Hakubi 白眉 Kyoto Univ.

偏微分方程式シミュレーションのための並列DSL「Paraiso」計画

村主崇行　@nushio

宇宙物理学者

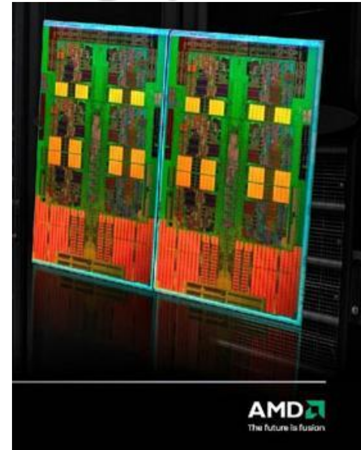
京都大学白眉センター　特定助教

Monadiusの作者です

高速化とは

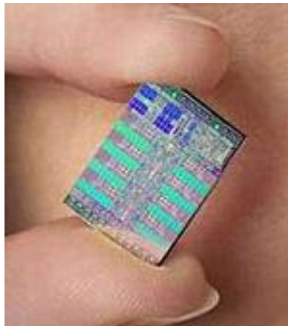
並列化である

MagnyCours



4x(8or12)コア

Cell B.E.



1+(4or8)x8コア

GTX 295

GPU

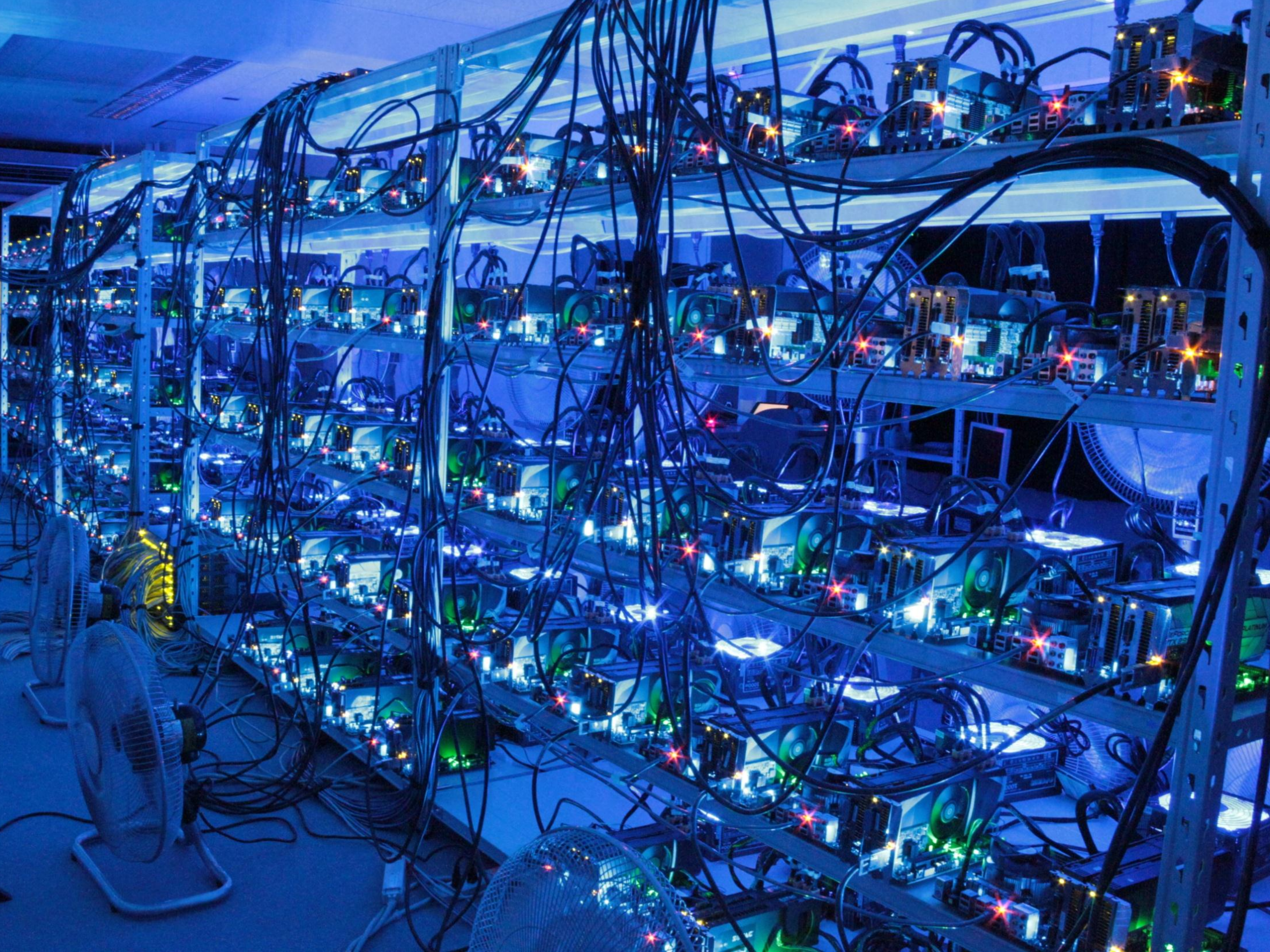
8x30コアx2

GPUのさらに高い並列計算性能

	Neharem	G200 (GTX285)
ベクトル長	4	8
コア数	4	30
レジスタ数	40?×4	4096×30
スレッド数	2×4	1024×30

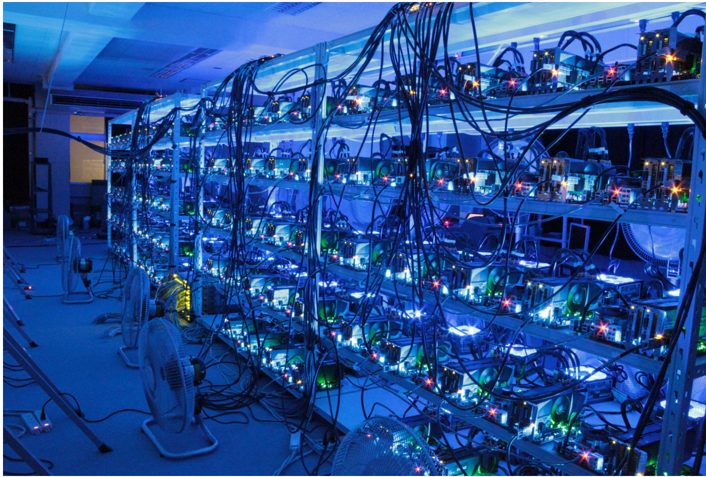
GPUの設計：**レジスタが圧倒的に多い**
→レジスタを退避させることなく**膨大な数のスレッド**を駆動
→様々なレイテンシを隠蔽

