2022/03/22-23
CfCA / NAOJ

Athena++による シミュレーション実習 応用編

富田賢吾（東北大学）
岩﨑一成（国立天文台）
高棹真介（大阪大学）
小野智弘（東京工業大学）
杉村和幸（東北大学）
森昇志（東北大学）

Multigrid重力ソルバの紹介

ポアソン方程式の緩和法

自己重力: $$\nabla^2\varphi = 4\pi G\rho$$

$$\frac{\varphi_{j,i+1} - 2\varphi_{j,i} + \varphi_{j,i-1}}{\Delta x^2} + \frac{\varphi_{j+1,i} - 2\varphi_{j,i} + \varphi_{j-1,i}}{\Delta y^2} = 4\pi G\rho_{j,i}$$

古典的な緩和法（反復法）ではこれを（残差についての）拡散方程式と見做し、その時間極限として定常解を求める($\tau \to \infty$で左辺 $\to 0$)：

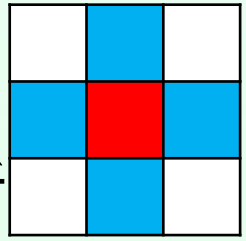
$$\frac{\partial\varphi}{\partial\tau} = \nabla^2\varphi - 4\pi G\rho$$

定常解を求めればよいので、途中経過は重要ではない
→ 最も早く拡散させるような計算法を選択する。
（注：ここで紹介する方法以外に共役勾配法等の反復法もある）

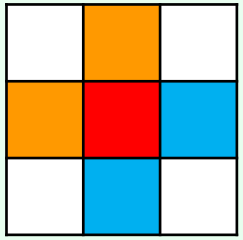
基本的な緩和法

$$\frac{\varphi_{j,i+1} - 2\varphi_{j,i} + \varphi_{j,i-1}}{\Delta x^2} + \frac{\varphi_{j+1,i} - 2\varphi_{j,i} + \varphi_{j-1,i}}{\Delta y^2} = 4\pi G \rho_{j,i}$$

Jacobi法:
前のステップの情報を
用いて**対象セル**を更新



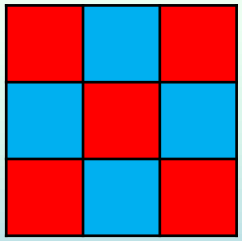
Gauss-Seidel法:
更新されたセルと
古いセルの情報を使用



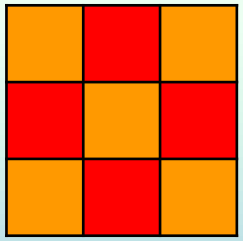
Jacobi: 簡単でベクトル化しやすいが、収束が遅い
Gauss-Seidel: 依存関係がある、違方性がある、収束もまだ遅い

改善: Red-Black Gauss-Seidel法

Step1:
赤いセルを**前の**
ステップの情報で更新



Step 2:
赤いセルを**更新された**
セルの情報を使って計算



⇒ ステップ内依存関係なし、等方（ただし赤と青は非対称）、まだ遅い

Multigrid法の概念

前項の緩和法が遅いのはこれらが拡散方程式に基づくものだから。

拡散のタイムスケール：$\tau \sim \frac{L^2}{D}$

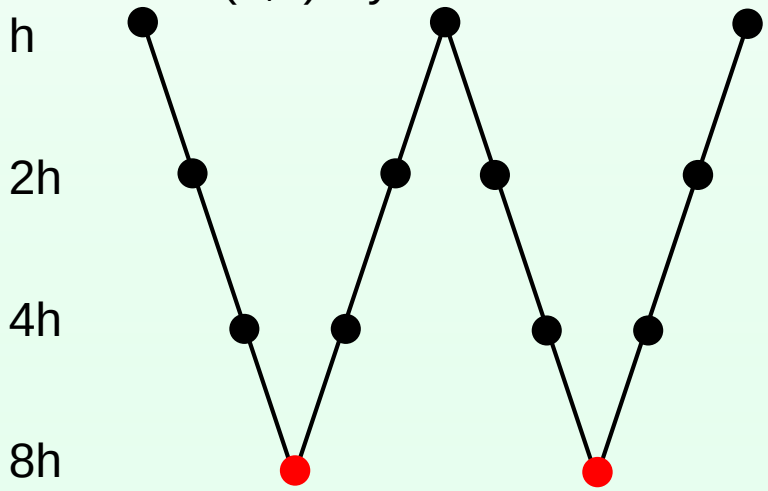
→短波長の残差はすぐに減衰するが、長波長（大スケール）の残差は
なかなか減少しない → 膨大な反復回数が必要（典型的にO(N^2)）。

