# Orthotope Machine

Takayuki Muranushi

April 13, 2011

In geometry, an *orthotope* (also called a hyperrectangle or a box) is the generalization of a rectangle for higher dimensions, formally defined as the Cartesian product of intervals.

"Orthotope" means "multidimensional array" in this document.

## 1 Introduction

This document describes the *Orthotope Machine*, a virtual machine that operates on multidimensional arrays. The Orthotope Machine is one of the main components for Paraiso project. The goal of Paraiso project is to create a high-level programming language for generating massively parallel, explicit solver algorithms of partial differential equations.

From astrophysical interest, the Paraiso project will make simulations much easier for basic equations such as hydrodynamics, magnetohydrodynamics, general relativity, relativistic radiative transfer and so on. Buiding complex model from combinations of these basic equations, chemistry, nuclear reactions, will also become much easier and tractable. The generated program will support various machines from 1-node GPU workstation to the K computer in Kobe. One feature that is desired but missing in this document is support for fixed mesh refinement (FMR). And of course, there are lots of problems in astrophysics, that do not reduce to solving partial differential equations explicitly.

From computational viewpoint, explicit solvers of partial differential equations belongs to the algorithm category called stencil codes. Stencil codes are algorithms that updates the array, each element accessing the nearby elements in the same pattern (c.f. Fig.1). Stencil codes are commonly used algorithms in fields such as solving partial differential equations and image processing. Code generations and automated tuning for stencil codes has been studied e.g.

```
double a[NY][NX], b[NY][NX];
for (int t=0; t<max_t; ++t) {
  for (int y=1; y<NY-1; ++y)
    for (int x=1; x<NX-1; ++x)
      b[y][x] = a[y][x-1] + a[y][x+1]
              + a[y-1][x] + a[y+1][x];

  for (int y=1; y<NY-1; ++y)
    for (int x=1; x<NX-1; ++x)
      a[y][x] += 0.25 * b[y][x];
}
```

Figure 1: An example of stencil code.

[Dat09, DMV+08].

There are many methods other than stencil codes for solving partial differential equations. They have different merits. A notable project in progress is Liszt [CDM+10], an embedded DSL(domain specific language) in programming language Scala, designed for generating hydrodynamics solver on unstructured mesh.

Many parallel and distributed programming languages has been implemented using Haskell [TLP02]. Data Parallel Haskell [PJ08] and Nepal[CKLP01] are implementations of NESL, a language for operating nested arrays. Accelerate [CKL+11] and Nikola [MM10] are languages to manipulate arrays on GPUs written in Haskell.

We need new languages for parallel hardwares — this is a long-standing idea. Many project sought for them, and some failed. Failures from which we can learn. High Performance Fortran was a very promising approach to introduce a high-level parallelism in Fortran but, as James Stone told me in Taiwan, and as is summarized by the project leader [KKZ07], it failed. DEQSOL [NCY15, KSSU86] was another project which had design similar to that of Paraiso. The language was initially designed for Hitachi vec-

tor machines. The extension of DEQSOL for parallel vector machines has been planned [NY15] but seemingly did not realize.

The unique point of Paraiso compared to those projects is its focus on computational domains that utilize localized access to multidimensional arrays. Paraiso is also not a caller of libraries for known algorithms; rather, it is a tool for testing and implementing new algorithms.

Multidimensional arrays are different from nested arrays. For example in the pseudocode Fig.1, in order to calculate `b[y][x]` you need to read from `a[y-1][x]` and `a[y+1][x]`, which are usually located much farther in the memory compared to `a[y][x-32]` or `a[y][x+64]`. The code generator must be aware of such locality in multidimensional space. For most of the cases, the basic equations to be solved is symmetric under exchange of the axes (X,Y,Z ...). Still, there are non-negligible differences between the axes from computational point of view, especially if the multidimensional arrays are stored in row-major or column-major order. To utilize the cache and/or vector instructions the code generator need to know and decide upon the order the array is stored in the memory.

In parallel machines, the array must be decomposed and distributed among computer nodes. It is important to take care of the continuity in multidimensional space when making the distribution, so that the communications cost is lowered. If the data to be communicated is stored sparsely in the memory, it is a good strategy to gather them manually into a single buffer and to make a single large communication instead of making lots of small communications. Due to this gather/scattering cost, not making any decomposition in one or more directions gives higher performance in some cases.

The Orthotope Machine is designed to capture and utilize these characteristics of the multidimensional array computations.

This document is organized as follows. In §2, I describe the computational natures of the algorithms used to solve the partial differential equations explicitly. In §3, I describe the overall design of Paraiso, to clarify the Orthotope Machine's role in it. In §4, I give definitions for Orthotope and Orthotree. §5 gives an idea for modeling the parallel computer hardware. §6 presents a c++ API for Paraiso. §7 gives the instruction set for Orthotope Machines. Finally, §8 deals with possible optimization techniques for Orthotope Machines.

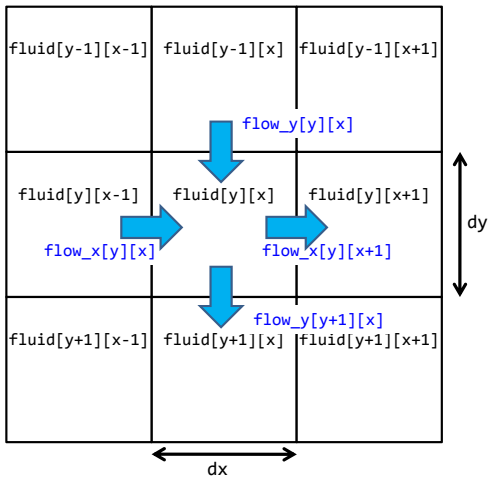


Figure 2: Schematic image of how a fluid simulator works.

