formura ユーザーマニュアル

構造格子サブワーキンググループ

2016 年 2 月 11 日

概要

formura は、構造格子上の近接相互作用に帰着されるような系のシミュレーションを対象とし、離散化されたアルゴリズムの数学的・簡潔な記述から、フラッグシップ的大規模並列計算機に最適化されたコードを生成するようなドメイン特化言語です。この文書では、formura の使い方を、例を挙げながら説明します。

目次

1 formura のインストールと使用方法 3
- 1.1 バイナリインストール . . . 3
- 1.2 ソースインストール . . . 3
- 1.3 使用方法 . . . 3

2 formura の生成するコード 4
- 2.1 コード実行の流れ . . . 4
- 2.2 C/C++ 言語からの呼び出し規約 . . . 4
- 2.3 Fortran からの呼び出し規約 . . . 7

3 formura コード例 8
- 3.1 コード例への導入 . . . 8
- 3.2 3 次元拡散方程式 . . . 8
- 3.3 2 次元反応拡散方程式 . . . 13

1 formura のインストールと使用方法

1.1 バイナリインストール

formura の公式サイト https://github.com/nushio3/formura に行き、案内に従って最新の release をダウンロードしてください。

1.2 ソースインストール

formura はプログラミング言語 Haskell で書かれています。そこで、Haskell プログラムの開発環境を整えるためのプログラム、stack を使います。stack は Windows, Mac OS X, 様々な Linux ディストリビューションに対応しています。

stack の公式サイト http://docs.haskellstack.org/en/stable/README.html に行き、指示に従って stack をインストールしてください。

次に、formura のレポジトリからソースコードをダウンロードし、レポジトリのルートディレクトリ内で stack install コマンドを実行してください。

```
$ git clone git@github.com:nushio3/formura.git
$ cd formura
$ stack install
```

formura の実行ファイル群が stack のインストール先（デフォルトではホームディレクトリの.local/bin 以下）にインストールされます。

1.3 使用方法

コマンドラインプログラム formura に formura ソースコードを渡すと、C++ 言語のコードとヘッダが生成されます。生成するコードのファイル名は-o オプションで指定できます。

```
$ formura example/diffusion.fmr -o output/diffusion.c
```

上記のコマンドにより、output/diffusion.h および output/diffusion.c が生成されます。生成されるコードの内容については §2 で詳しく解説します。

2 formura の生成するコード

2.1 コード実行の流れ

C 言語のプログラムが ‘main’ 関数から開始するように、formura のプログラムでは、‘init’ と ‘step’ という名前の関数が、コード生成の起点として特別な意味を持ちます。

formura のプログラムは、最初に ‘init’ 関数を一つ呼び出してシミュレーションの初期状態を作り、その後 ‘step’ 関数を繰り返し呼び出して状態を更新していきます。C 言語の擬似コードで書けば、次のようになります。

```
simulation_state = init();
fun (t = 0; t < T_MAX; ++t) {
  simulation_state = step(simulation_state);
}
```

‘init’ 関数は 0 個の引数をとり、グリッド型の値、またはいくつかのグリッドを含むタプルを返すような関数でなくてはなりません。‘step’ 関数は ‘init’ 関数が返す型を受け取り、同じ型を返す関数である必要があります。

formura は、コード生成にあたって、シミュレーションの独立変数を保持する配列変数を名付けるとき、‘step’ 関数の引数に使われた名前を採用します。なぜなら、formura プログラムの中でシミュレーションの全状態がひとかたまりになっている箇所は三つ（‘init’ 関数の返り値、‘step’ 関数の引数 ‘step’ 関数の返り値）ありますが、このうち「‘step’ 関数の引数」だけが左辺式であって、各変数に独立な名前がついていることが保証されているからです。