Multigrid法の基本概念：
計算格子の分解能を粗くすれば、長波長モードも短波長に見える
→ 減衰が速くなる（別の言い方をすると、長い時間刻みを取れる）
元の格子、２倍粗い格子、４倍粗い格子・・・と積み重ねて、
各レベルで緩和法を適用すれば、短波長～長波長全てのモードを
均一に減衰させることができる！

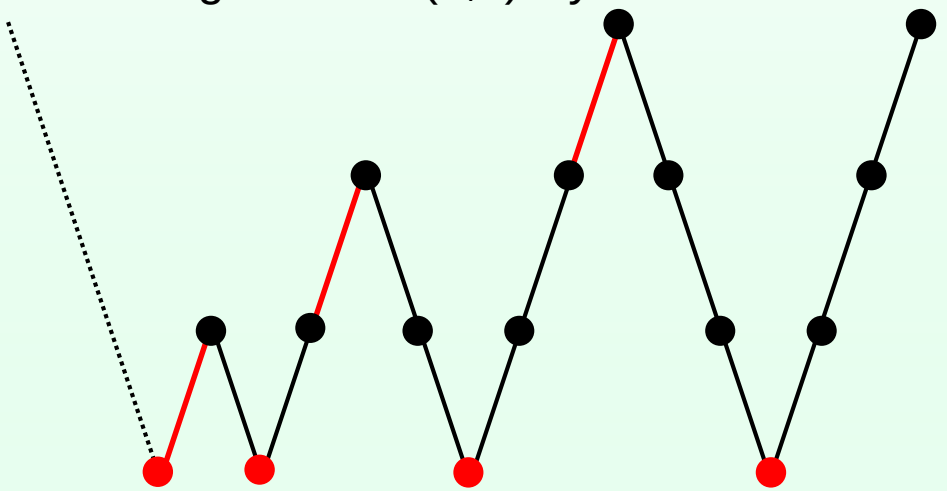
各レベルで最短波長のモードが十分に減衰すれば良い →
反復回数が分解能に依存しなくなる ⇒ 劇的に高速化