In short, I'd like to make something called Paraiso, that takes DPDEL (Fig. 5) as the input, then translates it to Orthotope Machine instruction (Fig.6), then to native program like (Fig.3), and generates a solver library with API described in §6.

# 2 Explicit Solvers of Partial Differential Equations

To begin with, let me describe what kind of problem I want to solve.

Let us take for example a hydrodynamics solver that represent the fluid by a mesh structure c.f. Fig. 2. Each mesh has volume $dx\,dy$. Each mesh stores the amount of fluid wihin that volume. The solver's algorithm is to calculate the flows of the fluid across the mesh boundaries, and add/subtract the amount of fluid.

Fig. 3 shows a pseudocode for a three-dimensional hydrodynamic equations solver. Although it is a bit simplified to be a real solvers, it shows what components needed to build one.

A common task for the solver is to simulate the evolution of the fluid for a certain interval of time, say, $0 < t < t_{max}$ (1). In the each step of the simulation, the time `t` increases by a certain amount `dt`. In many problems, the time-step `dt` is not a constant but depends on the state of the fluid. In such case, you need to calculate the time-steps adequate for the

```
double fluid[NZ][NY][NX];
double flow_x[NZ][NY][NX];
double flow_y[NZ][NY][NX];
double flow_z[NZ][NY][NX];
double dt_local[NZ][NY][NX];

// (1) simulation goes from time t=0 to t=t_max
for (double t=0; t<t_max; t+=dt) {

  // (2) calculate the timescale for each mesh
  for (int z=1; y<NZ-1; ++z)
    for (int y=1; y<NY-1; ++y)
      for (int x=1; x<NX-1; ++x)
        dt_local[z][y][x]=timescale(fluid[z][y][x]);

  // (3) calculate the minimum timescale
  double dt=max_t;
  for (int z=1; y<NZ-1; ++z)
    for (int y=1; y<NY-1; ++y)
      for (int x=1; x<NX-1; ++x)
        dt=min(dt, dt_local[z][y][x]);

  // (4) calculate the flow for each direction
  for (int z=1; y<NZ; ++z) {
    for (int y=1; y<NY; ++y) {
      for (int x=1; x<NX; ++x) {
        flow_x[z][y][x]=calc_fx(fluid[z][y][x-1], fluid[z][y][x]);
        flow_y[z][y][x]=calc_fy(fluid[z][y-1][x], fluid[z][y][x]);
        flow_z[z][y][x]=calc_fz(fluid[z-1][y][x], fluid[z][y][x]);
      }
    }
  }

  // (5) move the fluid according to the flow
  for (int z=1; y<NZ-1; ++z)
    for (int y=1; y<NY-1; ++y)
      for (int x=1; x<NX-1; ++x)
          fluid[z][y][x] += dt*(
            (flow_x[z][y][x]-flow_x[z][y][x+1])/dx +
            (flow_y[z][y][x]-flow_x[z][y+1][x])/dy +
            (flow_z[z][y][x]-flow_x[z+1][y][x])/dz);

}
```

Figure 3: A pseudocode showing the typical design of a hydrodynamic equations solving algorithm. c.f. Fig. 2

fluid state of every mesh (2), and then perform reduction over the entire computational domain to calculate the smallest `dt` (3). Then, amount of fluid flow across X, Y, and Z mesh boundaries are calculated from the fluid state of the meshes that are adjacent to the boundaries (4).

To generate a real-world fluid solver, there are more complexity compared to the example in Fig. 3. First, the `fluid` has not one but several component. For example, we typically need five 3D arrays (or an array of a struct with five elements) to solve hydrodynamics. They are density, three components of the velocity, and energy. For magneto-hydrodynamics, we need eight components. For general relativity we need twenty-one.

Second, the size of the stencil (the number of input mesh needed to calculate the answer for one mesh) is larger. `calc_fx` in Fig. 3 reads two mesh. In real solvers, they may read four, six or more meshes, to achieve higher-order interpolation in space. Third, the entire solver will be replicated several times, to achieve higher-order time integral. Runge-Kutta method is a well known example of such a technique. This also make the stencil larger.

Too large stencil is a bad thing. With larger stencil we need to perform more duplicated calculations, and we need communication buffer so large that distributed computation do not contribute to better performance. So we need to split the algorithm into several components with smaller stencil. Splitting it into too many pieces, is again not a good idea. We must search for the middle path.

To get a feeling of a real "real-world solver," please look at `proceed` function of my magneto-hydrodynamics solver I'm working right now : `https://github.com/nushio3/nmhd/blob/master/src/library/mhd_solver.inl` . I just wanted you to see how dirty the code is and how it repeats itself. The code is actually generated from a more compact, but less readable ruby script : `https://github.com/nushio3/nmhd/blob/master/src/library/mhd_solver.inlrb` .

# 3 Overall Design of Paraiso

In this section I briefly describe the overall design of Paraiso, to clarify the Orthotope Machine's role in it. Please also c.f. `http://paraiso-lang.org/wiki/index.php/Grand_Design`.

