粒子法の大規模並列化フレームワークと SPH法に関するいくつかの話題

牧野淳一郎
理化学研究所 計算科学研究機構
エクサスケールコンピューティング開発プロジェクト
コデザイン推進チーム
兼 粒子系シミュレータ開発チーム

2014/08/06 先駆的科学計算に関するフォーラム2014

話の構成

- 「粒子法の大規模並列化フレームワーク」の話
  1. どんなことをしたい (あるいはしたくない) か
  2. ではどうするか
  3. 現在の実装方針
  4. 現状と将来
- SPH の話題いくつか
  1. 陰解法と陽解法
     - 原理
     - 実例 1: 音速抑制法
     - 実例 2: 慣性変化法
     - その他の可能性
  2. 「疑似密度法」

「粒子法の大規模並列化フレームワーク」の話

1. どんなことをしたい (あるいはしたくない) か
2. ではどうするか
3. 現在の実装方針
4. 現状と将来

どんなことをしたいか

というよりむしろ、何がしたくないか

- MPI でプログラムなんか書きたくない
- キャッシュ再利用のためのわけのわからないループ分割とかしたくない
- 通信量減らすためにわけのわからない最適化とかするのも勘弁して欲しい
- SIMD 命令がでるようにコードをいじりまわすとかやめたい
- 機械毎にどういう最適化すればいいか全然違うとか、それ以前に言語から違うとかはもういやだ

そうはいっても――ではどうするか？

昔からある考え方はこんな感じ？

- 並列化コンパイラになんとかしてもらう
- 共有メモリハードウェアになんとかしてもらう
- 並列言語とコンパイラの組み合わせになんとかしてもらう

しかし……

- 若い人はそういう考え方があったことも既に知らないような気がする。「スパコンとはそういうものだ」みたいな。
- つまり、こういうアプローチはほぼ死滅した。
- 理由は簡単: 性能がでない。安価なハードウェアで高い性能がでるものがプログラミングが大変でも長期的には生き残る。

じゃあ本当のところどうするか？

1. 人生そういうものだと諦めて MPI でプログラム書いて最適化もする
   難点: 普通の人の場合性能がでない。難しいことをしようとすると無限に時間がかかって人生が終わってしまう。
2. 他人 (学生、ポスドク、外注、ベンダ等) に MPI でプログラム書かせて最適化もさせる
   難点: 他人が普通の人の場合、やはり性能でない。無限に人と時間とお金がかかる。あとでいじるにも無限に人と時間とお金がかかる。

- どちらも今一つというか今百くらいである。
- もちろん、「普通でない人」を確保できればなんとかなるがこれは希少資源である。

原理的には、「普通でない人」を有効利用すればいい？

どうやって有効利用するか？

色々な考え方がありえるが、我々の (というか私の) 考え方:

- 「普通でない人」がやった方法を一般化して、色々な問題に適用する。
- 例えば「粒子系一般」という程度
- DRY (Don't Repeat Yourself) の原則の徹底

どうやって有効利用するか？

色々な考え方がありえるが、我々の (というか私の) 考え方:

- 「普通でない人」がやった方法を一般化して、色々な問題に適用する。
- 例えば「粒子系一般」という程度
- DRY (Don't Repeat Yourself) の原則の徹底
- 「神戸人外王国」が詳細仕様策定及び実装

(神戸花鳥園は 2014 年 7 月から神戸どうぶつ王国に生まれ変わりました)

もうちょっと具体的には？

色々な粒子系計算

- 重力多体系
- 分子動力学
- 粒子法による流体 (SPH、MPS、MLS、その他)
- 構造解析等のメッシュフリー法

計算のほとんどは近傍粒子との相互作用 (遠距離力: Tree, FMM, PME その他)

なので、粒子分布を与えると

- 領域分割 (ロードバランスも考慮した)
- 粒子の移動
- 相互作用の計算 (そのために必要な通信も)

を高い効率 (実行効率・並列化効率) でやってくれるプログラムを「自動生成」できればいい

現在の実装方針

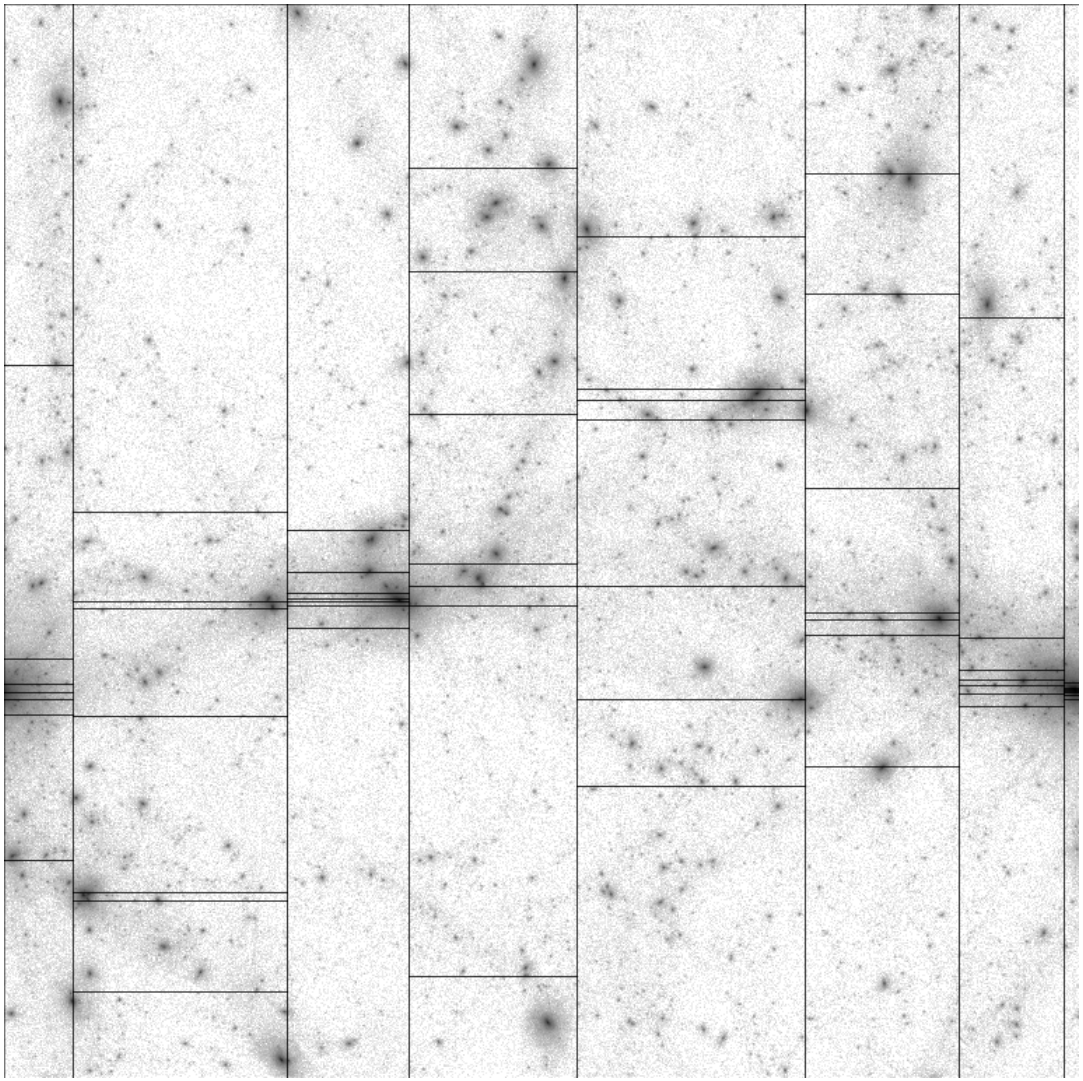
- API は C++ で定義
- ユーザーは
  - 粒子データクラス
  - 粒子間相互作用を計算する関数 (現在のところ、ユーザーが機種毎に最適化。この自動生成は並行して開発中)