どんなマシンを使うのか？



DEGIMA

DEstination of **G**PU **I**ntensive **MA**chines



- 長崎大学「超並列メニーコアコンピューティングセンター」のGPUクラスタ led by 濱田剛
- 「エッジなスパコン」

- 約800×NVIDIA GT200 GPU型演算器
- うち576個はInfiniBandで連結
- 単精度ピーク演算性能: 514.9TFlop/s
- 最大1769’4720スレッドを同時実行可能
- 総ビデオメモリ: 約460GB ビデオメモリ帯域幅: 64.454TB/s

TSUBAME

- 東京工業大学TSUBAME グリッドクラスタ

平成22年度学際大規模情報基盤共同利用･共同研究拠点公募型共同研究(試行)

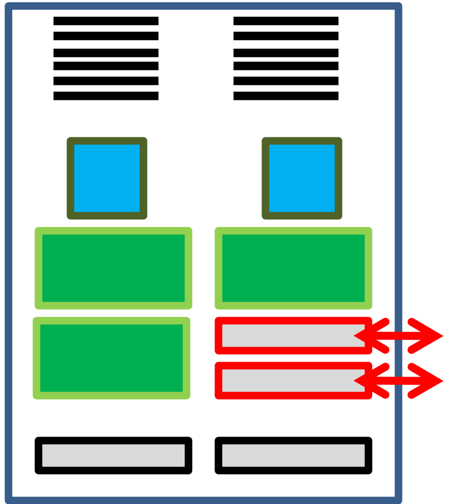
- 共同利用に供されているGPUスパコン
- GPUが利用できる最大のキュー = hpc1tes2

- 120×NVIDIA GT200 GPU型演算器
- 単精度ピーク演算性能: 124.2TFlop/s
- 最大122’880スレッドを同時実行可能
- 総ビデオメモリ: 480GB　ビデオメモリ帯域幅: 1.224TB/s

TSUBAME 2.0 メモリヒエラルキ(一部推測)

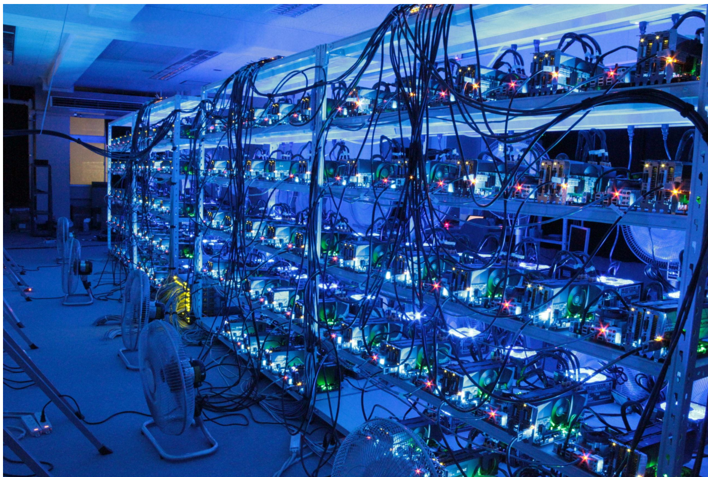
Hakubi

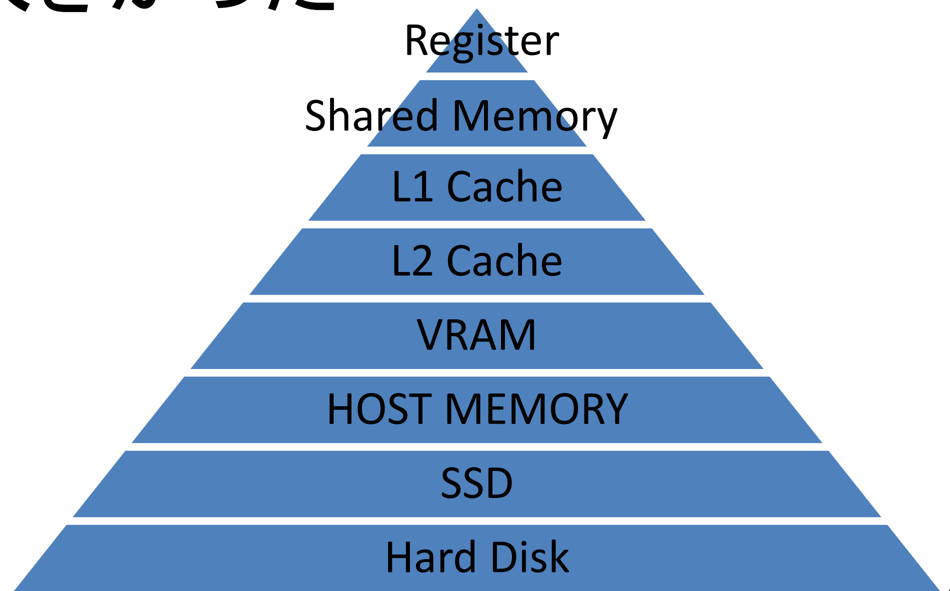
1ノード
(平民ノード)



メモリ: (4GBx6) + (8GBx3 + 2GBx3)
CPU : **Westmere EP** x2
GPU : **Tesla 2050** (515Gflops + 3GB) x3
通信 : Infiniband QDR 10GB/s
ローカルディスク : SSD x2 RAID0 (460MB/s read)