Multigrid Solvers

Iterative V(1,1) Cycle



Full Multigrid with V(1,1) Cycle



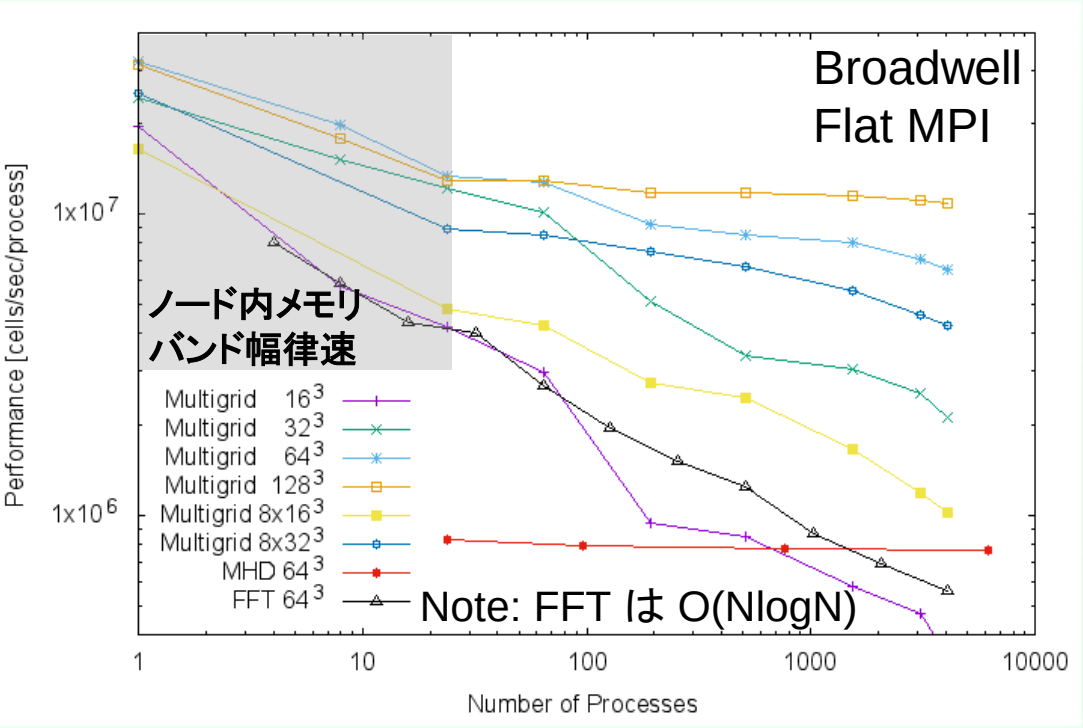
●smoothing ＼restriction ／prolongation

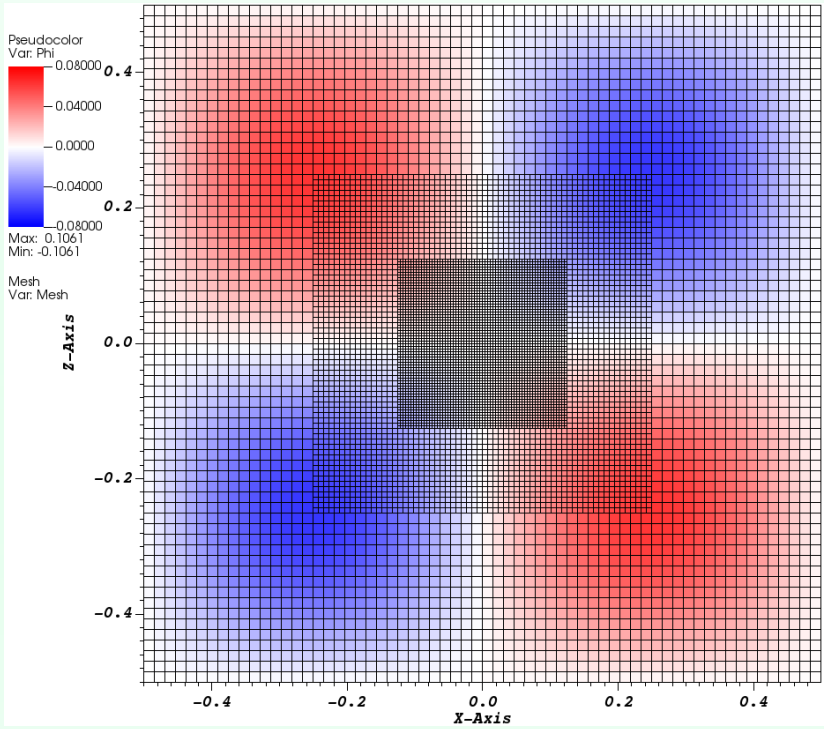
- 各レベルでRBGS法を適用
  - 全波長のエラーを一様に減衰
- 計算コスト: O(N)
- 1回の反復で残差は~1/10程度に
- 初期推定値が必要
  (前のタイムステップのポテンシャルは初期推定値としてはそれ程良くない。)

●analytic solve ／FMG prolongation

- 最も粗いレベルからはじめ、V-cycleを使って解を求める
- その結果を補間して次のレベルの初期推定値として用いる（高精度の補間アルゴリズムが必要）
- 初期推定値不要
- 必要ならV-cycleを適用し精度向上

性能とAMRへの対応





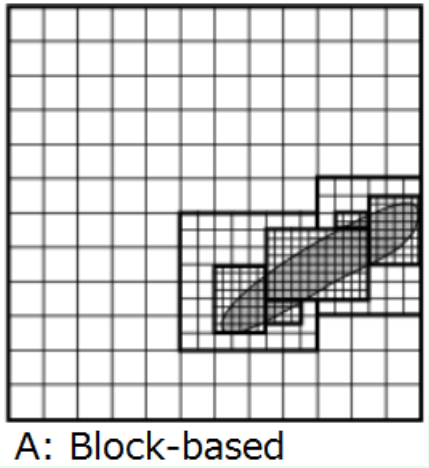
- 数千並列でもMHD部の5-10倍程度高速という高い性能を実現

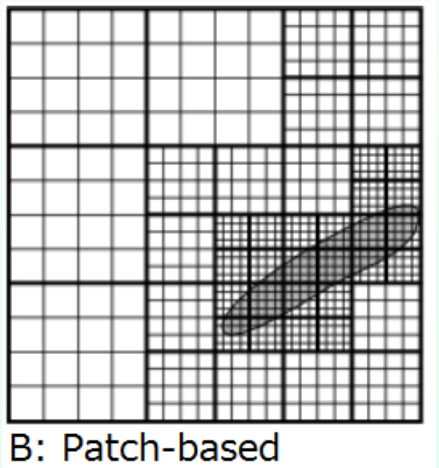
→ 既存の計算に10-20%程度の追加コストで自己重力を導入できる
（精度改善のために数回反復をかけてもMHD部より低コスト）