  を用意。さらにドライバープログラム (I/O ライブラリコール等も含む)、時間積分関数とかも書く
- 全体をコンパイルすると空間分割して粒子再配置して時間積分して、、、というプログラムになる

開発の現状

- 仕様を文書化した。
- 非並列版の実装を作った。重力多体計算、SPH 計算は一応できているようである。
- MPI 並列版作成中。
- 年度内リリース予定。

空間分割と並列化



空間分割して計算ノードに割り当て

Recursive Multi-section (JM 2004)

「京」の Tofu ネットワークに適した方法

計算時間が均等になるよう領域サイズ調整 (石山他 2009、2012)

重力多体計算のサンプルユーザーコード

1. ヘッダと計算結果クラス

```
#include<particle_simulator.h> // 必須

class ResultForce{ //名前自由。
public:
    void clear(){ //絶対必要。名前固定。
        acc = 0.0;
        pot = 0.0;
    }
    //以下名前含めユーザー定義。
    PS::F64vec acc;
    PS::F64 pot;
};
```

粒子クラス(1)

```
class RealPtcl{ //絶対必要。名前自由。
public:
    PS::F64vec getPos() const { //絶対必要。名前固定。
        return this->pos;
    }
    PS::F64 getCharge() const { //絶対必要。名前固定。
        return this->mass;
    }
    void copyFromForce(const ResultForce & force){
    //絶対必要。名前固定。
        this->acc = force.acc;
        this->pot = force.pot;
    }
```

粒子クラス(2)

```
//以下名前含めユーザー定義。
PS::S64 id;
PS::F64 mass;
PS::F64vec pos;
PS::F64vec vel;
PS::F64vec acc;
PS::F64 pot;
PS::S32 loadOneParticle(FILE * fp) {
    PS::S32 ret = 0;
    ret = fscanf(fp, "%lf%lf%lf%lf%lf%lf%lf",
                 &this->mass,
                 &this->pos[0], &this->pos[1], &this->pos[2
                 &this->vel[0], &this->vel[1], &this->vel[2
    std::cout<<"this->mass"<<this->mass<<std::endl;
    return ret;
}
```

粒子クラス(3)

```
    void dumpOneParticle(FILE * fp){
        fprintf(fp, "%lf   %lf   %lf   %lf   %lf   %lf   %lf
                this->mass,
                this->pos[0], this->pos[1], this->pos[2],
                this->vel[0], this->vel[1], this->vel[2]);
    }
};

class EssentialPtclI{
public:
    void copyFromRP(const RealPtcl & rp){ //絶対必要。
名前固定。
        pos = rp.pos;
        id = rp.id;
    }

    //以下名前含めユーザー定義。
    PS::F64vec pos;
    PS::S64 id;
};
```

粒子クラス(4)

```
class EssentialPtclJ{
public:
    void copyFromRP(const RealPtcl & rp){ //絶対必要。
名前固定。
        mass = rp.mass;
        pos = rp.pos;
        id = rp.id;
    }

    //以下名前含めユーザー定義。
    PS::S64 id;
    PS::F64 mass;
    PS::F64vec pos;
};
```

相互作用関数

```
void CalcForceEpEp(const EssentialPtclI * ep_i,
                   const PS::S32 n_ip,
                   const EssentialPtclJ * ep_j,
                   const PS::S32 n_jp,
                   ResultForce * force){
    for(PS::S32 i=0; i<n_ip; i++){
        PS::F64vec xi = ep_i[i].pos;
        PS::F64vec ai = 0.0;
        PS::F64 poti = 0.0;
        PS::S64 idi = ep_i[i].id;
        for(PS::S32 j=0; j<n_jp; j++){
            if( idi == ep_j[j].id ) continue;
            PS::F64vec rij = xi - ep_j[j].pos;
            PS::F64 r3_inv = rij * rij;
            PS::F64 r_inv = 1.0/sqrt(r3_inv);
            r3_inv = r_inv * r_inv;
            r_inv *= ep_j[j].mass;
            r3_inv *= r_inv;
            ai -= r3_inv * rij;
            poti -= r_inv;
```

相互作用関数(続き)

```
            }
            force[i].acc = ai;
            force[i].pot = poti;
        }
}
```

