5x10^7
2.4x10^7
2.3x10^7
individuals
family tree
ancestors
15000
16000
17000
18000
19000
20000
21000
22000
Generated solver ID

zoom-in (2)

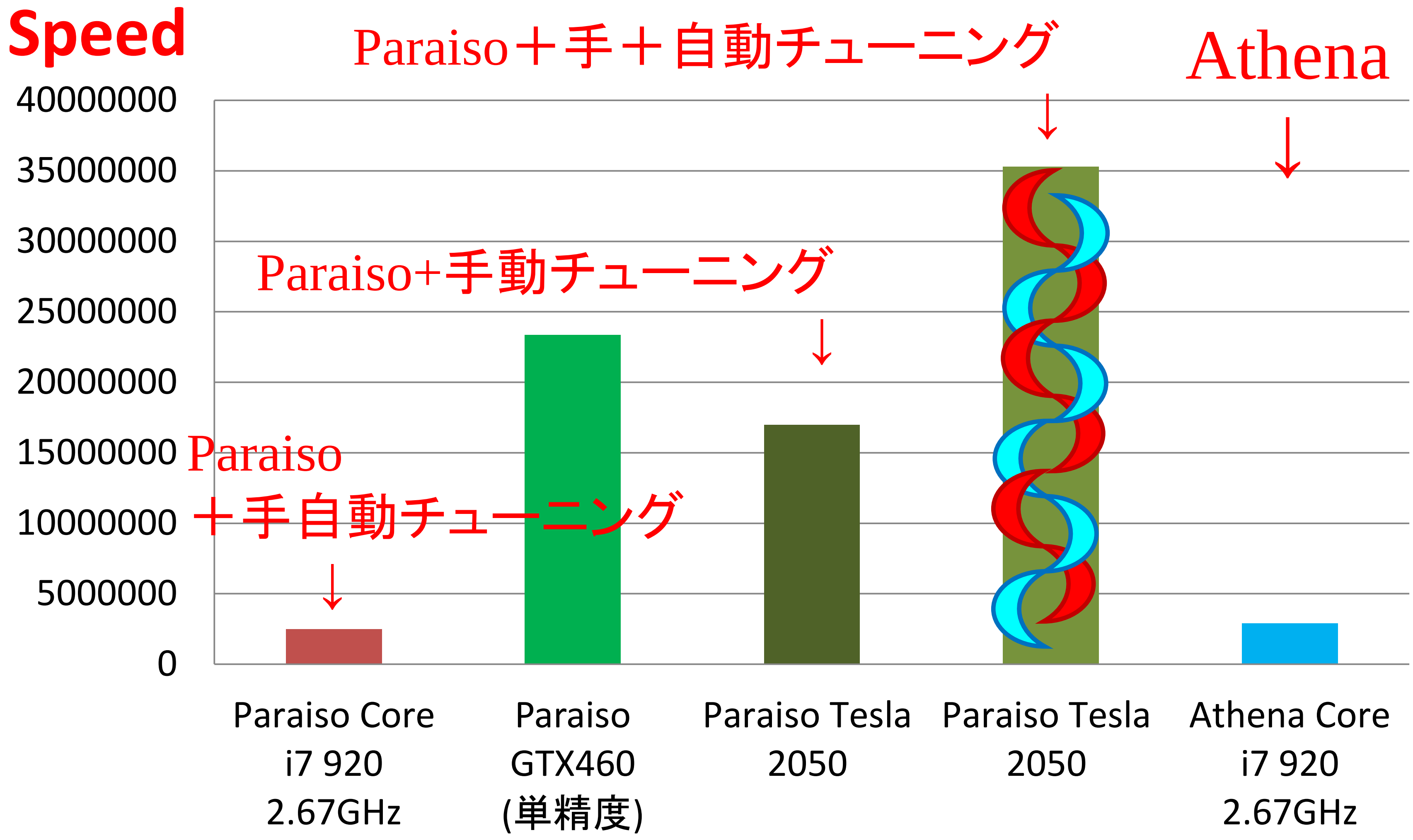

individuals
family tree
ancestors
2.6x10^7
2.5x10^7
2.4x10^7
16000
16500
17000
17500
18000
18500
19000
Performance figure
(cell/sec)
Generated solver ID