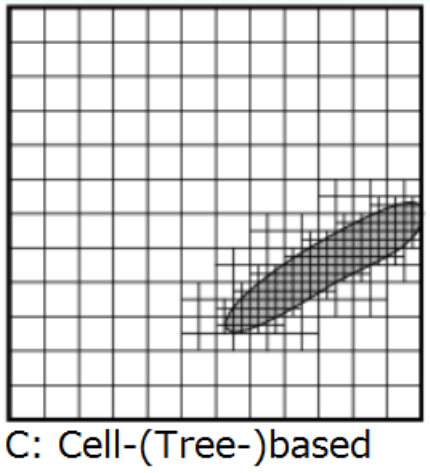
- AMRへの対応も完了、現在テストと高速化を進めている（右図）
- 多重極展開による孤立境界を実装、FLD輻射輸送等への応用も検討

Athena++の設計と性能

Athena++の格子設計


A: Block-based

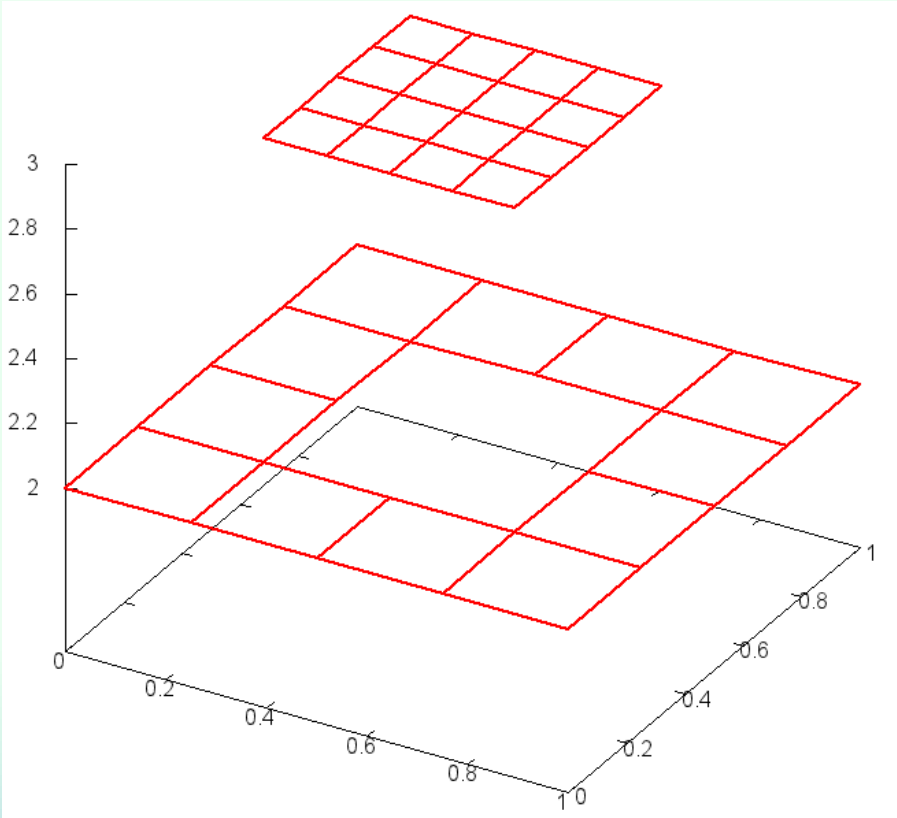

B: Patch-based


C: Cell-(Tree-)based

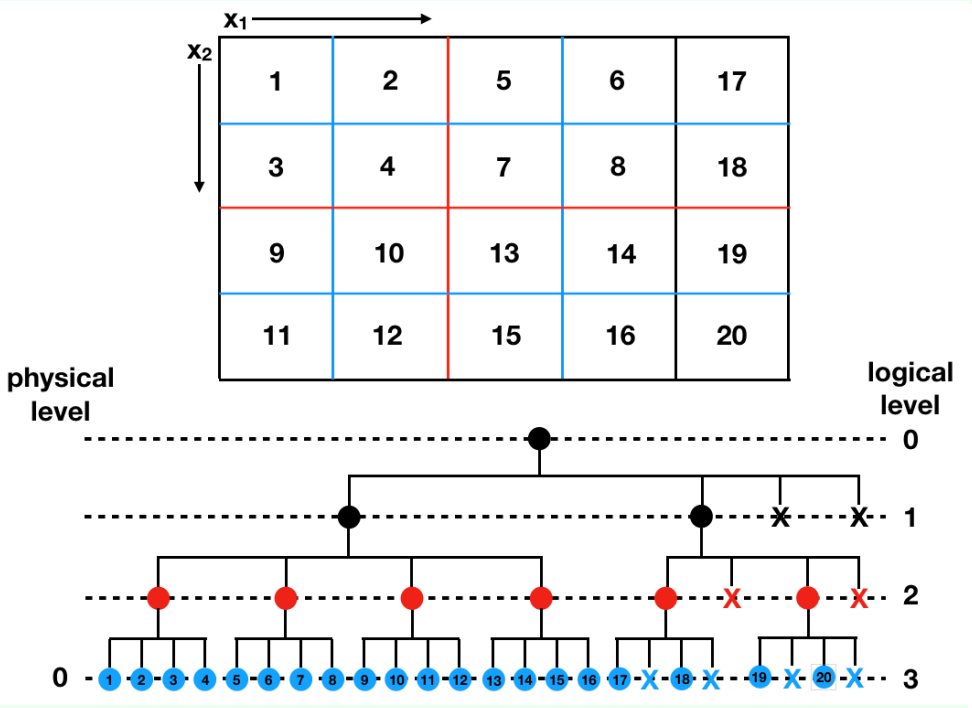
Pros	High efficiency **Uniform within block** **Use of existing scheme**	**Simple relations btw levels** **Uniform within block** **Use of existing scheme** Parallelization by space-filling curve	Highest efficiency Logically beautiful Parallelization by space-filling curve
Cons	Grids are not unique **Non-trivial grid generation** **Complex parallelization**	Lower efficiency (depending on patch size)	**Performance Issue** **Complicated grids** **(non-trivial neighbor cell)** **Hard to write,read,analyze**
Examples	Original: Berger & Colella 1989 Orion, PLUTO(Chombo) CASTRO(Boxlib), Enzo,…	FLASH(PARAMESH) Nirvana, SFUMATO, Enzo-E,… **Athena++**	RAMSES, ART