時間積分(ユーザープログラム内)

```
//積分(Leapfrog)
void Kick(PS::ParticleSystem<RealPtcl> system,
          const PS::F64 dt){
    RealPtcl * rp = system.getParticlePointer();
    PS::S32 ni = system.getNumberOfParticleLocal();
    for(int i=0; i<ni; i++){
        rp[i].vel  = rp[i].acc * dt;
    }
}

void Drift(PS::ParticleSystem<RealPtcl> system,
           const PS::F64 dt){
    RealPtcl * rp = system.getParticlePointer();
    PS::S32 ni = system.getNumberOfParticleLocal();
    for(int i=0; i<ni; i++){
        rp[i].pos  = rp[i].vel * dt;
    }
}
```

ユーザープログラム本体

```
int main(int argc, char *argv[]){

    PS::Initialize(argc, argv); //初期化
    PS::ParticleSystem<RealPtcl> nbody_system; //粒子
種の数だけ生成
    nbody_system.loadParticleSingle(argv[1], "r", &RealPtcl::lo

    PS::DomainInfo dinfo;
    dinfo.initialize("domain_info.para");
    dinfo.decomposeDomainAll(nbody_system);
    //重心展開単極子用ツリーを選択。
    PS::TreeType<ResultForce, EssentialPtclI, EssentialPtclJ>::
    PS::F32 theta = 0.5;
    PS::S32 n_leaf_max = 8;
    PS::S32 n_grp_max = 64;
    //デフォルト引数を使って、第一引数だけでも良い。
    grav_tree.initialize(nbody_system.getNumberOfParticleTotal(
```

ユーザープログラム本体

//粒子交換、LT作り、LET交換、GT作り、相互作用計算
をし、結果をnbody_systemに書き戻す。

```
grav_tree.calcForceAllAndWriteBack(CalcForceEpEp, dinfo, nb
```

```
PS::F64 dt = 0.125;
PS::F64 tend = 100.0;
PS::F64 tsys = 0.0;
Kick(nbody_system, dt*0.5);
while(tsys < tend){
    tsys += dt;
    Drift(nbody_system, dt);
    if( fmod(tsys, 1.0) == 0){
        //領域分割毎回やらない。
        dinfo.decomposeDomainAll(nbody_system);
    }
    grav_tree.calcForceAllAndWriteBack(CalcForceEpEp, dinfo
    Kick(nbody_system, dt);
}
```

ユーザープログラム本体

```
    PS::Finalize();
    return 0;
}
```

コメント等

- 一応動く。
- それなりに最適化された並列化ツリーアルゴリズムを 150 行くらいのユーザープログラムで使える。
- まだちょっと記述量が多い。もうちょっとコンパクトな記述から粒子定義クラスとかほぼ必須な関数とかを自動生成したい (牧野の希望)。
- 年度内リリース予定。

陰解法と陽解法

- 原理
- 実例 1: 音速抑制法
- 実例 2: 慣性変化法
- その他の可能性

原理

まず、陰解法とは？何故使うのか？という話。
典型的な例: 「非圧縮」流体
非圧縮性流体の基本方程式:

$$\nabla \cdot v = 0$$
$$\frac{\partial v}{\partial t} + (v \cdot \nabla)v = -\nabla p + \frac{1}{\mathrm{Re}}\nabla^2 v$$

- 形式的には、本当の流体に「音速無限大」の近似をしたもの
- 本当の流体の遅い流れをそのまま解くと、CFL 条件によって音波モードでタイムステップが制限される
- 音速無限大の近似をして音波モードを消す代わりに、圧力に対するポアソン方程式がでてくる
- タイムステップは大きくとれるが、ポアソン方程式を解く必要あり

ポアソン方程式の問題点

- 空間 1 次元なら直接解けて計算量も少ない
- 2、3 次元では反復法。分解能あげたり、適応格子にしたりすると段々収束性悪くなる
- CG 反復にしてもマルチグリッド反復にしても、メモリアクセス負荷、通信負荷 (レイテンシに対する要求) が高く大規模並列化が困難

近似の方向を逆に

- 非圧縮近似: 音速無限大の極限
- 逆の近似: 音速を、解の性質が変わらない範囲で「小さく」する
  - CFL 条件が緩和されてタイムステップを大きくとれる
  - 格子マッハ数が 1 よりある程度小さければ大丈夫: 非圧縮近似と同程度のタイムステップがとれる
- 非圧縮近似に比べて悪いところはない
- (何故非圧縮近似がこれまで何十年も使われてきたのかよくわからない)

これは別に新発明ではなくて

(割合最近だけど) 既に例がある。

- Explicit-MPS (山田他 2011, MPS の S って semi-implicit じゃなかったっけ感満載)
- 音速抑制法 (Fan et al. 2003, Hotta et al. 2012)
  - 恒星内部や惑星大気のような、高さが圧力スケールハイトより大きい場合の方法
  - レファレンスの断熱温度分布からの摂動方程式に書き換える
  - それから連続の式だけいじる。

音速抑制法

$$\frac{\partial \rho_1}{\partial t} = -\frac{1}{\xi^2}\nabla \cdot (\rho_0 \boldsymbol{v}),$$

$$\frac{\partial \boldsymbol{v}}{\partial t} = -(\boldsymbol{v}\cdot\nabla)\boldsymbol{v} - \frac{\nabla p_1}{\rho_0} - \frac{\rho_1}{\rho_0} g\boldsymbol{e}_z + \frac{1}{\rho_0}\nabla\cdot\boldsymbol{\Pi},$$

$$\frac{\partial s_1}{\partial t} = -(\boldsymbol{v}\cdot\nabla)(s_0 + s_1) + \frac{1}{\rho_0 T_0}\nabla\cdot(K\rho_0 T_0 \nabla s_1)$$

$$+\frac{\gamma-1}{p_0}(\boldsymbol{\Pi}\cdot\nabla)\cdot\boldsymbol{v},$$

$$p_1 = p_0\left(\gamma\frac{\rho_1}{\rho_0} + s_1\right),$$

添字0がレファレンス、1が摂動。ξ が音速を変える係数
ξ は場所の関数でもかまわない
「京」での大規模計算が初めて可能に

慣性変化法

目的: マントル対流のような、高粘性流体の計算の陽解法化
過去数十年使われてきたマントル対流の数値計算法:

- **非圧縮近似 (拡張ブシネスク近似)**
- **運動方程式の慣性項を落とす (小さいので) 近似**

の組み合わせ。普通の音速抑制法では粘性で決まる時間ステップが短いまま

もう一度近似の方向を逆に

- 普通の慣性項を落とす近似: 慣性項が小さいからできる
- 従って、慣性項を「大きく」してもいいはず
- どこまで大きくしていいか: ストークス流の性質が維持される範囲、すなわち、レイノルズ数が1よりある程度小さいところ

支配方程式とその変形

元々の方程式

$$\rho \frac{dv}{dt} = -\nabla p - \rho g e_z + \nabla \cdot \Pi$$
$$\rho c_p \frac{dT}{dt} = -\lambda \frac{dp}{dt} = \nabla \cdot (k \nabla T) + (others)$$

変形した式

$$\xi \rho \frac{dv}{dt} = -\nabla p - \rho g e_z + \frac{1}{c} \nabla \cdot \Pi$$
$$\rho c_p \frac{dT}{dt} = -\lambda \frac{dp}{dt} = \nabla \cdot (c k \nabla T) + (others)$$

導入したパラメータの意味

- ξ: 慣性項を変化させる。レイノルズ数、マッハ数が小さい限り解に影響しない。位置や状態量の関数でいい
- c: 熱伝導と粘性をコンシステントに変化させる。時間発展の速度を変えるので、グローバルな量

Q: 何故パラメータが1つではなく2つか？

A: マッハ数、レイノルズ数を同時に1に近付けるため。

Q: 粘性が大きく変化する系では破綻しないか？

A: 上手くパラメータとその粘性依存性を調整すると、粘性の比を Rとして、マッハ数、レイノルズ数共に $R^{-1/3}$ より小さくならないようにできる。

こんな方法が本当に上手くいくのか？

- やってみたら本当に上手くいく (竹山、卒業研究)
- 粘性が変化する場合とかも大丈夫みたい
- マントル対流の超高分解能計算への可能性が開けた

これのどこが SPH の話なんだっけ？

- 今回 SPH で実装した。
- 対流を扱うのに、非圧縮とか拡張ブシネスクとかの近似は一切必要ない。元々の方程式のまま (で、慣性項を変えるのと拡散係数のスケーリングのみ行う)。
- (まだやってみてないけど)3 次元対流とかでもデカルト座標でそのままやれる。

計算例

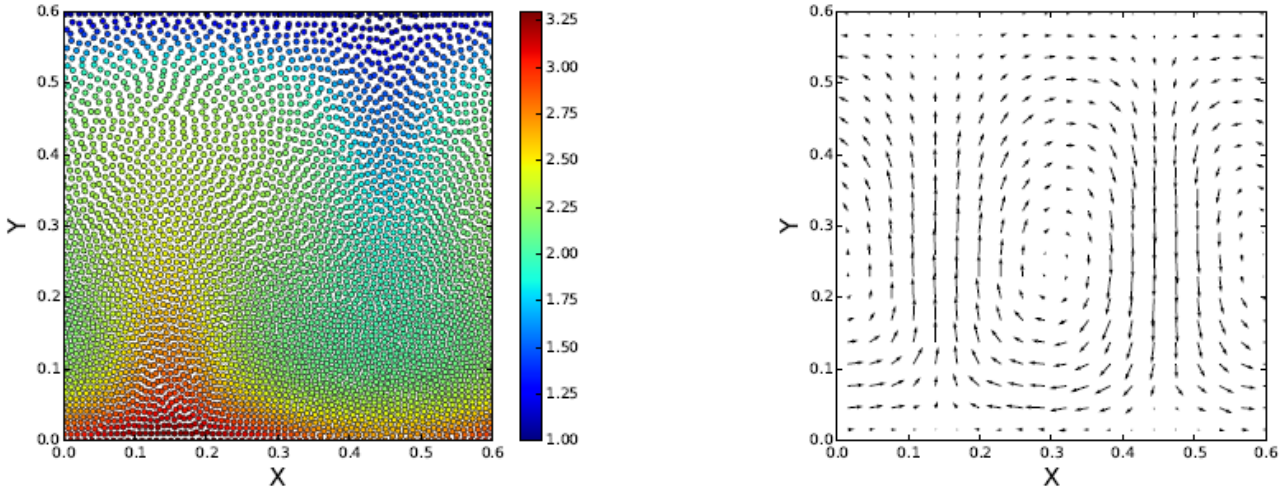
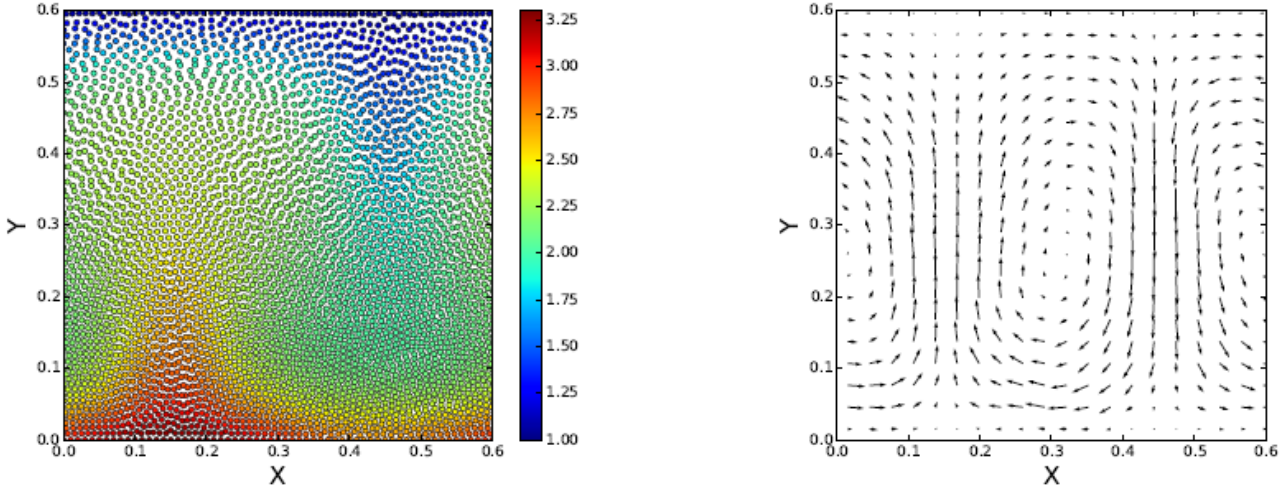
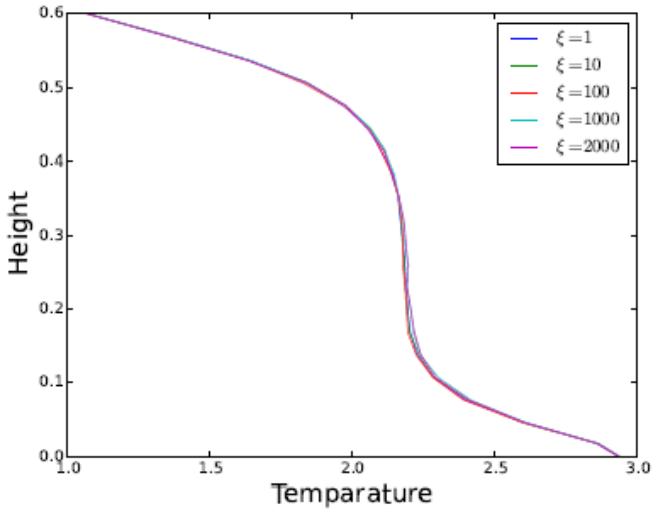