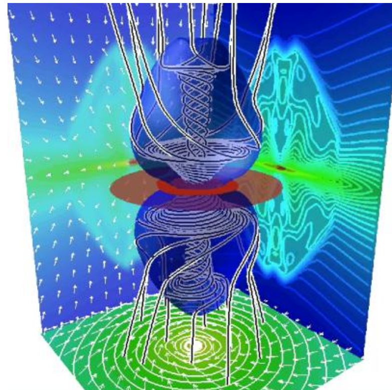
- アーキテクトたちの努力のおかげで、今日ではすばらしい演算性能・莫大なメモリを持った計算機が使えるようになった。
- だが、その代償も大きかった


最大1769’4720スレッドを同時実行

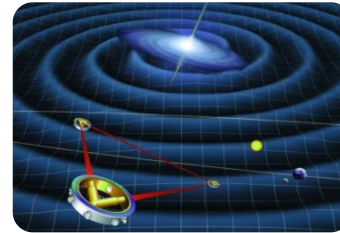


どのような計算をしたいのか？

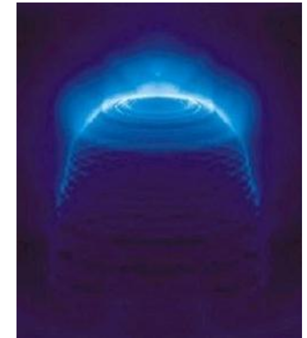
流体力学
（気体・液体）



磁気流体力学
（プラズマ）



一般相対性理論
（時空・座標）



輻射輸送
（光の放出・反射・
吸収・伝搬）

- 解きたい偏微分方程式があって、それを数値アルゴリズムに変換して解く。
- 最大フロー(解きたい問題)に対しフォードファルカーソン法・エドモンドカープ法など(アルゴリズム)があるようなもの

偏微分方程式の陽解法とは？

- 流体などを、3次元配列で表す。
- 偏微分方程式を「3次元、実数セルオートマトンのルール」に変換する。
- 世代を進めていく。
- 各セルの次世代の状態は、前の世代の近所のセルの状態から決まる。

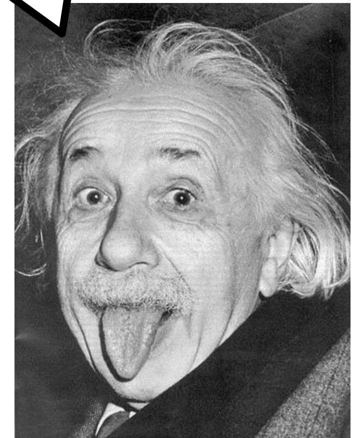
Hakubi Kyoto Univ.

どんな方程式を解きたいのか？

- たとえば、一般相対性理論の方程式であるアインシュタイン方程式は・・・

$$R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu}$$

ね、簡単でしょう？



どんなアルゴリズムで解くのか？

- アインシュタイン方程式を解くためのBSSN法は･･･

B. Einstein's equation

Our formulation for Einstein's equations is the same as in [6] in three spatial dimensions and in [31] in axial symmetry. Here, we briefly review the basic equations in our formulation. Einstein's equations are split into constraint and evolution equations. The Hamiltonian and momentum constraint equations are written as

$$R_k^k - \tilde{A}_{ij}\tilde{A}^{ij} + \frac{2}{3}K^2 = 16\pi\rho_H, \quad (26)$$

$$D_i\tilde{A}^i{}_j - \frac{2}{3}D_jK = 8\pi J_j, \quad (27)$$

or, equivalently

$$\tilde{\Delta}\psi = \frac{\psi}{8}\tilde{R}_k{}^k - 2\pi\rho_H\psi^5 - \frac{\psi^5}{8}\left(\tilde{A}_{ij}\tilde{A}^{ij} - \frac{2}{3}K^2\right), \quad (28)$$

$$\tilde{D}_i(\psi^6\tilde{A}^i{}_j) - \frac{2}{3}\psi^6\tilde{D}_jK = 8\pi J_j\psi^6, \quad (29)$$

where $\psi \equiv e^{\phi}$. These constraint equations are solved to set initial conditions. A method in the case of GRMHD is presented in Sec. IV.

In the following of this subsection, we assume that Einstein's equations are solved in the Cartesian coordinates (x, y, z) for simplicity. Although we apply the implementation described here to axisymmetric issues as well as nonaxisymmetric ones, this causes no problem since Einstein's equations in axial symmetry can be solved using the so-called Cartoon method in which an axisymmetric boundary condition is appropriately imposed in the Cartesian coordinates [31–33]: In the Cartoon method, the field equations are solved only in the $y = 0$ plane, and grid points of $y = \pm\Delta x$ (Δx denotes the grid spacing in the uniform grid) are used for imposing the axisymmetric boundary conditions.

We solve Einstein's evolution equations in our latest BSSN formalism [6,29]. In this formalism, a set of variables $(\tilde{\gamma}_{ij}, \phi, \tilde{A}_{ij}, K, F_i)$ are evolved. Here, we adopt an auxiliary variable $F_i \equiv \delta^{jl}\partial_l\tilde{\gamma}_{ij}$ that is the one originally proposed and different from the variable adopted in [10] in which $\partial_i\tilde{\gamma}^{ij}$ is used. Evolution equations for $\tilde{\gamma}_{ij}$, ϕ, $\tilde{A}_{ij}$, and K are

$$(\partial_t - \beta^l\partial_l)\tilde{\gamma}_{ij} = -2\alpha\tilde{A}_{ij} + \tilde{\gamma}_{ik}\beta^k{}_{,j} + \tilde{\gamma}_{jk}\beta^k{}_{,i} - \frac{2}{3}\tilde{\gamma}_{ij}\beta^k{}_{,k}, \quad (30)$$

$$\begin{aligned}(\partial_t - \beta^l\partial_l)\tilde{A}_{ij} = e^{-4\phi}&\left[\alpha\left(R_{ij} - \frac{1}{3}e^{4\phi}\tilde{\gamma}_{ij}R_k{}^k\right) - \left(D_iD_j\alpha - \frac{1}{3}e^{4\phi}\tilde{\gamma}_{ij}\Delta\alpha\right)\right] \\ &+ \alpha(K\tilde{A}_{ij} - 2\tilde{A}_{ik}\tilde{A}_j{}^k) + \beta^k{}_{,i}\tilde{A}_{kj} \\ &+ \beta^k{}_{,j}\tilde{A}_{ki} - \frac{2}{3}\beta^k{}_{,k}\tilde{A}_{ij} \\ &- 8\pi\alpha\left(e^{-4\phi}S_{ij} - \frac{1}{3}\tilde{\gamma}_{ij}S_k{}^k\right), \end{aligned} \quad (31)$$

$$(\partial_t - \beta^l\partial_l)\phi = \frac{1}{6}(-\alpha K + \beta^k{}_{,k}), \quad (32)$$

$$(\partial_t - \beta^l\partial_l)K = \alpha\left[\tilde{A}_{ij}\tilde{A}^{ij} + \frac{1}{3}K^2\right] - \Delta\alpha + 4\pi\alpha(\rho_H + S_k{}^k). \quad (33)$$