Paraiso translates the input programs into native programs with several steps: c.f. Table 1. The input language is DPDEL (Discretized Partial Differential Equation Language); the language to describe the numerical simulation algorithms, in as simple form as possible c.f. `http://paraiso-lang.org/wiki/index.php/DPDEL`. Then it is translated to Orthotope Machine (OM) program. Its name used to be Virtual Vector Machine (VVM) in original Paraiso proposal c.f. `http://paraiso-lang.org/wiki/index.php/VVM`. There are two versions of Orthotope Machine: one is Primordial OM, the other is Distributed OM. DPDEL program is first translated to Primordial OM program, on which it can use arbitrary large arrays. Then "machine division" operations are applied, according to the hardware specifications, producing Distributed OM instructions. Distributed OM instructions are then translated to native codes. Native codes are compiled by existing compilers, and finally, all programs are executable.

## 3.1 Discretized Partial Differential Equation Language (DPDEL)

DPDEL program is constructed within a monad: This technique is described in tanakh's blog `http://d.hatena.ne.jp/tanakh/20070918` and is featured in the first prototype of Paraiso. The monad is called `Dsl` and defined in `https://github.com/nushio3/Paraiso/blob/master/attic/paraiso-2008-ODEsolver/Paraiso.hs`.

Fig.4 shows the possible DPDEL program. Although it is fairly readable, it has some shabby points.

- Calculations for X,Y,Z directions are repeated,
- Vectors are represented in scary ways: `((-1):.0:.0)`,
- Most part of the program is to assign an expression `expr` to a new AST node `a`, but you cannot write `a <- expr` since `a` is also an expression and `a` and `expr` have the same type; `expr` cannot be a monad that returns `expr`. Expression must go through some function e.g. `bind :: AST a -> DPDEL (AST a)`.
- You are forced to use customized operators `+. *.` ... instead of usual operators `+ *`... . This may not happen in the arithmetic operators, but at least it surely happens for comparison operators such as `>`, because their type is `a -> a -> Bool`. You cannot get it have the type `AST`

| Language | Sample code | Handled by |
|---|---|---|
| DPDEL | Fig.4,5 | DPDEL monad, quasiquoter and parser |
| Primordial Orthotope Machine AST | Fig.6 | Orthotope Machine divider |
| Distributed Orthotope Machine AST | | Orthotope AST optimizer and compiler |
| Native codes (Fortran,C,CUDA,OpenCL with MPI) | Fig.3 | Native compilers |
| Executables | | Real machines |

Table 1: The software stack of Paraiso, through which DPDEL programs are translated to executables.

`a -> AST a -> AST Bool`. This barrier is also observed in Nikola, a DSL for GPU in Haskell [MM10]. Well, you can, in Haskell, but in nasty ways.

Fig.5 shows another possible version of DPDEL with above drawbacks removed. First improvement is the concept of Axis. In the sample code, axis has two role. One is to specify a certain component of three-dimensional vector, e.g. `flow_axis`. The other is to shift the indices for three-dimensional arrays. The indices are represented as lambda-bound variable `o` and shifted like `[o+axis]`.

This double-roled Axis is invented for my magnetohydrodynamics program. The two role combined, it really helps writing template functions, instantiated three times, each of which handles one of X,Y,Z directions. The concept of Axis is defined in `https://github.com/nushio3/nmhd/blob/master/src/library/direction.h` as a set of dummy structs `XAxis, YAxis, Zaxis` etc. It is extensively used to construct three-dimensionally accessible arrays `https://github.com/nushio3/nmhd/blob/master/src/library/crystal.h` as well as to write the main program.

But how does Haskell know that the name `flow_axis` is related with the name `axis`? Isn't `flow_axis` a single name identifier in Haskell syntax? Here comes the second point.

Let me introduce `qb`, quasiquote-and-bind operation. This lifts the constraints coming from the Haskell syntax. It parses its argument string and does several things: it changes `flow_axis` into something like `getComponent axis flow`, and translates `[o+axis]` into type-correct expression like `[o+getUnitVector axis]`, thus enables the use of double-roled Axis with succinct notation. It renames other arithmetic operators appropriately. It has implicit antiquotes. Finally, `qb` adds `bind` to the entire expression. `qb` grants us magical power.

Well, although what `qb` offers us seem so alluring, things may not go too well with `qb`. We may find some of the facilities described above impossible to implement. We may need several quasiquoters instead of just one. Even so, use of quasiquote is essential in making DPDEL syntax simpler.

## 3.2 Orthotope Machine

DPDEL programs are translated into programs for the Orthotope Machine, the main topic of this document. The OM program describes data-flow from one orthotope to others, in static single assignment (SSA) style machine-language-flavor program, forming a directed acyclic graph. The DPDEL program in Fig. 5 will translate to something like Fig. 6 . Also look at `https://github.com/nushio3/Paraiso/blob/master/attic/paraiso-2010-lifegame/` for the working example, the second prototype of Paraiso. When you `make` this program, you'll see the Conway's game of life running. `Main.hs` contains the program for the cellular automata. `VVM.hs` defines the instruction set and the syntax tree, and `VVMCompiler.hs` is a compiler.

```
timescale = ...
calcFX f1 f2 = ...
calcFY f1 f2 = ...
calcFZ f1 f2 = ...

proceed :: DPDEL ...
proceed = do
  t <- input $ Orthotope0
  fluid <- input $ Orthotope3
  dtLocal <- bind $ timescale fluid
  dt <- bind $ reduce Min dtLocal

  flowX <- bind $ calcFX fluid (shift ((-1):.0:.0) fluid)
  flowY <- bind $ calcFY fluid (shift (0:.(-1):.0) fluid)
  flowZ <- bind $ calcFZ fluid (shift (0:.0:.(-1)) fluid)

  newFluid <- bind $ fluid +. dt *. (
    (flowX -. shift (1:.0:.0) flowX)/.dx +
    (flowY -. shift (0:.1:.0) flowY)/.dy +
    (flowZ -. shift (0:.0:.1) flowZ)/.dz )

  output (t+dt) t
  output newFluid fluid
```