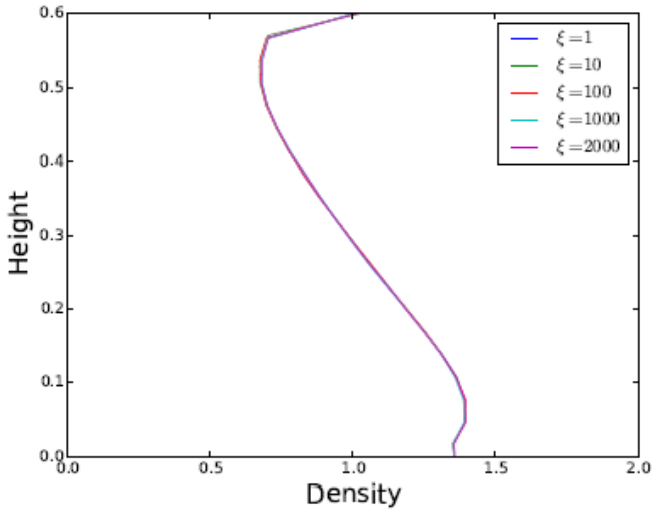
Figure 5: The temperature distribution and the velocity distribution when ξ=1.



下は慣性項を 1000 倍にしたもの。みため同じ。

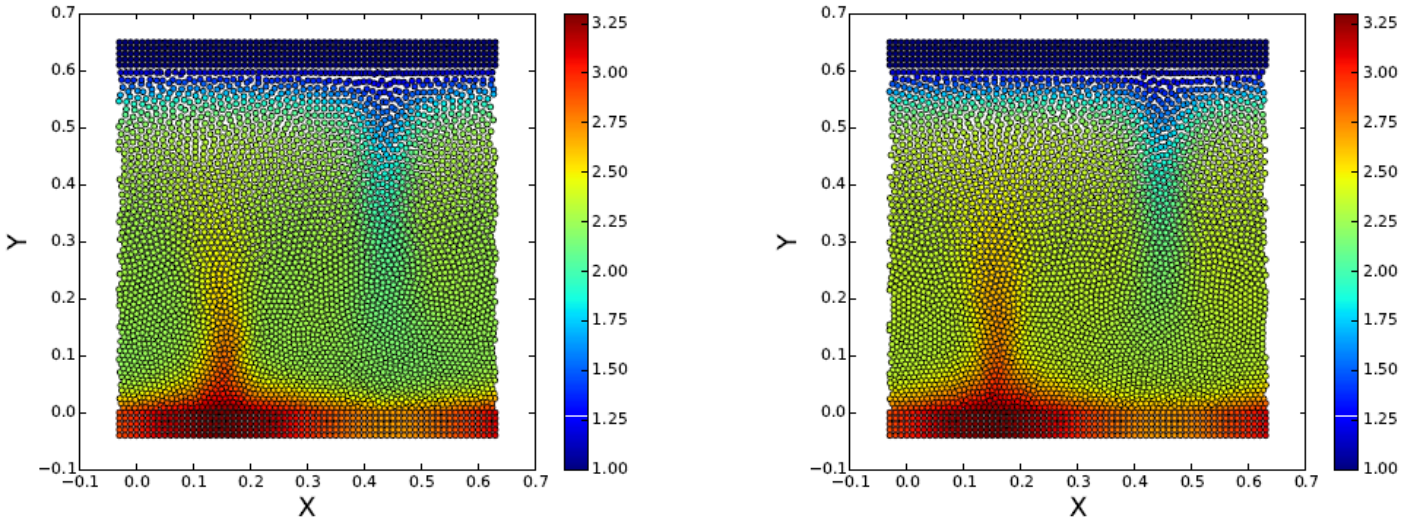
計算例続き





- 温度分布・密度分布ももちろん同じ。他の物理量、熱流等も同じ。
- 計算時間は慣性項を 1000 倍にすると $1/1000$ になる。ほぼダイナミカルな計算になる。レイノルズ数で 1 くらい。

温度依存する粘性



- 上と下で粘性２桁違う
- 慣性項を温度依存で変化させた。特に問題なく計算できた。

他の応用

時間発展のどこかに「速度無限大」をいれて陰解法にしている話なら原理的になんでも応用可能なはず

- 輻射伝達 (光速を遅くする)
- 化学反応を伴う流れ
- 電磁場、重力場 (??)