For a solution of ϕ, the following conservative form may be adopted [6]:

$$\partial_t e^{6\phi} - \partial_l(\beta^l e^{6\phi}) = -\alpha K e^{6\phi}. \quad (34)$$

For computation of R_{ij} in the evolution equation of $\tilde{A}_{ij}$, we decompose

$$R_{ij} = \tilde{R}_{ij} + R^{\phi}_{ij}, \quad (35)$$

where

$$R^{\phi}_{ij} = -2\tilde{D}_i\tilde{D}_j\phi - 2\tilde{\gamma}_{ij}\tilde{\Delta}\phi + 4\tilde{D}_i\phi\tilde{D}_j\phi - 4\tilde{\gamma}_{ij}\tilde{D}_k\phi\tilde{D}^k\phi, \quad (36)$$

$$\tilde{R}_{ij} = \frac{1}{2}\left[\delta^{kl}(-h_{ij,kl} + h_{ik,lj} + h_{jk,li}) + 2\partial_k(f^{kl}\Gamma_{l,ij}) - 2\Gamma^l_{kj}\Gamma^k_{il}\right]. \quad (37)$$

In Eq. (37), we split $\tilde{\gamma}_{ij}$ and $\tilde{\gamma}^{ij}$ as $\delta_{ij} + h_{ij}$ and $\delta^{ij} + f^{ij}$, respectively. Γ^k_{ij} is the Christoffel symbol with respect to $\tilde{\gamma}_{ij}$, and $\Gamma_{k,ij} = \tilde{\gamma}_{kl}\Gamma^l_{ij}$. Because of the definition $\det(\tilde{\gamma}_{ij}) = 1$ (in the Cartesian coordinates), we use $\Gamma^k_{ki} = 0$.

In addition to a flat Laplacian of h_{ij}, $\tilde{R}_{ij}$ involves terms linear in h_{ij} as $\delta^{kl}h_{ik,lj} + \delta^{kl}h_{jk,li}$. To perform numerical simulation stably, we replace these terms by $F_{i,j} + F_{j,i}$. This is the most important part in the BSSN formalism, pointed out originally by Nakamura [26]. The evolution equation of F_i is derived by substituting Eq. (30) into the momentum constraint as

$$\begin{aligned}(\partial_t - \beta^l\partial_l)F_i = -16\pi\alpha J_i &+ 2\alpha\left[f^{kj}\tilde{A}_{ik,j} + f^{kj}{}_{,j}\tilde{A}_{ik} - \frac{1}{2}\tilde{A}^{jl}h_{jl,i} + 6\phi_{,k}\tilde{A}^k{}_i - \frac{2}{3}K_{,i}\right] \\ &+ \delta^{jk}\left[-2\alpha_{,k}\tilde{A}_{ij} + \beta^l{}_{,k}h_{ij,l} + \left(\tilde{\gamma}_{il}\beta^l{}_{,j} + \tilde{\gamma}_{jl}\beta^l{}_{,i} - \frac{2}{3}\tilde{\gamma}_{ij}\beta^l{}_{,l}\right)_{,k}\right]. \end{aligned} \quad (38)$$

We also have two additional notes for handling the evolution equation of $\tilde{A}_{ij}$. One is on the method for evaluation of $R_k{}^k$ for which there are two options, use of the Hamiltonian constraint and direct calculation by

$$R_{ij}\gamma^{ij} = e^{-4\phi}(\tilde{R}_k{}^k + R^{\phi}_{ij}\tilde{\gamma}^{ij}). \quad (39)$$

We always adopt the latter one since with this, the conservation of the relation $\tilde{A}_{ij}\tilde{\gamma}^{ij} = 0$ is much better preserved. The other is on the handling of a term of $\tilde{\gamma}^{ij}\delta^{kl}h_{ij,kl}$ which appears in $\tilde{R}_k{}^k$. This term is written by

$$\tilde{\gamma}^{ij}\delta^{kl}h_{ij,kl} = -\delta^{kl}h_{ij,k}f^{ij}{}_{,l}, \quad (40)$$

where we use $\det(\tilde{\gamma}_{ij}) = 1$ (in the Cartesian coordinates).

As the time slicing condition, an approximate maximal slice condition $K \approx 0$ is adopted following previous papers (e.g., [34]). As the spatial gauge condition, we adopt a hyperbolic gauge condition as in [6,35]. Successful numerical results for merger of binary neutron stars and stellar core collapse in these gauge conditions are presented in [6–8,24]. We note that these are also different from those in [10].

どんなコードで解くのか？

Hakubi
Kyoto Univ.

もっと美しく書けないのか？

- たぶん、可能

- ベクトルやテンソルをクラスとして定義すれば
- 演算子オーバーロードとか使えば
- テンプレートとか使えば
- エクスプレッション・テンプレートとか使えば

問題

- コードを書くのは一回きりではない

- よいアルゴリズムを探すために試行錯誤したい
- 3次元セル配列のメモリへの格納順序とか変えたくなるかもしれない
- アーキテクチャが変わってSSEだのCUDAだの使いたくなるかもしれない
- 分散型計算機つかうにはMPI通信なども必要

上のうちどれか1つの変更をとっても、コードのかなりの部分から必要個所を探して修正が必要（つらい・・・）

Hakubi Kyoto Univ.

モジュール化されていて
移植が楽で
ちゃんと速度も出るコードを
美しく書けないのか？

- 絶望的

Hakubi Kyoto Univ.