2.2 C/C++ 言語からの呼び出し規約

formura の生成するコードの命名規約は、MPI ライブラリと違和感なく使えるように定めることにします。formura は C++ のヘッダとソースコードを組で生成します。ユーザーは通常の方法でコンパイル・自身のプログラムとリンクして実行ファイルを作ってください。

formura が生成したコードを利用するユーザー側のプログラムは次のようになります。

```
#include "mpi.h"
#include "output.h"

int main (int argc, char **argv) {
  Formura_Navigator navi;

  MPI_Init(argc, argv);
  Formura_Init(&navi, MPI_WORLD_COMM);

  while(navi.time_step < T_MAX) {
    Formura_Forward(&navi);
  }

  MPI_Finalize(argc, argv);
}
```

■生成されるコードの内容

formura　の生成するヘッダには、次のものが含まれます。

- シミュレーションの独立変数に対応する配列変数（以下、状態配列 state array と呼びます）
- Formura_Navigator 構造体
- Formura_Init 関数
- Formura_Forward、関数

状態配列は、formura が生成したコードからも、ユーザーからも読み書きされることを想定しています。formura の生成した関数を呼び出していないとき、ユーザーは状態配列を自由に更新することができます。

Formura_Navigator 構造体は、formura の関数からユーザーに対し、いま状態配列に、どの時点・どの範囲の物理量が入っているのか通知するために使います。

たとえば、dimension :: 2;　axes :: x, y が指定されている場合、Formura_Navigator 構造体は次のようになります。

```
struct Formura_Navigator {
  int time_step;
  int offset_x;
  int lower_x;
  int upper_x;
  int offset_y;
  int lower_y;
  int upper_y;
};
```

これは、状態配列の添字 lower_x $\leq x <$ upper_x, lower_y $\leq y <$ upper_y の範囲に、シミュレーションの $t =$ time_step, lower_x + offset_x $\leq x <$ upper_x + offset_x, lower_y + offset_y $\leq y <$ upper_y + offset_y の範囲の値が格納されていることを意味しています。

formura は次のことを保証します：monitor_interval タイムステップごとの、計算領域内の物理量は、少なくとも 1 つの MPI ノードから読むことができる。

■Formura_Init 関数

```
int Formura_Init (Formura_Navigator *navi, MPI_Comm comm)
```

Formura_Init 関数は、formura ソースコードの 'init' 関数に対応する計算を行い、結果を状態配列に書き込みます。

Formura_Init 関数が返すナビゲータは、必ず time_step = 0 を満たします。

Formura_Init 関数は MPI_Comm 型の引数をひとつ取ります。formura が行う通信は、ここの Formura_Init で指定された MPI コミュニケータを使います。

■Formura_Forward 関数

```
int Formura_Forward (Formura_Navigator *navi)
```

Formura_Forward 関数は、状態配列を読み取り、formura ソースコードの 'step' 関数に対応する計算を monitor_interval 回繰り返し行い、結果を状態配列に書き込みます。

Formura_Forward 関数が返すナビゲータの time_step は、直前の値より monitor_interval だけ増加しています。

■formura の生成するコードの並列度や解像度

formura の生成するコードの並列度や解像度は、formura の特殊宣言での指定から決まります。以下に、formura の特殊宣言の例を示します。

```
dimension :: 3
```

```
axes :: x, y, z
intra_node_shape :: 64, 64, 64
mpi_grid_shape :: 40, 20, 5
temporal_blocking_interval :: 3
monitor_interval :: 72
```

各ノードが担当する状態配列の大きさは、およそ 'intra_node_shape' 程度の大きさになります。但し、temporal blocking や通信の都合上、この大きさは多少変更されることがあります。これらのノードを、'mpi_grid_shape' で指定された数だけ x, y, z 方向に並べた領域が計算領域となります。したがって、上記の例では計算の解像度は $2560 \times 1280 \times 320$ となります。さらに、'step' 関数の計算を temporal blocking で 3 ステップ一気に計算するコードを生成すること、Formura_Forward は 72 ステップ (temporal blocking されたコードを 24 回呼び出すことに相当) の計算を一気に行うことが指定されています。

この仕様にともない、Formura_Init 関数に渡す MPI コミュニケータ内にある MPI ランクの数は、'mpi_grid_shape' の積と一致している必要があります。

2.3 Fortran からの呼び出し規約

formura は将来的に Fortran からも使えるようにする予定です。ここで障害となるのは、Fortran 処理系ごとに配列変数のメモリ上での表現が異なっている場合がある、ということです。