- 5000 lines of haskell code

- Navier-Stokes Eq. solver
- 464 lines

Benchmark

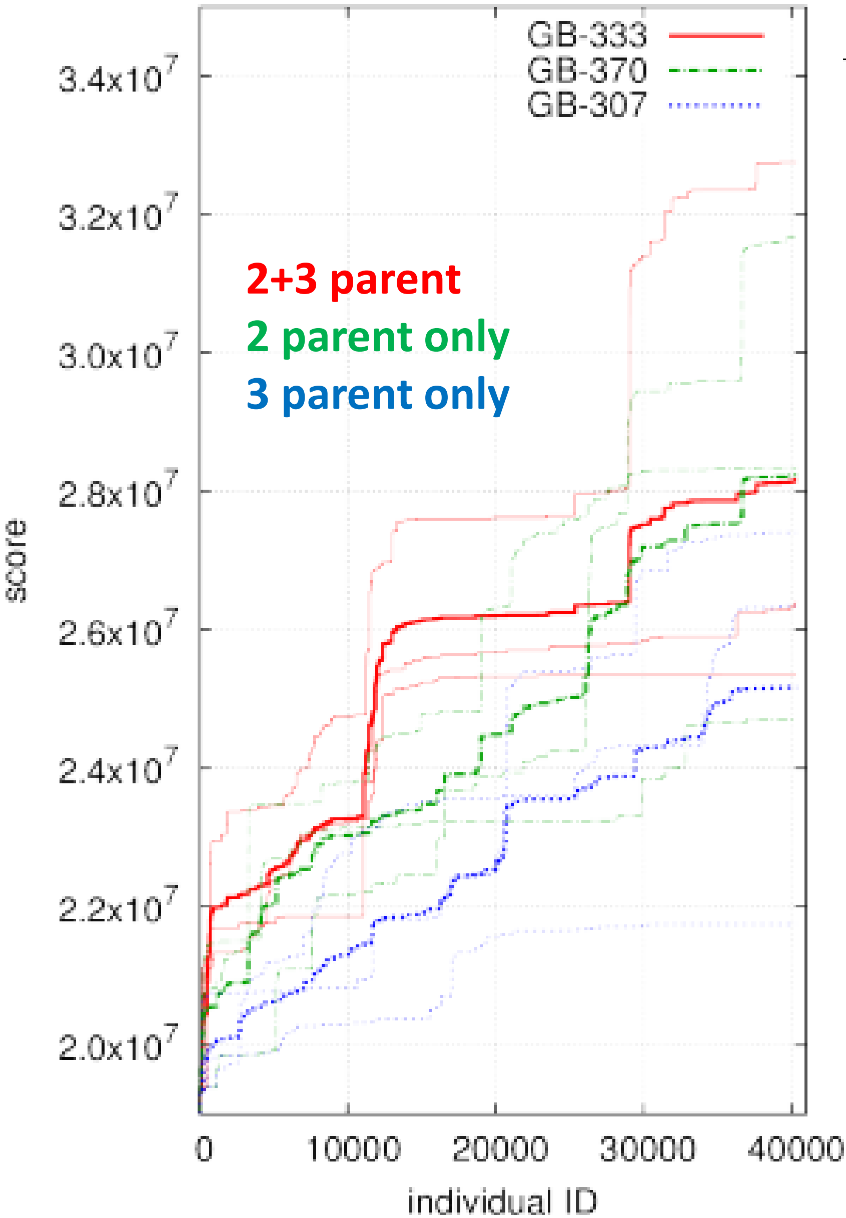

Speed
Paraiso＋手＋自動チューニング
Athena
Paraiso+手動チューニング
Paraiso
＋手自動チューニング
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Paraiso Core i7 920 2.67GHz
Paraiso GTX460 (単精度)
Paraiso Tesla 2050
Paraiso Tesla 2050
Athena Core i7 920 2.67GHz

Statistical Anaylsis of the evolution

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

Table 11. 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.

- Markov chain analysis of the family tree
- Family tree being 0th and 1st Markov process rejected
- 2parent → 2parent is significantly not likely
- 3parent → 3parent is significantly likely



mutation 33420(1.000)			crossover 15412(1.000)				triangulation 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 A11. Children relative fitness classification for Experiment GA-D.

- 2parent crossover is good at making larger jumps, while 3parent crossover is good at accumulating small updates.
- Having both 2parent and 3parent crossover is better than having just either one of 2 or 3-parent.

Automated tuning challenge (?)

Haskell can

- generate random instances
- {-# DeriveTraversable #-}
- optimize anything that is **`Traversable`**

```
class Arbitrary a where
```

Random generation and shrinking of values.

Methods

```
arbitrary :: Gen a
```

A generator for values of the given type.

The cmaes package hackage.haskell.org/package/cmaes

```
minimize :: ([Double] -> Double) -> [Double] -> Config [Double]
minimizeT :: Traversable t => (t Double -> Double) -> t Double -> Config (t Double)
minimizeG :: Data a => (a -> Double) -> a -> Config a
```

ary algo

Can Haskell (or your favorite language) provide automated tuning over arbitrary types by

- define typeclasses for 2- or 3-parent crossover?
- derive instances for such **`Crossover`** classes?

references

- **http://arxiv.org/abs/1204.4779**
- automated tuning script

https://github.com/nushio3/Paraiso/blob/master/examples-old/GA/make_task.rb

- “gene bank” of initial species

https://github.com/nushio3/Paraiso/tree/master/examples-old/GA/genomeBank

- OM dataflow graph description for Hydro

https://raw.githubusercontent.com/nushio3/Paraiso/master/examples-old/Hydro-exampled/output/OM.txt

- Generated CUDA program for Hydro

https://github.com/nushio3/Paraiso/tree/master/examples-old/Hydro-exampled/dist-cuda