人間がコードを書くのに必要な知識のバイト数 << 実際のコードのバイト数

いわばこんな風になっている

プログラムの総行数は、部品ごとの知識量の積

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

方程式の数、積分法や補間法の次数、ベクトルの要素数、ハードウェアを複数使う場合はその数・・・

それをこんな風にしたい！

プログラムを書くのに必要な知識を個別に与えさえすれば、コードが生成される

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

Hakubi

- シミュレーションを行うのに必要な、代数構造、物理的方程式を解くためのアルゴリズム、時間積分法、空間補間法、最適化技術、ハードウェアの知識などを、**モジュラーで、自由に再利用・組み合わせできる形**で表現できるような言語を作りたい。
- 偏微分方程式の陽解法に限定

Paraiso

数式処理システム

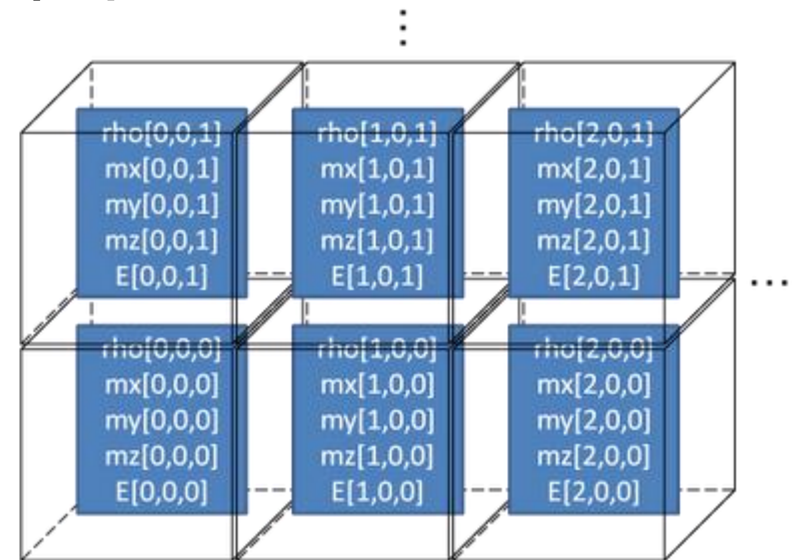
＋3次元、実数セルオートマトンのコードジェネレータ　for Distributed, Accelerated Machines

Hakubi

Virtual Vector Machine

- 実数セルオートマトンに対応する仮想マシン
- 3次元配列状にならんだレジスタを持つ
- 演算命令は基本的に全セルに並列に作用
- 隣のセルからロードする命令なども

実行するための仮想マシンではなく、
データフローグラフを構築するための仮想マシン



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

$$\boldsymbol{Q}_i = \begin{pmatrix} m_i \\ \boldsymbol{p}_i \\ E_i \end{pmatrix} = \int_{V_i} \boldsymbol{U}\,\mathrm{d}V.$$

$$\boldsymbol{Q}_i^{(n+1)} = \boldsymbol{Q}_i^{(n)} - \Delta t \sum_j A_{ij} \hat{\boldsymbol{F}}_{ij}^{(n+1/2)},$$

基礎方程式

```
d_dt (q [i])
   = (a [i,j]) (f [j])
f [j] = … …
```

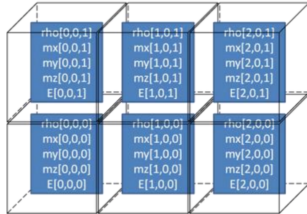
さしあたり**人手**

離散化形

自動

VVM上のコード

```
ld  r2, g2[0,0,0]
ld  r1, g2[0,0,1]
add r1,r2,r3
st  r3,g1
```



仮想ベクトルマシン：VVM

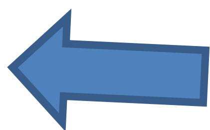
自動

実マシン上のコード

```
*q=cudaMalloc(…);
__shared__ a,b;
a=q[idx];
b=q[idx+1];
p[idx]=a+b;
```

既存コンパイラ

実マシン上の実行ファイル



結果

Paraiso 2008

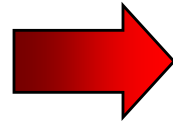
Hakubi Kyoto Univ.

- プロトタイプ。
- 偏微分方程式が対象ではない。単に多数の並列した計算を行うようなコードを生成する
- GPGPUをもふもふする会でも使ったり

Paraiso2008 の文法とコード生成

```
<->[X] 0+ MainLogistic.hs
import System.Environment
import Paraiso

main = do
  args <- getArgs
  let arch = if "--cuda" `elem` args then
               CUDA 128 128
             else
               X86
  putStrLn $ compile arch code
    where
      code = do
        parallel 16384 $ do
          r <- allocate
          x <- allocate
          r =$ Rand  0.0  (4.0::Double)
          x =$ Rand  0.0  1.0
          cuda $ do
            sequential 65536 $ do
              r =$ r * x * (1-x)
          output [r,x]
```

Paraiso Code



```
<->[X] 0+ main.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

double drand (double lo, double hi) {
  return lo + rand () / (double) RAND_MAX *(hi - lo);
}

int main (int argc, char **argv) {
  double a2[16384];
  double a3[16384];
  for (int a1 = 0; a1 < 16384; ++a1) {
    a2[a1] = drand (0.0, 4.0);
    a3[a1] = drand (0.0, 1.0);
    for (int a4 = 0; a4 < 65536; ++a4) {
      a2[a1] = ((a2[a1] * a3[a1]) * (1.0 - a3[a1]));
    } cout << a2[a1] << " " << a3[a1] << endl;
  }
  return 0;
}
```

C++ code

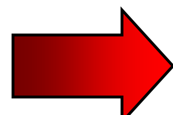
- parallel と sequentialとでループを生成
- allocate でメモリを確保
- 四則演算とかはいつも通り