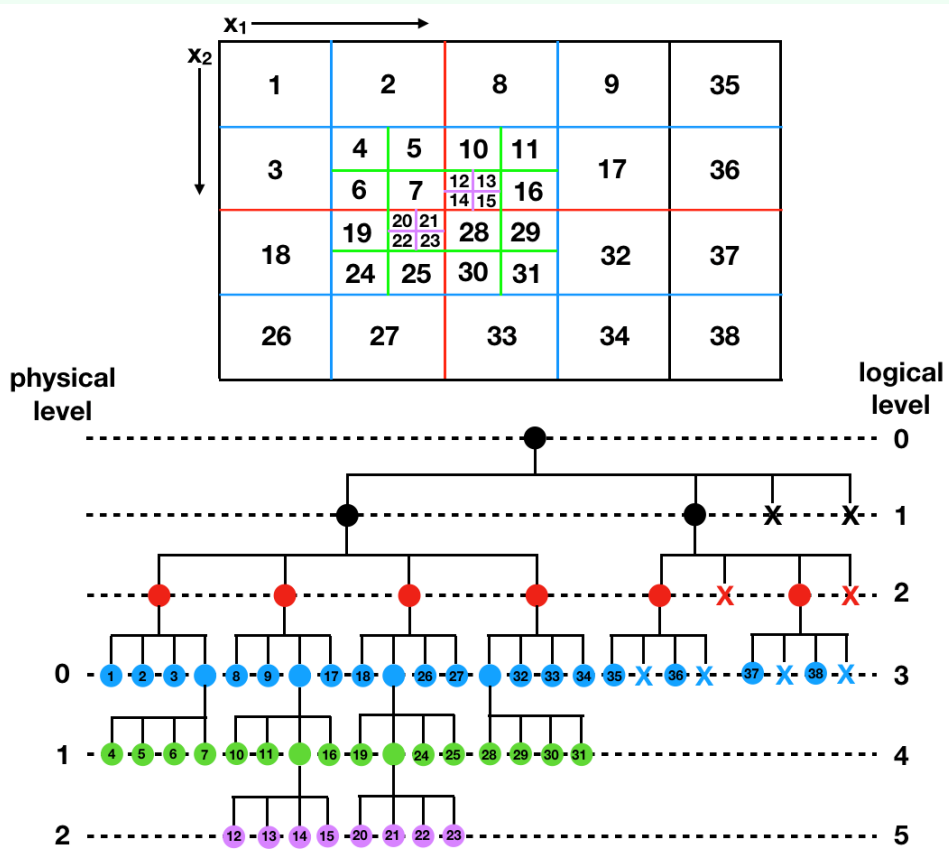
Athena++のAMRの特徴

- Athena++では細かい格子を生成した時、重複する粗い格子は消去
→空間の１点は一つ（だけ）の格子に含まれる。粗い格子には穴がある
- 一様時間刻みのみサポート
計算量は多めだが並列化が簡単かつ性能も出しやすい
実際には非一様時間刻みでのメリットはそれほど多くはない
- デフォルトでは格子数を均等するように並列化するが、これを修正する「コスト」の概念や、自動ロードバランスモードもあり