まとめ(陰解法関係の話だけ)

- 亜音速流体に対しては、音速を遅くした上で圧縮性流体向きの解法を使うのが好ましい
  - 反復が不要、メモリバンド幅要求が下がる、通信レイテンシ要求も下がる
- 単純に状態方程式をいじることができないケースでも適用可能(音速抑制法、慣性変化法)
- 陰解法が必須な場面というのはそれほどないのではないか？

疑似密度法

話の順番

- SPH の基本的な定式化
- 接触不連続
- 対策
- まとめ

SPH の基本式

ある物理量 f の推定

$$\langle f \rangle(\vec{x}) = \int f(\vec{x}')W(\vec{x} - \vec{x}')d\vec{x}'. \tag{1}$$

密度の推定

$$\rho(\vec{x}) = \sum_j m_j W(\vec{x} - \vec{x}_j), \tag{2}$$

SPH 近似

$$\langle f \rangle = \sum_j m_j \frac{f_j}{\rho(\vec{x})} W(\vec{x} - \vec{x}_j). \tag{3}$$

SPH の基本のつづき (1)

f の微分: $\langle \nabla f \rangle = \nabla \langle f \rangle$ で、以下の恒等式

$$1 = \sum_j m_j \frac{1}{\rho(\vec{x})} W(\vec{x} - \vec{x}_j). \tag{4}$$

を使って、さらにもうちょっと近似して

$$\langle \nabla f \rangle(\vec{x}) \sim \sum_j m_j \frac{f(\vec{x}_j)}{\rho(\vec{x}_j)} \nabla W(\vec{x} - \vec{x}_j). \tag{5}$$

SPH の基本のつづき (2)

運動方程式は $-\frac{1}{\rho}\nabla P$ を計算する。この時に恒等式

$$\frac{1}{\rho}\nabla P = \frac{P}{\rho^2}\nabla\rho + \nabla\frac{P}{\rho^2}. \tag{6}$$

を使って対称化すると

$$\dot{v}_i = -\sum_j m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\right) \frac{\partial}{\partial x_i} W(x_i - x_j), \tag{7}$$

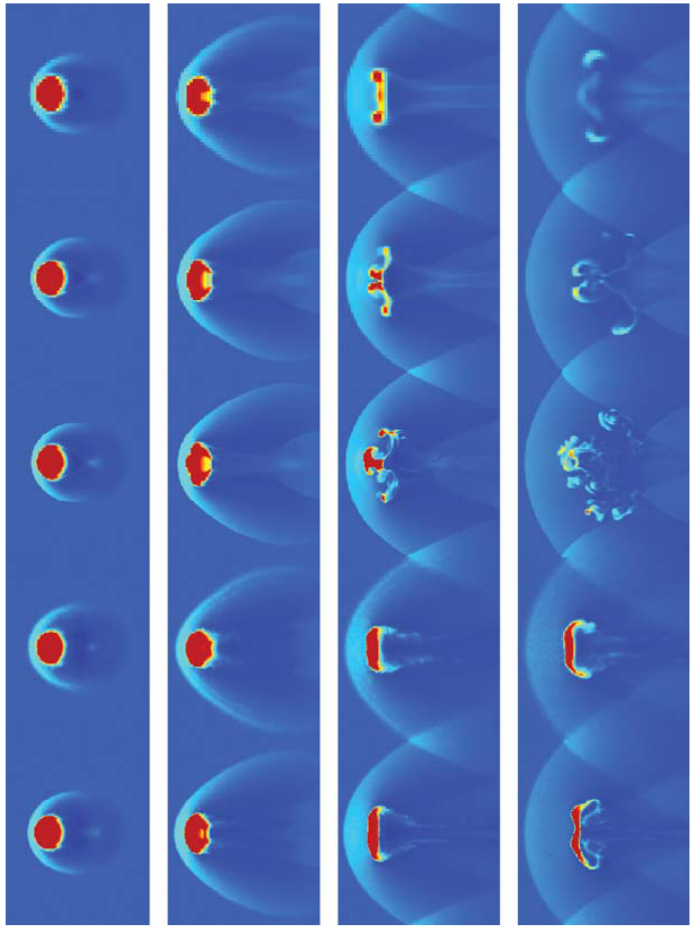
になる。

SPH と接触不連続、KH 不安定

Agertz et al (MN 2007, 380,963)