ハードウェアに特化したコードの生成

Hakubi

Paraiso Code

```
<->[X] 0+ MainLogistic.hs
import System.Environment
import Paraiso

main = do
  args <- getArgs
  let arch = if "--cuda" `elem` args then
               CUDA 128 128
             else
               X86
  putStrLn $ compile arch code
    where
      code = do
        parallel 16384 $ do
          r <- allocate
          x <- allocate
          r =$ Rand  0.0  (4.0::Double)
          x =$ Rand  0.0  1.0
          cuda $ do
            sequential 65536 $ do
              r =$ r * x * (1-x)
          output [r,x]
```



CUDA：
nvidia GPUの言語

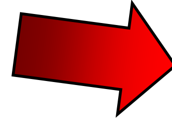
```
<->[X] 0+ main.cu
#include <iostream>
#include <cstdlib>
#include <cutil.h>
using namespace std;

double drand (double lo, double hi) {
  return lo + rand () / (double) RAND_MAX *(hi - lo);
}

__global__ void function_on_GPU_a4 (float *a2_dev, float *a3_dev) {
  int a1 = blockIdx.x * gridDim.x + threadIdx.x;
  float a2_sha;
  a2_sha = a2_dev[a1];
  float a3_sha;
  a3_sha = a3_dev[a1];
  for (int a4 = 0; a4 < 65536; ++a4) {
    a2_sha = ((a2_sha * a3_sha) * (1.0 - a3_sha));
  }
  a2_dev[a1] = a2_sha;
  a3_dev[a1] = a3_sha;
}

int main (int argc, char **argv) {
  dim3 grids (128);
  dim3 threads (128);
  float *a2 = (float *) malloc (sizeof (float) * 16384);
  float *a2_dev;
  cudaMalloc ((void **) &a2_dev, sizeof (float) * 16384);
  float *a3 = (float *) malloc (sizeof (float) * 16384);
  float *a3_dev;
  cudaMalloc ((void **) &a3_dev, sizeof (float) * 16384);
  for (int a1 = 0; a1 < 16384; ++a1) {
    a2[a1] = drand (0.0, 4.0);
    a3[a1] = drand (0.0, 1.0);
  }
  cudaMemcpy (a2_dev, a2, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  cudaMemcpy (a3_dev, a3, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  function_on_GPU_a4 <<< grids, threads >>> (a2_dev, a3_dev);
  cudaMemcpy (a2, a2_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  cudaMemcpy (a3, a3_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  for (int a1 = 0; a1 < 16384; ++a1) {
    cout << a2[a1] << " " << a3[a1] << endl;
  }
  return 0;
}
```

- 同一のParaisoコードから、多様なハードウェア向けの言語を生成できる。

数学構造の扱い

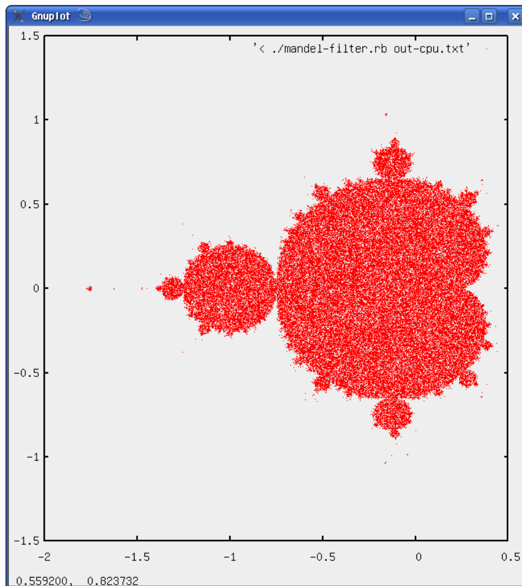
Paraiso Code

```
code = do
  parallel 1048576 $ do
    c <- allocate
    z <- allocate
    c =$ (Rand (-2.0) 2.0) :+ (Rand (-2.0) 2.0)
    z =$ (0 :+ 0 :: Complex (Expr Double))
    cuda $ do
      sequential 256 $ do
        z =$ z * z + c
    output [realPart c,imagPart c, realPart z, imagPart z]
```



C++ code

```
for (int a1 = 0; a1 < 1048576; ++a1) {
  a6[a1] = drand (-2.0, 2.0);
  a3[a1] = drand (-2.0, 2.0);
  a2[a1] = a6[a1];
  a7[a1] = 0.0;
  a5[a1] = 0.0;
  a4[a1] = a7[a1];
  for (int a8 = 0; a8 < 256; ++a8) {
    a9[a1] = (((a4[a1] * a4[a1]) - (a5[a1] * a5[a1])) + a2[a1]);
    a5[a1] = (((a4[a1] * a5[a1]) + (a5[a1] * a4[a1])) + a3[a1]);
    a4[a1] = a9[a1];
  }
  cout << a2[a1] << " " << a3[a1] << " " << a4[a1] << " " << a5[a1]
       << endl;
}
```



example: drawing Mandlebrot set

232Gflops on GPU

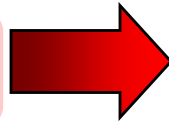
1.15Gflops on CPU（1th）

- 複素数やベクトルといった構造、内積とかの演算が使える。
- <u>普通の数に定義された演算がコード生成にもそのまま使える。</u>
- 基本演算に分解されるので、C++とかでクラスを使ったような場合のオーバヘッドもない。

アルゴリズムの生成

```
code = do
  parallel 16384 $ do
    x <- allocate
    p <- allocate
    x =$ Rand 0.1 (1.0::Double)
    cuda $ do
       integrate4 0.01 2.56 $ [
             d_dt x $ p*x
                ]
    output [x]
```

Paraiso Code



```
for (int a10 = 0; a10 < 256; ++a10) {
  a5[a1] = a2[a1];
  a6[a1] = (a3[a1] * a2[a1]);
  a4[a1] = (a4[a1] + 5.0e-3);
  a2[a1] = (a5[a1] + (5.0e-3 * a6[a1]));
  a7[a1] = (a3[a1] * a2[a1]);
  a2[a1] = (a5[a1] + (5.0e-3 * a7[a1]));
  a8[a1] = (a3[a1] * a2[a1]);
  a4[a1] = (a4[a1] + 5.0e-3);
  a2[a1] = (a5[a1] + (1.0e-2 * a8[a1]));
  a9[a1] = (a3[a1] * a2[a1]);
  a2[a1] =
    (a5[a1] +
     (1.6666666666666666e-3 *
      (((a6[a1] + (2.0 * a7[a1])) + (2.0 * a8[a1])) + a9[a1])));
}
```

C++ code

- Paraisoで古典的ルンゲクッタ積分を生成するコードを書いた。
- ハミルトニアンをあたえたら機械的に微分してsymplectic積分を生成とかもできるだろう。

簡単な使用例：ローレンツアトラクタ

Hakubi Kyoto Univ.