MeshBlockTree構造

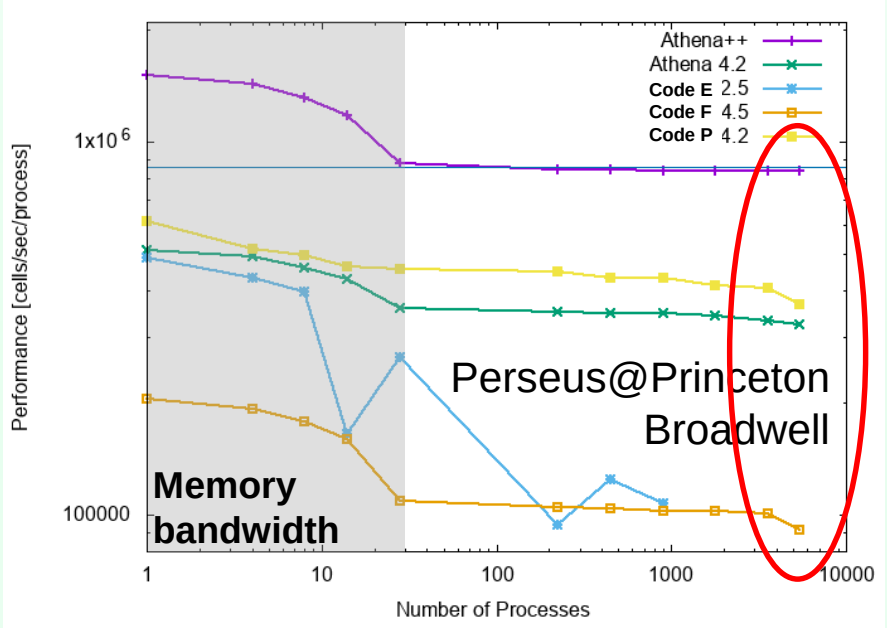


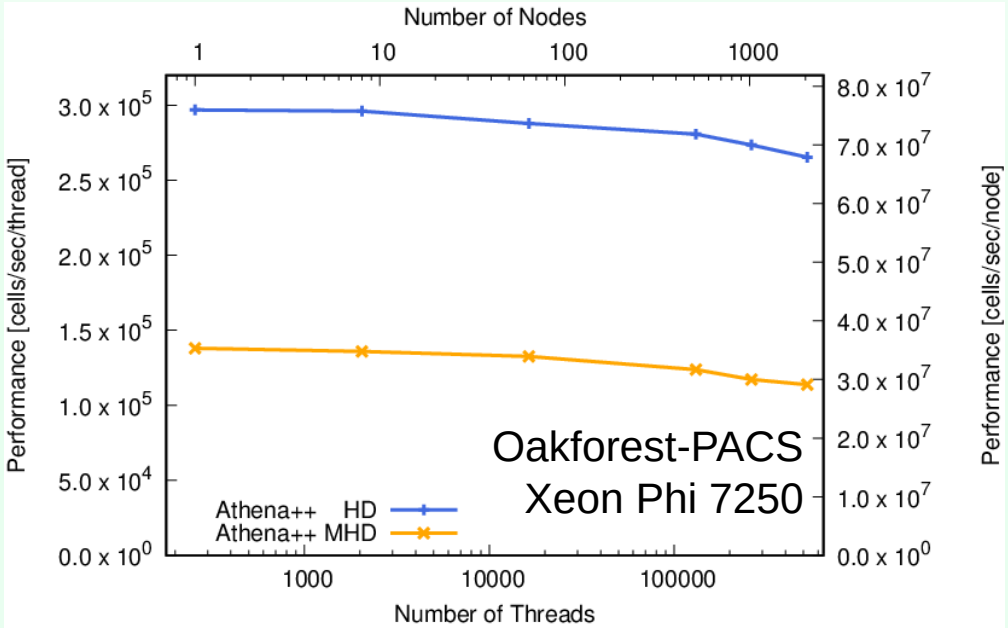


左: 一様格子, 右: AMR

- MeshBlockは8分木（3次元、2次元なら4分木）に格納・番号付け
- Global ID (gid) : Z-orderingで計算される一意なID
- LogicalLocation (loc): ツリー上での論理的位置 (lx1,lx2,lx3,level)

Athena++の性能





64^3 cells / core用いた理想磁気流体力学の弱スケーリングテスト

- ~10,000 並列でも>97%の弱スケーリング性能
- ~50万並列でも~85% という高い性能を発揮
- 公開されている宇宙物理学向けAMR MHDコードの中で最も高速
- 開発費用も非常に安い（Code EとFは大規模プロジェクト）

計算機の特徴を知る

高性能なコードを作るには計算機の構造を知ることが重要。
（CPUだけでなくGPU等複数アーキテクチャで性能を出すのはより大変）

1. ベクトル（SIMD: Single Instruction Multiple Data）化
   現代的なCPUは256bit-512bit（倍精度実数4-8個）同時に計算できる。
   ただしメモリ上連続的なデータでないと適用できないなど制限がある。
   → 使わないとCPU性能の1/4-1/8しか出ない。データ配置に工夫が必要。

2. メモリ-キャッシュ階層
   演算性能に対してメモリは低速。これを小容量・高速なキャッシュで補う。
   Skylakeの例： L1D: 32KB, L2: 1MB, L3: 1.375MB
   格子法の（磁気）流体力学は大抵メモリ律速。