このため、Fortran 版では、monitor_interval ごとに、物理量を Fortran 形式の配列に変換して書き込んでユーザーに見せ、変更があった場合はその配列から再度読み込んでシミュレーションを継続する方式を予定しています。

3 formura コード例

3.1 コード例への導入

この節では、簡単なアプリケーションに対する formura のサンプルコードを例示し、使用方法を解説していきます。

なお、この節のサンプルコードや生成されたコードは以下の URL からも参照することができます。併せてご覧ください。

https://github.com/nushio3/formura/tree/master/examples-generated

3.2 3 次元拡散方程式

formura の構文を用いて、ステンシル計算を書いてみます。

formura は、シンプルなグリッド変数どうしの演算を組み合わせることで、ステンシル計算に必要な以下のような計算を記述することができます。

1. グリッドの次元とサイズをを宣言する
2. 各グリッド点にある変数の値を計算する
3. 差分スキームを記述する（空間的に隣接するセルの値を取ってくる）
4. 時間積分スキームを記述する（時間的に次のセルに値を渡す）

以下に、formura を使って記述した拡散方程式のプログラムを示します。

```
dimension :: 3
axes :: x, y, z

begin function ddx(a) returns b
      b = (a[i+1/2,j,k] - a[i-1/2,j,k])/2
end function

begin function ddy(a) returns b
      b = (a[i,j+1/2,k] - a[i,j-1/2,k])/2
end function

begin function ddz(a) returns b
      b = (a[i,j,k+1/2] - a[i,j,k-1/2])/2
end function

∂ = (ddx,ddy,ddz)

begin function Σ(e) returns sum
```

```
        sum = e(0) + e(1) + e(2)
end function

begin function init() returns dens_init
        float [] :: dens_init = 0
end function

begin function dens_next = step(dens)
   float :: Dx, Dt
   Dx = 4.2
   Dt = 0.1
   dens_next = dens + Dx ** 2/Dt * Σ fun(i) ∂ i (∂ i dens)
end function
```

短い関数定義が並ぶと、やや冗長に感じられるでしょうか？例えば、次の関数定義

```
begin function ddx(a) returns b
        b = (a[i+1/2,j,k] - a[i-1/2,j,k])/2
end function
```

は、`returns` の後ろには式が書けることを利用して、二行で書いてしまうこともできます。

```
begin function ddx(a) returns (a[i+1/2,j,k] - a[i-1/2,j,k])/2
end function
```

さらに、`fun (a) expr` の形式を利用して次のように書くこともできます。ここで、`fun (a) expr` は、`a` を引数にとって、`expr` を返す関数、という意味です。プログラミング言語の用語でいえばラムダ式です。

```
ddx(a) = fun (a) (a[i+1/2,j,k] - a[i-1/2,j,k])/2
```

さきほどの拡散方程式のコードを、ラムダ式を活用して書き直すと、次のようになります。

```
   dimension :: 3
   axes :: x, y, z

   ddx = fun(a) (a[i+1/2,j,k] - a[i-1/2,j,k])/2
   ddy = fun(a) (a[i,j+1/2,k] - a[i,j-1/2,k])/2
   ddz = fun(a) (a[i,j,k+1/2] - a[i,j,k-1/2])/2

   ∂ = (ddx,ddy,ddz)

   Σ = fun (e) e(0) + e(1) + e(2)

   begin function init() returns dens_init
```

```
        float [] :: dens_init = 0
  end function

  begin function dens_next = step(dens)
    float :: Dx, Dt
    Dx = 4.2
    Dt = 0.1
    dens_next = dens + Dt / Dx**2 * Σ fun(i) (∂ i . ∂ i) dens
  end function
```