Figure 4: A DPDEL monad that corresponds to the pseudocode in Fig. 3.

```
timescale = ...
calcF axis f1 f2 = ...

proceed :: DPDEL ...
proceed = do
  t <- input $ Orthotope0
  fluid <- input $ Orthotope3
  dtLocal <- bind $ timescale fluid
  dt <- bind $ reduce Min dtLocal

  flow <- forAxes (\axis ->
    [qb| o -> calcF axis fluid[o] fluid[o-axis]] )


  updates <- forAxes (\axis -> [qb| o-> (flow_axis[o] - flow_axis[o+axis]) / dr_axis ]

  newFluid = [qb| o-> fluid[o] + dt * (sum updates)[o]]

  output (t+dt) t
  output newFluid fluid
```

Figure 5: A more sophisticated DPDEL code using quasi-quote and Axis. **qb** is a quoted bind. The argument of **qb** is parsed by a separate parser that understands e.g. the Einstein rules, and a **bind** is added. Compare with Fig. 4.

```
t <- input
fluid <- input
dtLocal <- ...
dt <- reduce dtLocal

a1 <- shift ((-1):.0:.0) fluid
flowX   <- arith calcFX fluid a1
a2 <- shift (0:.(-1):.0) fluid
flowY   <- arith calcFY fluid a2
a3 <- shift (0:.0:.(-1)) fluid
flowZ   <- arith calcFZ fluid a3

a4 <- shift (1:.0:.0) flowX
a5 <- arith (-) flowX a4
a6 <- arith (/) a5 dx
a7 <- shift (0:.1:.0) flowY
a8 <- arith (-) flowY a7
a9 <- arith (/) a8 dy
a10 <- shift (0:.0:.1) flowZ
a11 <- arith (-) flowZ a10
a12 <- arith (/) a11 dz
a13 <- arith (+) a6 a9
a14 <- arith (+) a13 a12
a15 <- arith (*) a14 dt
a16 <- arith (+) a15 fluid

newT <- arith (+) t dt

output newT t
output a16 fluid
```

Figure 6: A sketch of how an Orthotope Machine instruction will look like.

# 4 Orthotope and Orthotree

o

## 4.1 Orthotope

Orthotope is a rectangle in $n$-dimension. A zero-dimensional orthotope is a point; an one-dimensional orthotope is an interval; a two-dimensional orthotope is a rectangle and a three dimensional orthotope is a box. The intersection of two orthotopes is also an orthotope, but union of two orthotopes is not. The set of all $n$-dimensional orthotopes is closed under intersections, but not under unions. Such system is called a $\pi$-system in mathematics.

```
class PiSystem a where
  empty :: a
  null :: a -> Bool
  intersection :: a -> a -> a
```

We do not introduce union between two orthotopes. The reason why we don't need union will be explained in the next subsection.

An interval is either empty, or a pair of two ordered elements. Interval is an instance of $\pi$-system.

```
data Interval a = Empty |
     Interval {lower::a, upper::a}
instance (Ord a) => PiSystem (Interval a)
  where
```

A zero-dimensional orthotope can have two unit state: it can either be vacant (`Z0`) or occupied (`Z`). Orthotopes of higher dimensions are built using the *snoc* operator `(:.)` borrowed from Repa [KCL+10].

```
data Orthotope0 a = Z0 | Z

type Orthotope1 a=Orthotope0 a:.Interval a
type Orthotope2 a=Orthotope1 a:.Interval a
type Orthotope3 a=Orthotope2 a:.Interval a
```

For more detail, please refer to the source `https://github.com/nushio3/Paraiso/blob/master/Language/Paraiso/Orthotope.hs` and example `https://github.com/nushio3/Paraiso/blob/master/attic/TestOrthotope.hs` .

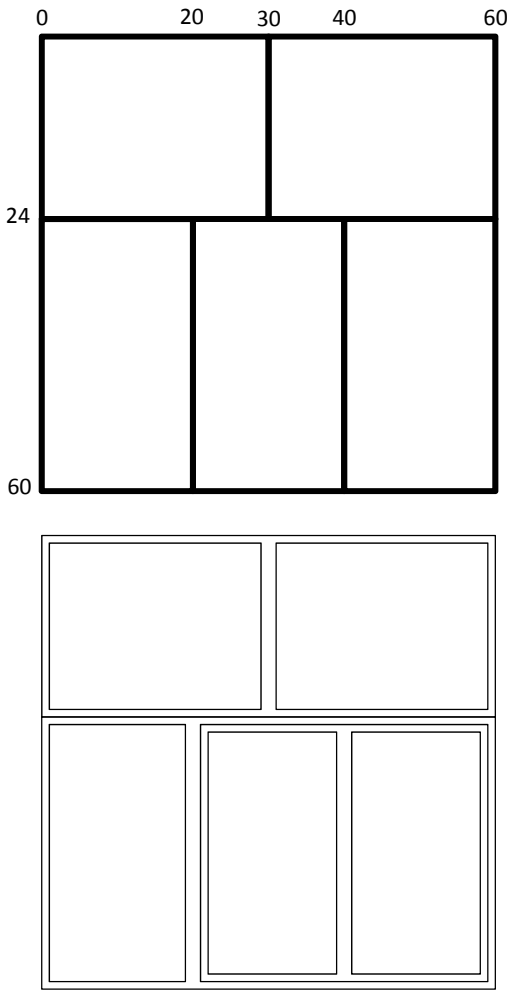


Figure 7: An Orthotree with five leaves.