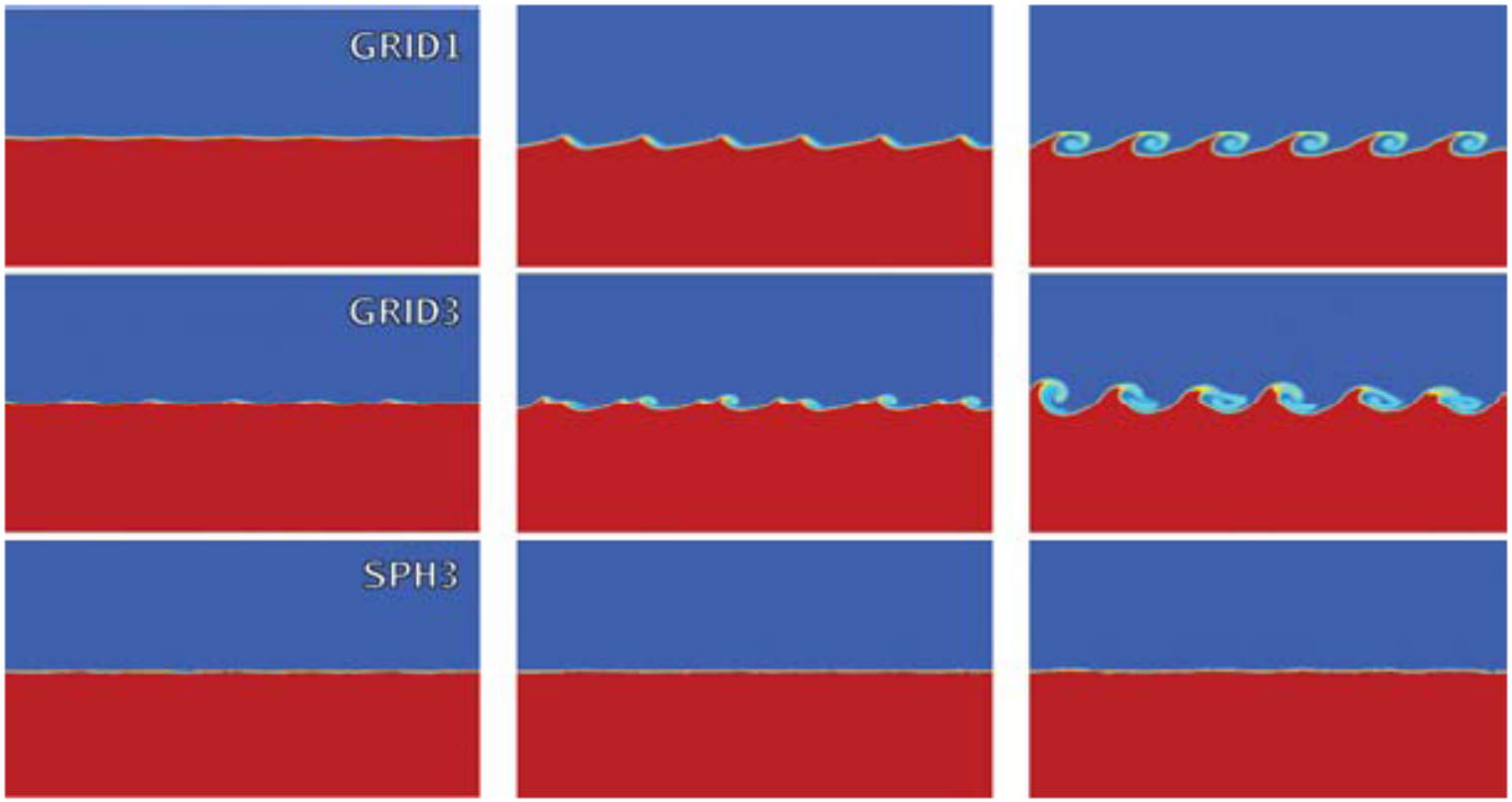
- SPH と Grid コードで、「Brob test」の答が全然違う
- もっと簡単な Kelvin-Helmholtz 不安定 (計算は 3 次元) でも全然違う
- SPH だめじゃん

どれくらい違うか (1)



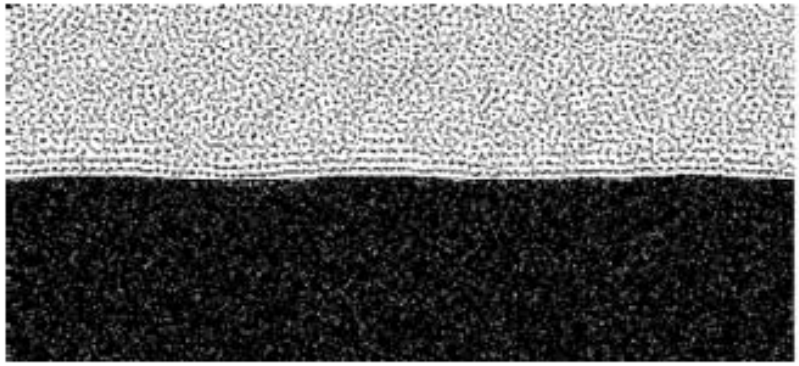
- 周りよりつめたい (温度 1/10、、密度 10 倍) ガスの球を超音速で動かす
- 上から 3 個は Grid
- 下の 2 つは Gasoline (下は 10M 粒子)
- SPH では 境界での不安定が起きないで、冷たい流体が固まりのまま。

どれくらい違うか (2)



SPH では KH 不安定が起きない。

どれくらい違うか (3)



2 流体の境界面で妙な隙間ができる。このため力が働かない？

密度不連続面での振る舞い

通常の SPH では、変形の 2 箇所で ρ の微分可能性を仮定している。以下の 2 つの「恒等式」である

$$1 = \sum_j m_j \frac{1}{\rho(\vec{x})} W(\vec{x} - \vec{x}_j). \tag{8}$$

$$\frac{1}{\rho} \nabla P = \frac{P}{\rho^2} \nabla \rho + \nabla \frac{P}{\rho^2}. \tag{9}$$

SPH でカーネル推定した密度は滑らか。このため、

- 接触不連続の低密度側では、密度を過大推定、高密度側では過小推定する
- その結果、圧力、その空間微分もデタラメになる。結果として粒子が再配置される

対策

「根本的」な理由:

ρ は滑らかだけど u (内部エネルギー) はジャンプがあるまま。

この観点では、u を滑らかにすればよい。色々提案あり。

- **u にもカーネル推定した量を使う**
- **u を拡散させる (人工熱伝導)**
- **質量密度でない密度 (数密度とか) を使う**

それぞれ、それなりにうまくいくケースもある。

新しい提案 — 思想

圧力が本来変わってないのに、密度が不連続なだけでおかしなことが起こるのは何故か？

物理量 (とその微分) の推定式に密度を使うから:

$$\langle f \rangle(\vec{x}) = \sum_j \frac{m_j f(\vec{x}_j)}{\rho(\vec{x}_j)} W(\vec{x} - \vec{x}_j). \tag{10}$$