基礎方程式

$$\frac{dx}{dt} = -px + py$$
$$\frac{dy}{dt} = -xz + rx - y$$
$$\frac{dz}{dt} = xy - bz$$

Paraiso Code

```
code = do
  parallel 16384 $ do
    x <- allocate
    y <- allocate
    z <- allocate
    x =$ Rand 10 (20.0::Double)
    y =$ Rand 10 20
    z =$ Rand 10 20
    cuda $ do
      p <- allocate
      r <- allocate
      b <- allocate
      p =$ 10
      r =$ 28
      b =$ 8/3
      integrate4 0.01 (10 * 655.36) $ [
            d_dt x $ - p*x + p*y      ,
            d_dt y $ - x*z + r*x - y ,
            d_dt z $   x*y - b*z      ]
    output [x,y,z]
```

CUDA code

```
#include <iostream>
#include <cstdlib>
#include <cutil.h>
using namespace std;
double
drand (double lo, double hi)
{
  return lo + rand () / (double) RAND_MAX *(hi - lo);
}
__global__ void
function_on_GPU_a5 (float *a2_dev, float *a3_dev, float *a4_dev)
{
  int a1 = blockIdx.x * gridDim.x + threadIdx.x;
  float a2_sha;
  a2_sha = a2_dev[a1];
  float a3_sha;
  a3_sha = a3_dev[a1];
  float a4_sha;
  a4_sha = a4_dev[a1];
  __shared__ float a5_sha;
  __shared__ float a6_sha;
  __shared__ float a7_sha;
  a5_sha = 10.0;
  a6_sha = 28.0;
  a7_sha = (8.0 / 3.0);
  __shared__ float a8_sha;
  a8_sha = 0.0;
  __shared__ float a9_sha;
  __shared__ float a10_sha;
  __shared__ float a11_sha;
  __shared__ float a12_sha;
  __shared__ float a13_sha;
  __shared__ float a14_sha;
  __shared__ float a15_sha;
  __shared__ float a16_sha;
  __shared__ float a17_sha;
  __shared__ float a18_sha;
  __shared__ float a19_sha;
  __shared__ float a20_sha;
  __shared__ float a21_sha;
  __shared__ float a22_sha;
  __shared__ float a23_sha;
  for (int a24 = 0; a24 < 655360; ++a24) {
    a9_sha = a2_sha;
    a10_sha = a3_sha;
    a11_sha = a4_sha;
    a12_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a13_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a14_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a8_sha = (a8_sha + 5.0e-3);
    a2_sha = (a9_sha + (5.0e-3 * a12_sha));
    a3_sha = (a10_sha + (5.0e-3 * a13_sha));
    a4_sha = (a11_sha + (5.0e-3 * a14_sha));
    a15_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a16_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a17_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a2_sha = (a9_sha + (5.0e-3 * a15_sha));
    a3_sha = (a10_sha + (5.0e-3 * a16_sha));
    a4_sha = (a11_sha + (5.0e-3 * a17_sha));
    a18_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a19_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a20_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a8_sha = (a8_sha + 5.0e-3);
    a2_sha = (a9_sha + (1.0e-2 * a18_sha));
    a3_sha = (a10_sha + (1.0e-2 * a19_sha));
    a4_sha = (a11_sha + (1.0e-2 * a20_sha));
    a21_sha = ((0.0 - (a5_sha * a2_sha)) + (a5_sha * a3_sha));
    a22_sha = (((0.0 - (a2_sha * a4_sha)) + (a6_sha * a2_sha)) - a3_sha);
    a23_sha = ((a2_sha * a3_sha) - (a7_sha * a4_sha));
    a2_sha =
      (a9_sha +
       (1.6666666666666666e-3 *
        (((a12_sha + (2.0 * a15_sha)) + (2.0 * a18_sha)) + a21_sha)));
    a3_sha =
      (a10_sha +
       (1.6666666666666666e-3 *
        (((a13_sha + (2.0 * a16_sha)) + (2.0 * a19_sha)) + a22_sha)));
    a4_sha =
      (a11_sha +
       (1.6666666666666666e-3 *
        (((a14_sha + (2.0 * a17_sha)) + (2.0 * a20_sha)) + a23_sha)));
  }
  a2_dev[a1] = a2_sha;
  a3_dev[a1] = a3_sha;
  a4_dev[a1] = a4_sha;
}

int
main (int argc, char **argv)
{
  dim3 grids (128);
  dim3 threads (128);
  float *a2 = (float *) malloc (sizeof (float) * 16384);
  float *a2_dev;
  cudaMalloc ((void **) &a2_dev, sizeof (float) * 16384);
  float *a3 = (float *) malloc (sizeof (float) * 16384);
  float *a3_dev;
  cudaMalloc ((void **) &a3_dev, sizeof (float) * 16384);
  float *a4 = (float *) malloc (sizeof (float) * 16384);
  float *a4_dev;
  cudaMalloc ((void **) &a4_dev, sizeof (float) * 16384);
  for (int a1 = 0; a1 < 16384; ++a1) {
    a2[a1] = drand (10.0, 20.0);
    a3[a1] = drand (10.0, 20.0);
    a4[a1] = drand (10.0, 20.0);
  }
  cudaMemcpy (a2_dev, a2, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  cudaMemcpy (a3_dev, a3, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  cudaMemcpy (a4_dev, a4, sizeof (float) * 16384, cudaMemcpyHostToDevice);
  function_on_GPU_a5 <<< grids, threads >>> (a2_dev, a3_dev, a4_dev);
  cudaMemcpy (a2, a2_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  cudaMemcpy (a3, a3_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  cudaMemcpy (a4, a4_dev, sizeof (float) * 16384, cudaMemcpyDeviceToHost);
  for (int a1 = 0; a1 < 16384; ++a1) {
    cout << a2[a1] << " " << a3[a1] << " " << a4[a1] << endl;
  }
  return 0;
}
-:--  main.cu       All (119,0)     (C++/l Abbrev)---------------------------
```