### 4.2 Orthotree

So far, we assumed a single ideal machine with infinite amount of memory. Real machines have finite memories and consist of distributed components connected with limited bandwidth. So we have to fit the ideal into the reality.

We first start from an infinitely large orthotope, and cut it recursively until each computer node has a portion of the orthotope. Introducing boundary conditions and domain splitting are treated uniformly. Either of the operations introduces restrictions on the parent orthotope, and generates needs for additional communications so that the divided calculations is identical to the calculations before division.

The recursive division of an Orthotope is denoted as an Orthotree :

```
data Orthotree a =
  Leaf |
  Warp Axis (Interval a) Orthotree |
  Divide Axis a Orthotree Orthotree
```

Here, `Warp` introduces a cyclic boundary condition, and `Divide` denotes an orthotope division. For example,

```
t :: Orthotree Int
t = Warp Y (Interval 0 60) $
    Warp X (Interval 0 60) $
    Divide Y 24
      (Divide X 30 Leaf Leaf)
      (Divide X 20 Leaf
        (Divide X 40 Leaf Leaf))
```

denotes the structure in Fig. 7 Orthotree is a variation of data structures to represent domain decompositions, such as oct-trees.

The reason why we do not introduce union between orthotopes is that we can always back-trace this Orthotree. Unions are allowed only between orthotopes that are apart but used to be united. Those are only kinds of unions that we need, and the union results are always guaranteed to be orthotopes.

### 4.3 Load Balancing

Generally speaking, boundary indices are not compile-time constants. This is because each MPI instance of the program do not know which portion of the Orthotree to solve until execution. Moreover, we can design Distributed OM so that boundary indices are changeable at runtime. It opens path to dynamic load balancing. If we achieve that, and also achieve the overlapped communication and calculation, any program that is generated from Paraiso and communication time is less than computation time will achieve near 100% machine utilization on any heterogeneous architecture. This is quite amazing.

## 5 Hardware Model

Machine division is performed to fit a target parallel computer, so we need to describe the hardware configuration in machine-readable form. Fig. 8 shows a hardware model, and is generated by graphViz program — from machine readable `.dot` graph definition. So there's some hope.

In Fig. 8 some ideas for the hardware model is presented. There are three kinds of node in the graph : processors, storages, and interconnects. Processors and storages can not be directly connected. You must interleave at least one interconnect node.

Each hardware node is labeled in the triplet `method : tag : performance`. The `method` represents the knowledge upon how to use the hardware: for processors, how to perform calculations and access their primary storage; for storages, how to allocate the required amount of storage spaces; for interconnects, how to move data from one side to the other. The `method` field is actually a code generator that takes part of a Distributed OM instructions and generates the appropriate codes for the request. We still have to make surveys and experiments on how we interface these code generators.

The `performance` field shows some quantitative performance of the hardware: for processors, its peak performance; for storage, its capacity; for interconnects, its bandwidth. More performance can be added, such as latency, power consumption, sustained performance of some sort, etc. The `performance` field is used to determine the initial machine division, which will then slightly modified via benchmarking and dynamic load-balancing.

Subgraphs with multiplier label e.g. `144x` are used to represent the identical component in the model.

## 6 API for Orthotope Machine

An orthotope machine definition consists of

- a list of name and shape of Static Orthotopes, and

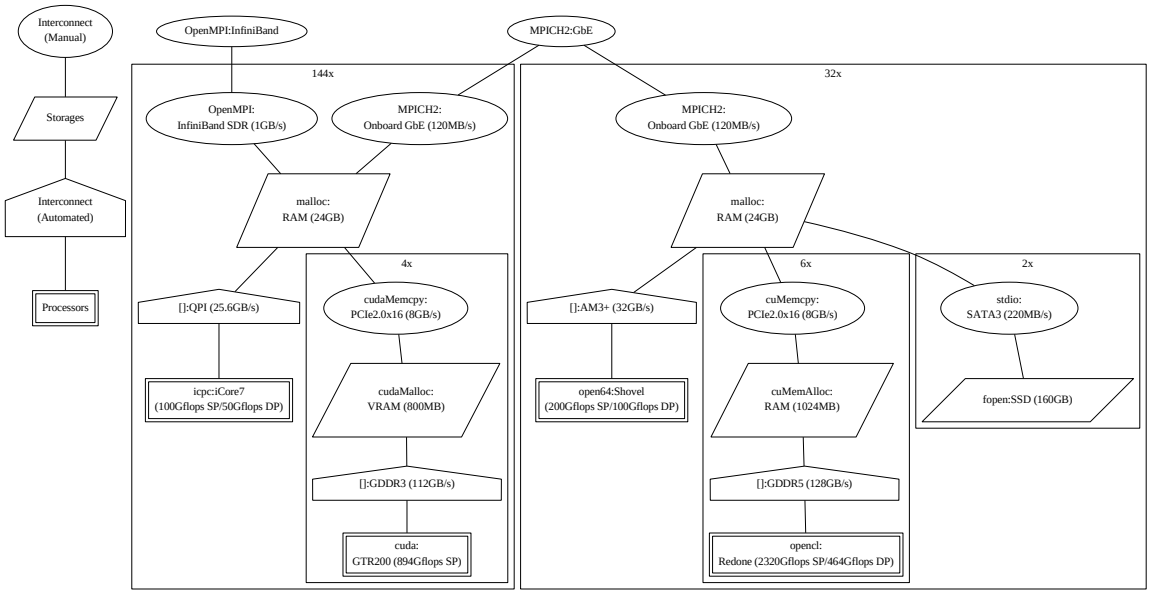


Figure 8: Possible hardware model.