■生成されたcプログラム diffusion.h

```
#pragma once
#ifdef __cplusplus
extern "C" {
#endif
#include <mpi.h>



  extern float dens[64][64][64];

  struct Formura_Navigator {
    int time_step;
    int lower_x;
    int upper_x;
    int offset_x;
    int lower_y;
    int upper_y;
    int offset_y;
    int lower_z;
    int upper_z;
    int offset_z;
  };



  int Formura_Init(struct Formura_Navigator *navi, MPI_Comm comm);

  int Formura_Forward(struct Formura_Navigator *navi);
#ifdef __cplusplus
}
```

```
#endif
```

diffusion.c

```
#include <mpi.h>
#include <math.h>
#include "diffusion.h"



float dens[64][64][64];
float dens_init;
;
float a[64][64][64];
float Dx;
float Dt;
float dens_next_0[64][64][64];
;




int Formura_Init(struct Formura_Navigator *navi, MPI_Comm comm)
{
  int ix, iy, iz;
  const int NX = 64, NY = 64, NZ = 64;
  dens_init = 0.0;
  for (ix = 0; ix < NX + 0; ++ix) {
    for (iy = 0; iy < NY + 0; ++iy) {
      for (iz = 1; iz < NZ + 1; ++iz) {
        dens[ix][iy][iz] = dens_init;
      }
    }
  }

  navi->time_step = 0;
  navi->lower_x = 0;
  navi->offset_x = 0;
  navi->upper_x = 64;
  navi->lower_y = 0;
  navi->offset_y = 0;
  navi->upper_y = 64;
  navi->lower_z = 0;
```

```
  navi->offset_z = 0;
  navi->upper_z = 64;
}


int Formura_Forward(struct Formura_Navigator *navi)
{
  int ix, iy, iz;
  const int NX = 64, NY = 64, NZ = 64;
  int timestep;
  for (timestep = 0; timestep < 20; ++timestep) {
    for (ix = 0; ix < NX + 0; ++ix) {
      for (iy = 0; iy < NY + 0; ++iy) {
        for (iz = 1; iz < NZ + 1; ++iz) {
          a[ix][iy][iz] = dens[ix][iy][iz];
        }
      }
    }

    Dx = 4.2;
    Dt = 0.1;
    for (ix = 1; ix < NX + -1; ++ix) {
      for (iy = 1; iy < NY + -1; ++iy) {
        for (iz = 1; iz < NZ + -1; ++iz) {
          dens_next_0[ix][iy][iz] =
              (a[ix][iy][iz] +
               ((Dt / pow(Dx, 2.0)) *
                ((((((a[ix - 1][iy][iz] - a[ix][iy][iz]) / 2.0) -
                    ((a[ix][iy][iz] - a[ix + 1][iy][iz]) / 2.0)) / 2.0) +
                  ((((a[ix][iy - 1][iz] - a[ix][iy][iz]) / 2.0) -
                    ((a[ix][iy][iz] - a[ix][iy + 1][iz]) / 2.0)) / 2.0)) +
                 ((((a[ix][iy][iz - 1] - a[ix][iy][iz]) / 2.0) -
                   ((a[ix][iy][iz] - a[ix][iy][iz + 1]) / 2.0)) / 2.0))));
        }
      }
    }

    for (ix = 1; ix < NX + -1; ++ix) {
      for (iy = 1; iy < NY + -1; ++iy) {
```

```
        for (iz = 1; iz < NZ + -1; ++iz) {
          dens[ix][iy][iz] = dens_next_0[ix][iy][iz];
        }
      }
    }

  }
  navi->time_step += 20;
}
```

3.3 2 次元反応拡散方程式

もう少し複雑な例として、Pearson [1993] による反応拡散方程式を取り上げます。Pearson [1993] によれば、次のような 2 変数偏微分方程式

$$\frac{\partial U}{\partial t} = D_u \nabla^2 U - UV^2 + F(1-U) \tag{1}$$
$$\frac{\partial V}{\partial t} = D_v \nabla^2 V + UV^2 - (F+k)V \tag{2}$$

は、パラメータによっては時間依存する複雑な挙動を示します。今回は次のパラメータを採用します。

$$k = 0.05 \tag{3}$$
$$F = 0.015 \tag{4}$$
$$D_u = 2 \times 10^{-5} \tag{5}$$
$$D_v = 10^{-5} \tag{6}$$