Gnuplot

'lorenz-rk4-0.05.txt'
'lorenz-int1-0.01.txt'

50 45 40 35 30 25 20 15 10 5 0

20 15 10 5 0 -5 -10

-30 -20 -10 0 10 20 30

view: 54.0000, 195.000 scale: 1.31299, 1.13867

アルゴリズムを記述

Paraiso Code

Paraiso

数値計算コードを生成

C++ code

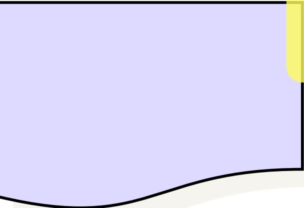
C++ とMPI

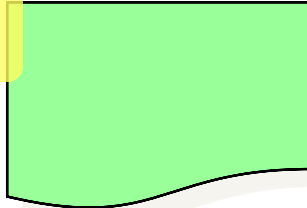
Fortran

CUDA

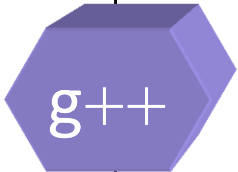
Future Architectures

各アーキテクチャ向けのコード

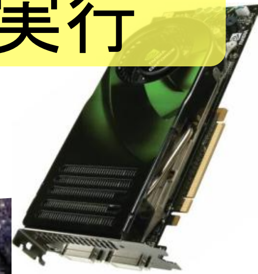




各アーキテクチャのコンパイラ



各マシンで実行・ハイブリッドマシンで実行



Hakubi
Kyoto Univ.

先行研究

Vol. 26 No. 1 情報処理学会論文誌 Jan. 1985

数値シミュレーション用プログラミング言語 DEQSOL†

梅谷征雄†† 辻 みちる†† 岩沢京子††

DEQSOL (Differential EQuation SOlver Language) は数値シミュレーション用に設計された問題向きプログラミング言語であり，偏微分方程式の解法を数値アルゴリズムのレベルで記述することを目的とする．言語設計の狙いは二つあり，第1はこの分野におけるプログラム生産性を飛躍的に向上させることであり，第2は数値シミュレーションに通常用いられるベクトル計算機や並列計算機に向けたプログラムに対しシステム側の最適化の余地を保持することである．このために言語の設計にあわせて，DEQSOL で書かれたプログラムをベクトル/並列計算機向き FORTRAN コードに自動変換するトランスレータを検討・開発した．言語の設計にあたっては，領域形状，メッシュ分割，微分演算子，ベクトル，境界条件など偏微分方程式系の数値シミュレーションに固有の概念を導入して，差分法に基づく計算手順を簡潔・柔軟に記述できるように工夫した結果，典型的な3次元流体シミュレーションプログラムを FORTRAN の10分の1以下の行数で作成する実績を得た．また差分法や線形解法に内在する並列性を生かした FORTRAN コードの生成に努めた結果，M 200 H IAP にて90% 以下のベクトル化を達成することができた．DEQSOL は解法記述言語である点で PDEL[4] や ELLPACK[5] などのソルバとは異なり，むしろ PDELAN[7] のアプローチを拡張・発展させたものである．

かなり似ている・・・

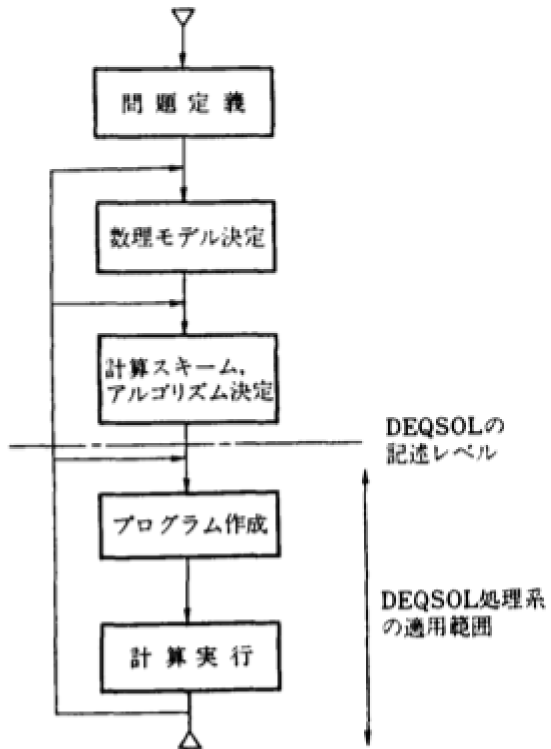


図 1 数値シミュレーション過程と DEQSOL の役割
Fig. 1 Numerical simulation process and the role of DEQSOL system.

基礎方程式

さしあたり人手

離散化形

自動

VVM上のコード

自動

実マシン上のコード

既存コンパイラ

実マシン上の実行ファイル

Hakubi

Hi Takayuki,

DEQSOL was originally developped on Hitachi vector machines (S810/820). It seems Hitachi continued to provide it as a commercial product, at least untill around 2000, but only on shared-memory machines. See:

http://www.cc.u-tokyo.ac.jp/publication/news/VOL4/No2/tetsuzuki-gaiyou.pdf
http://www.hucc.hokudai.ac.jp/sokuho/2001/08.html

So it seems they could not port the DEQSOL program to distributed memory machines (or nobody was using it anyway and there was no demand?)

大丈夫かな・・・・

- ユーザーは少なくとも一人いるし・・・
- 1985年にはHaskellはなかったし・・・

Hakubi Kyoto Univ.

Wikipediaより

- 現在LLという概念が重要視される背景には、計算機資源の増大に伴い、プログラマという人的資源の価値が相対的に上昇したこと、また開発するプログラムの対象そのものが複雑化しており、安全で効率の良いプログラミングのために**人間側の利便性に最適化の主眼が移りつつあることなどがある。**

抽象化により、コーディング速度も実行速度も最適化してみせたい！！

ありがとうございました