- one or more *kernel* definitions in DPDEL. Kernels are programs that modifies the Static Orthotopes.

There are Static and Local Orthotopes. Static Orthotopes, like static variables, survive kernel calls. On the other hand, Local Orthotopes exist within OM AST. There are lots of them, but most of them don't even get allocated, due to loop fusions and other optimizations.

Kernels are large-grained portion of solver program. We do not attempt right now to write the entire solver in DPDEL; instead, we prefer to write a few kernels and control them from more sophisticated languages. Simple solver may consist of two kernels, one for setting the initial condition of the simulation, and the other is for performing the one step of the simulation.

An API for Distributed Orthotope Machine has

- an interface to read/write any 0-dimensional static orthotopes
- interface for invoking each kernels
- a function that causes the entire machine state saved to a file
- a function that loads from such save-points

This will do for simple simulation tasks. A c++/MPI interface will be a good starting point. Upon constructing the solver, all but rank0 goes to a passive mode, and APIs are called from the rank0 node.

```
Solver s;
s.load("old");
if (mpi_rank!=0) {
  s.passive();
} else {
  s.set_initial_condition();
  for(s.t()=0; s.t() < time_max;
      s.t()+=s.dt()){
    s.proceed();
  }
}
s.save("new");
```

We can be easily imagine interfaces for other languages.

# 7 Instruction Set

Orthotope Machine (OM) is a virtual machine that has an instruction set operating on orthotopes. Each instruction is presented in static single assignment

(SSA) form. For example, `add` instruction in common CPUs or in LLVM adds two scalar value. The `add` instruction in Orthotope Machine takes two orthotope and returns one, acting much like `zipWith (+)`.

## 7.1 Static Addressed Single Assignment

We extend the static single assignment (SSA) form concept to static addressed single assignment(SASA) form. Consider the following DPDEL program, where `a,b,c` are orthotopes:

```
b <- shift (-1) a * shift 1 a;
c <- shift (-1) b + b + shift (+1) b;
```

the program is in SSA (static single assignment) form. If we generate the following code from the DPDEL, it will give a wrong result.

```
double a[N], b[N], c[N];
for (int i=2; i<N-2; ++i) {
  b[i] =a[i-1]*a[i+1];
  c[i] =b[i-1]+b[i]+b[i+1];
}
```

In parallel language such as CUDA, the execution order of the loop is nondeterministic, so the result of the above program is not even well-defined. In SSA (static single assignment) form, every variable is only assigned once. However, there remains some ambiguity. Assigning *each orthotope* variable only once does not fix the meaning of program uniquely in such nondeterministically parallel environment.

One way to fix this is to assure that *each element of each orthotope* is statically assigned at most once. To do this, first we must detect how the output (`c[i]` in this case) depends on every SSA orthotope and how large is the stencil (dependency size). Then we can generate code like this:

```
double a[N], c[N];
for (int i=2; i<N-2; ++i) {
  a_i_minus_2 = a[i-2];
  a_i_minus_1 = a[i-1];
  a_i = a[i];
  a_i_plus_1 = a[i+1];
  a_i_plus_2 = a[i+2];
  b_i_minus_1 = a_i_minus_2 + a_i;
  b_i = a_i_minus_1 + a_i_plus_1;
  b_i_plus_1 = a_i + a_i_plus_2;
  c_i = b_i_minus_1 + b_i + b_i_plus_1;
  c[i] = c_i;
}
```

Let us call this SASA (Static and Addressed Single Assignment) form. Generating programs in SASA form introduces a lot of redundant calculations. But it produces correct results.

At any point of execution timeline, we can insert a global synchronization, thus dividing the abstract syntax tree into several parts, like this:

```
b <- shift (-1) a * shift 1 a;
-- cut here --
c <- shift (-1) b + b + shift (+1) b;
```

From this AST, following code is generated:

```
double a[N], b[N], c[N];
for (int i=1; i<N-1; ++i) {
  a_i_minus_1 = a[i-1];
  a_i_plus_1 = a[i+1];
  b[i] = a_i_minus_1 + a_i_plus_1;
}
for (int i=2; i<N-2; ++i) {
  b_i_minus_1 = b[i-1];
  b_i = b[i];
  b_i_plus_1 = b[i+1];
  c_i = b_i_minus_1 + b_i + b_i_plus_1;
  c[i] = c_i;
}
```

Synchronization insertion reduces the number of arithmetic operations performed per mesh, at the cost of increased number of load/store operations and increased storage consumption. Since the ratio of calculation speed to storage access speed vary from hardware to hardware, we need to search for the best number and place to insert synchronizations for each hardware to achieve maximum performance .

## 7.2 Instruction Set for Primordial Orthotope Machine

Primordial OM is an imaginary machine with infinite amount of memory and infinitely long vector arithmetic instructions.

To begin with, let us assume that each OM has a fixed dimension $n$. An $n$-dimensional OM only deals with $n$-dimensional orthotopes and 0-dimensional orthotopes. 0-dimensional orthotopes are just variables.

See also `http://paraiso-lang.org/wiki/index.php/VVM` for VVM instruction set plan, an ancestor of OM.

For this and next section, let us denote `n`-dimensional Orthotope of element type `a` as `O n a`.

#### reduce and broadcast

```
reduce :: ReduceOp -> O n a-> (O 0 a)
broadcast :: (O 0 a)-> O n a
```

`reduce` creates a 0-dimensional orthotope from an n-dimensional orthotope, by `ReduceOp`, an associative operator. This will possibly use MPI_Reduce, and thus `ReduceOp` will be restricted to one of `Max`, `Min`, `Sum`, `Multiply`.