これを実装した `formura` プログラムは次のようになります。

```
dimension :: 2
axes :: x, y
intra_node_shape :: 256, 256
mpi_grid_shape :: 1, 1
temporal_blocking_interval :: 4
monitor_interval :: 20

ddx = fun(a) (a[i+1/2,j] - a[i-1/2,j])
ddy = fun(a) (a[i,j+1/2] - a[i,j-1/2])

∂ = (ddx,ddy)

Σ = fun (e) e(0) + e(1)
```

```
begin function init() returns (U,V)
  float [] :: U = 0, V = 0
end function

begin function step(U,V) returns (U_next, V_next)
  float :: k = 0.05, F = 0.015, Du = 2e-5, Dv = 1e-5
  float :: dt = 1.0, dx = 1.0

  dU_dt = -U * V**2 + F * (1-U) + Du/dx**2 * Σ fun(i) (∂ i . ∂ i) U
  dV_dt =  U * V**2 - (F+k) * V + Dv/dx**2 * Σ fun(i) (∂ i . ∂ i) V

  U_next = U + dt * dU_dt
  V_next = V + dt * dV_dt
end function
```

上記の formura ソースコードから生成される C コードは次のようになります。

■生成された c プログラム pearson.h

```
#pragma once
#ifdef __cplusplus
extern "C"
{
#endif
#include <mpi.h>



  extern float U[256][256];
  extern float V[256][256];

  struct Formura_Navigator
  {
    int time_step;
    int lower_x;
    int upper_x;
    int offset_x;
    int lower_y;
    int upper_y;
    int offset_y;
  };
```

```
  int Formura_Init (struct Formura_Navigator *navi, MPI_Comm comm);

  int Formura_Forward (struct Formura_Navigator *navi);
#ifdef __cplusplus
}
#endif
```

pearson.c

```
#include <mpi.h>
#include <math.h>
#include "pearson.h"



float U[256][256];
float V[256][256];
float U_0;
float V_0;
;
;
float a[256][256];
float a_0[256][256];
float k;
float F;
float Du;
float Dv;
float dt;
float dx;
float dU_dt[256][256];
float dV_dt[256][256];
float U_next_0[256][256];
float V_next_0[256][256];
;
;




int
Formura_Init (struct Formura_Navigator *navi, MPI_Comm comm)
```

```
{
  int ix, iy;
  const int NX = 256, NY = 256;
  U_0 = 0.0;
  V_0 = 0.0;
  for (ix = 0; ix < NX + 0; ++ix) {
    for (iy = 0; iy < NY + 0; ++iy) {
      U[ix][iy] = U_0;
    }
  }

  for (ix = 0; ix < NX + 0; ++ix) {
    for (iy = 0; iy < NY + 0; ++iy) {
      V[ix][iy] = V_0;
    }
  }

  navi->time_step = 0;
  navi->lower_x = 0;
  navi->offset_x = 0;
  navi->upper_x = 256;
  navi->lower_y = 0;
  navi->offset_y = 0;
  navi->upper_y = 256;
}



int
Formura_Forward (struct Formura_Navigator *navi)
{
  int ix, iy;
  const int NX = 256, NY = 256;
  int timestep;
  for (timestep = 0; timestep < 20; ++timestep) {
    for (ix = 0; ix < NX + 0; ++ix) {
      for (iy = 0; iy < NY + 0; ++iy) {
        a[ix][iy] = U[ix][iy];
      }
    }
```

```
for (ix = 0; ix < NX + 0; ++ix) {
  for (iy = 0; iy < NY + 0; ++iy) {
    a_0[ix][iy] = V[ix][iy];
  }
}

k = 5.0e-2;
F = 1.5e-2;
Du = 2.0e-5;
Dv = 1.0e-5;
dt = 1.0;
dx = 1.0e-2;
for (ix = 1; ix < NX + -1; ++ix) {
  for (iy = 1; iy < NY + -1; ++iy) {
    dU_dt[ix][iy] =
      (((-(a[ix][iy] * pow (a_0[ix][iy],
              2.0))) + (F * (1.0 - a[ix][iy]))) + ((Du / pow (dx,
            2.0)) * (((a[ix - 1][iy] - a[ix][iy]) - (a[ix][iy] - a[ix +
                1][iy])) + ((a[ix][iy - 1] - a[ix][iy]) - (a[ix][iy] -
              a[ix][iy + 1])))));
  }
}

for (ix = 1; ix < NX + -1; ++ix) {
  for (iy = 1; iy < NY + -1; ++iy) {
    dV_dt[ix][iy] =
      (((a[ix][iy] * pow (a_0[ix][iy],
            2.0)) - ((F + k) * a_0[ix][iy])) + ((Dv / pow (dx,
            2.0)) * (((a_0[ix - 1][iy] - a_0[ix][iy]) - (a_0[ix][iy] -
              a_0[ix + 1][iy])) + ((a_0[ix][iy - 1] - a_0[ix][iy]) -
            (a_0[ix][iy] - a_0[ix][iy + 1])))));
  }
}

for (ix = 1; ix < NX + -1; ++ix) {
  for (iy = 1; iy < NY + -1; ++iy) {
    U_next_0[ix][iy] = (a[ix][iy] + (dt * dU_dt[ix][iy]));
  }
```

```
    }

    for (ix = 1; ix < NX + -1; ++ix) {
      for (iy = 1; iy < NY + -1; ++iy) {
        V_next_0[ix][iy] = (a_0[ix][iy] + (dt * dV_dt[ix][iy]));
      }
    }

    for (ix = 1; ix < NX + -1; ++ix) {
      for (iy = 1; iy < NY + -1; ++iy) {
        U[ix][iy] = U_next_0[ix][iy];
      }
    }

    for (ix = 1; ix < NX + -1; ++ix) {
      for (iy = 1; iy < NY + -1; ++iy) {
        V[ix][iy] = V_next_0[ix][iy];
      }
    }

  }
  navi->time_step += 20;
}
```