ここでやっていることは、本質的には体積要素 $d\vec{x}$ を $m_j/\rho(\vec{x}_j)$ で置き換えているだけ。

粒子の占める体積の推定さえできれば別に何を使ってもいいはず

新しい提案 — 実際

これまでいくつかやってみた

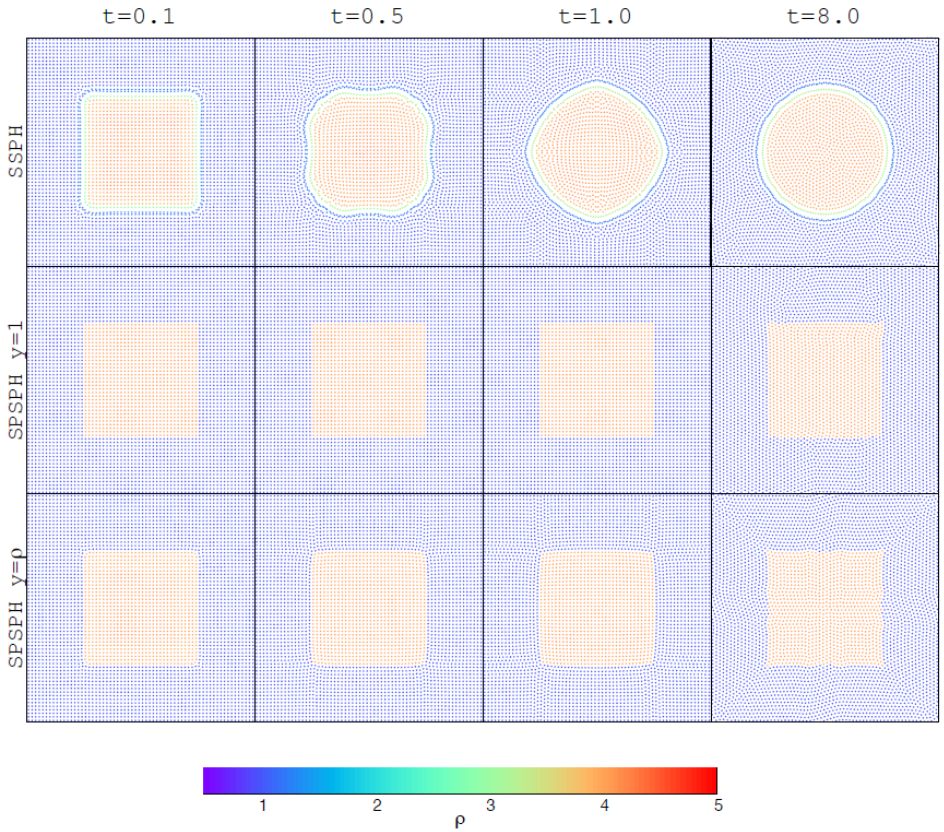
- 密度の代わりに圧力を使う (Saito and Makino 2013, Hosono et al. 2013)
- 圧力の弱いべき乗を使って、圧力変化が大きいところの振舞いを改善する (SM13, Hosono et al. 2014)

これはこれで上手くいくが、圧力のスケールハイトが粒子間隔くらい以下になるとやはり上手くない。ので別の方法を考えた。(Yamamoto et al. submitted)

- 体積要素を計算するためだけに「疑似質量」を導入
- 「疑似密度」は適当な拡散係数で拡散させることで不連続面を消す。これは粒子のエネルギー、圧力を (誤差の範囲でしか) 変えない。

定式化の細かい話は今日は省略。

数値例(1)



静水圧平衡の密度だけ違う流体。なにも起きないのが正しい。

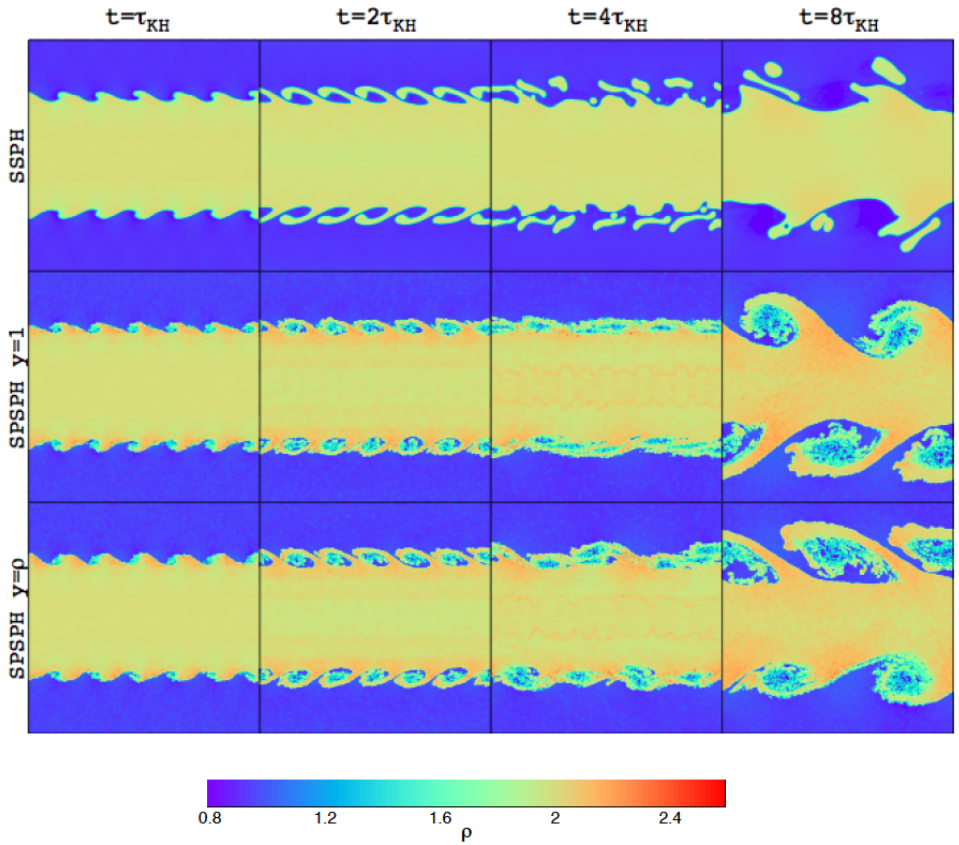
上:従来の。

中央:初期疑似密度$=1$

下:初期疑似密度$=$本当の密度

新しい方向では初期設定によらずちゃんと計算できた。

数値例(2)



KHI。ちゃんと渦まいて欲しい

上:従来の。

中央:初期疑似密度$=1$

下:初期疑似密度$=$本当の密度

新しい方向では初期設定によらずちゃんと計算できた。

まとめ (疑似密度法)

- SPH で、接触不連続でおかしなことが起きないようにする方法を定式化した
- 水と空気のような、密度比が極端に大きい場合でも 0 次では問題ない
- 圧力勾配の不連続は表現できないので、運動方程式に高次の誤差項は残るはず。
- とはいえ、今までのよりはるかにまし。
- 自由表面も扱えるように拡張したい。