`broadcast` creates an n-dimensional orthotope from a 0-dimensional orthotope, simply copying its value.

#### shift

```
shift :: (Vector n) -> O n a-> O n a
```

Coordinate-shift the ingredient of the orthotope. The shift vector must be a small, and compile-time constant value.

For example,

```
b <- shift (sx:.sy:.sz) a
```

means

```
for (int z=0; y<NZ; ++z)
  for (int y=0; y<NY; ++y)
    for (int x=0; x<NX; ++x)
      b[z][y][x] = a[z-sz][y-sy][x-sx];
```

#### arithmetic operations

```
imm :: a -> O n a
add :: O n a -> O n a -> O n a
cmp :: O n a -> O n a -> (O n Bool)
hatena :: (O n Bool) -> O n a -> O n a
                      -> O n a
sincos :: O n a -> (O n a, O n a)
```

There are arithmetic operations of various kind. Arithmetic operations can generally have $n_i$ input and $n_o$ output, but all the reads and writes go to the same coordinate in an orthotope.

Possibly, we can summarize the various arithmetic operations to a single polymorphic instruction `arith`. We can think of a way to store the information on generating arithmetic operations for various hardware in pluggable manner.

The shapes of the output orthotopes of an `arith` are the intersection of all the input orthotopes.

### 7.3 Instruction Set for Distributed Orthotope Machine

Distributed Orthotope Machine operates on an Orthotree. Each leaf of the Orthotree is tied to a Storage in hardware representation. Orthotope now has the form `(O (Maybe Storage) n a)`, where `Maybe Storage` denotes the storage the orthotope is stored. If it's `Nothing`, then it's a global value (possibly stored at MPI rank 0.)

#### communication

```
communicate :: Storage -> Storage -> O s n a
            -> O s n a
```

Move data from one storage to other.

#### split and merge

```
split :: Cutter -> O s n a
      -> (O s n a, O s n a)
merge :: Cutter -> O s n a -> O s n a
      -> O s n a
```

`split` divides an orthotope into two, and `merge` does the opposite. As is mentioned in §4.2, `merge` instruction is not issued arbitrary, but only for those orthotopes that was once united. As long as this rule is obeyed, there's no worry of an impossible merge request.

## 8 Possible Program Transformations

### 8.1 Common Techniques

A lot of important optimization techniques are known for scalar processors. Some of them are equally applicable to Orthotope Machine AST. Even if we emit the code without such optimizations, we can expect the native compilers do the job for us. At least Paraiso should not hinder native optimizations. It's better if we can optimize OM AST using modern optimization platform such as LLVM.

**Constant Folding** Constant folding is to calculate constant expression in compile time. It'll be fairly easy, and if we forget to do it, native compilers are also good at it. One very stupid thing is to store a globally constant value onto an array. We should avoid this.

**Loop Fusion** Loop fusion is a source of high performance in Haskell's array operating libraries `Data.Vector` and `Data.Array.Repa` [KCL+10], and also in many other libraries in other languages.

**Common Subexpression Elimination** To avoid performing the same calculation twice, by storing the intermediate value to the memory.

## 8.2 Time-step Fusion

Fuse several time-step into one kernel, and access to the storage only at the beginning and end of the fused kernel. This increases the ratio of calculation per storage access. This is easier if the time-step is always a constant and we do not need a global reduce within a kernel. The simulation for relativistic systems meets such criteria because time-step is always bound by the speed of light.

## 8.3 Use of Scratchpad

Some processors have scratchpad regions, where programmers manually store some data and have fast access. GPU's scratchpad has size of tens of kilobytes. On the other hand, when we attempt a ultra-high-resolution simulation where we store the simulation state on larger but slower storage such as SSDs, the main memory itself can be regarded as large scratchpad.

## 8.4 AST Manipulations by User Specification

One of the reason High Performance Fortran (HPF) has failed [KKZ07] is because the user of HPF did not have fine-grained control over the generated code. First of all, the machine generated code like Fig. 6 is unreadable.

I'm thinking of introducing an annotation operator like Parsec's `<?>` (c.f. `http://bit.ly/ef15ty`) to Paraiso. You can choose the name for the AST nodes so that the code is more tractable. One good idea is that every identifier that passes `qb` gets annotated as well as antiquoted.

AST is a large, directed acyclic graph and we have large degree of freedom in how we fold it into one time-line. Here, users can also add pragmas such as `(<?> balloon)` and `(<?> stone)` to the DPDEL code portion, so that the portion get executed as soon/late as possible.

## 8.5 Synchronization Insertion

This issue is mentioned in §7.1. At any point of execution time-line, we can insert a global synchronization, thus dividing the abstract syntax tree into several parts. This reduces the number of arithmetic operations performed per mesh, at the cost of increased number of load/store operations and increased storage consumption.

The paragraphs of the programs, such as steps of the higher order Runge-Kutta method, are good candidate for synchronization insertion. Often the actual programs are written so. So before implementing an automated synchronization insertion, we can let users provide synchronization point candidates, and try and benchmark on them.

## 8.6 Trapezium Splitting

Trapezium splitting (c.f. Fig. 9) is a transformation of one computation to a set of computation that requires less storage space. To perform the trapezium splitting, first split the output orthotope into several smaller portions. Then for each of the small orthotope, load the portion of the input such that the output portion depends on, and calculate the answer for the portion. Repeat this process until you obtain results for all the output.

This method is fairly easy, and can be a building block for other optimization such as use of scratchpad and computation / communication overlapping. However, trapezium splitting introduces duplicated computations.