■メインプログラム　上記プログラムを呼び出すメインプログラムのほうは次のようになります。

pearson-main.cpp

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pearson.h"

const int T_MAX = 20000;

Formura_Navigator navi;

float frand() {
  return rand() / float(RAND_MAX);
}
```

```
void init() {
  for(int y = navi.lower_y; y < navi.upper_y; ++y) {
    for(int x = navi.lower_x; x < navi.upper_x; ++x) {
      U[y][x] = 1;
      V[y][x] = 0;
    }
  }
  for (int y = 118; y < 138; ++y) {
    for (int x = 118; x < 138; ++x) {
      U[y][x] = 0.5+0.01*frand();
      V[y][x] = 0.25+0.01*frand();
    }
  }
}

int main (int argc, char **argv) {
  system("mkdir -p frames");
  srand(time(NULL));
  MPI_Init(&argc, &argv);
  Formura_Init(&navi, MPI_COMM_WORLD);

  init();

  while(navi.time_step < T_MAX) {
    if(navi.time_step % 100 == 0) {
      printf("t = %d\n", navi.time_step);
      char fn[256];
      sprintf(fn, "frames/%06d.txt", navi.time_step);
      FILE *fp = fopen(fn,"w");
      for(int y = navi.lower_y; y < navi.upper_y; ++y) {
        for(int x = navi.lower_x; x < navi.upper_x; ++x) {
          fprintf(fp, "%d %d %f\n", x, y, U[y][x]);
        }
        fprintf(fp, "\n");
      }
      fclose(fp);
    }
    Formura_Forward(&navi);
```

```
  }
  MPI_Finalize();
}
```

Acknowledgements

タプル型の設計にあたっては Pierce [2002] を参考にしました。また、タプルの言語への組み込みにあたっては Oliveira et al. [2015] を参考にしました。

参考文献

B. C. d. S. Oliveira, S.-C. Mu, and S.-H. You. Modular reifiable matching: a list-of-functors approach to two-level types. In Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, pages 82–93. ACM, 2015.

J. E. Pearson. Complex patterns in a simple system. Science, 261(5118):189–192, 1993.

B. C. Pierce. Types and programming languages. MIT press, 2002.