## 8.7 Parallelogram Splitting

Parallelogram splitting (c.f. Fig. 10) is an improved version of trapezium splitting. The difference is that in parallelogram splitting, after every computation for a portion you keep the "rightmost states" within the scratchpad so that duplicated computations do not happen.

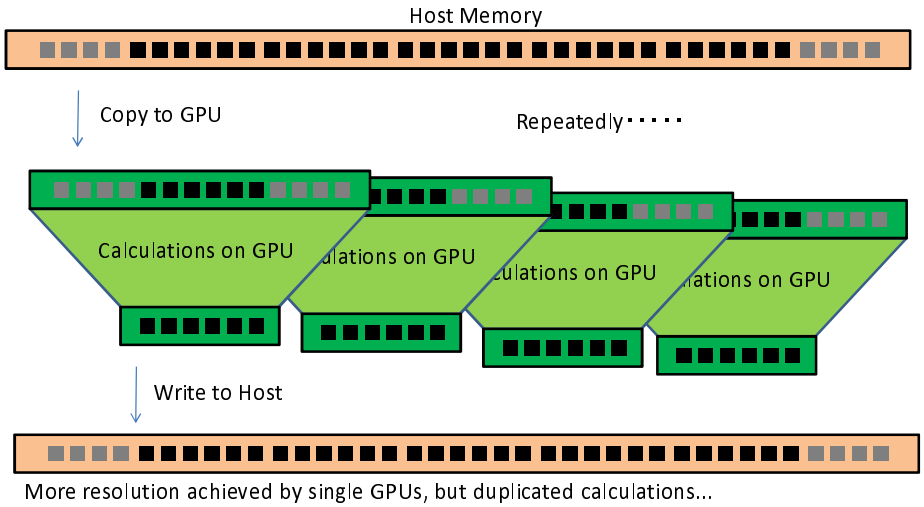


Figure 9: Trapezium Splitting

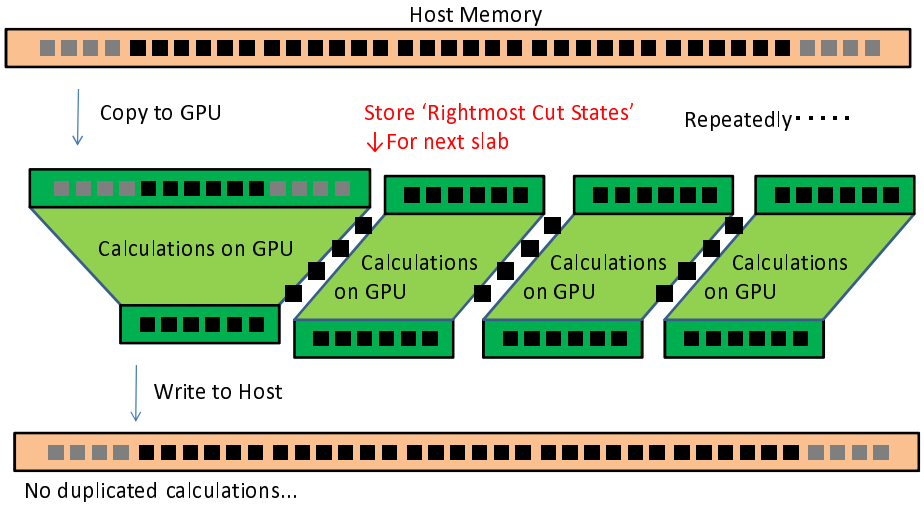


Figure 10: Parallelogram Splitting

### 8.8 Block Skew

Fluid consists of multiple orthotope of the same shape and extent. As in Fig. 2, some components are half-mesh shifted. If we shift some fluid components by half or one mesh, it may alter the performance.

### 8.9 Batch Reduce/Broadcast

Reduce and broadcast needs global communications. To have large-grained communications, it will be a good idea to perform many reduce/broadcast instructions at once.

### 8.10 Overlapped Computation and Communication

It's good if we can detect the portions of calculations that do not depend on the broadcast data, and perform such calculations in parallel while the communications or reduces/broadcasts are going on. We may need trapezium or parallelogram splitting to detach some of the computation regions from the rest that are waiting for incoming data.

## References


[CDM+10] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. *SIGPLAN Not.*, 45:835–847, 2010.

[CKL+11] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[CKLP01] Manuel Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal nested data parallelism in haskell. In Rizos Sakellariou, John Gurd, Len Freeman, and John Keane, editors, *Euro-Par 2001 Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 524–534. Springer Berlin / Heidelberg, 2001.

[Dat09] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[DMV+08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[KCL+10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *SIGPLAN Not.*, 45:261–272, September 2010.

[KKZ07] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[KSSU86] Chisato Kon'no, Miyuki Saji, Nobutoshi Sagawa, and Yukio Umetani. Advanced implicit solution function of deqsol and its evaluation. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1026–1033, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[MM10] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.

[NCY15] SAGAWA NOBUTOSHI, KON'NO CHISATO, and UMETANI YUKIO.

Numerical simulation language deqsol. *Transactions of Information Processing Society of Japan*, 30(1):36–45, 1989-01-15.

[NY15] Sagawa Nobutoshi and Umetani Yukio. Basic research on deqsol for parallel processors. *Zenkoku Taikai Kouen Ronbun Shu*, 38(3):1492–1493, 1989-03-15.

[PJ08] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 138–138. Springer Berlin / Heidelberg, 2008.

[TLP02] P. W. TRINDER, H.-W. LOIDL, and R. F. POINTON. Parallel and distributed haskells. *Journal of Functional Programming*, 12(4-5):469–510, 2002.