   → キャッシュが有効に働くよう、データ配置に工夫が必要

3. ネットワーク（並列化）
   メモリと比べてネットワークは更に遅い（バンド幅・レイテンシ共に）
   通信は最低限に、かつ可能な限り隠蔽する

多次元データの格納方法

Fortranでは多次元配列が組み込みでサポートされている。

C言語の動的な（実行時にサイズを決めて確保する）多次元配列は、
実際には配列への多重ポインタとして実装される。
→ 確保が面倒な上にメモリ上にデータが連続に配置される保証がないため、
　高速計算には不適
C++ではstd::vectorが使えるが、やや冗長な上に連続性については同様

Athena++ではAthenaArrayテンプレートを定義（athena_arrays.hpp）し、
１次元的に連続なメモリ上に多次元データをマップしている。
→ コンパイラにとってデータの配置が分かりやすい→最適化されやすい
→ デストラクタによるメモリの自動解放などが可能

```
AthenaArray<Real> data1d, data3d;
data1d.NewAthenaArray(128);
data3d.NewAthenaArray(128,128,128);
```

データ構造：AoS vs SoA

3次元流体力学では（ρ,vx,vy,vz,P）の5変数（保存量でx2）を各セルで保持する。
データの配置には大きく分けて二種類ある：

- Array of Structures (AoS)
  （ρ,vx,vy,vz,P）をメモリ上連続に配置し、セル番号については不連続
  $[\rho_0,vx_0,vy_0,vz_0,P_0,\ \rho_1,vx_1,vy_1,vz_1,P_1\ ,\ \rho_2,vx_2,vy_2,vz_2,P_2,\ ...]$
- Structure of Arrays (SoA)
  （x方向の）セル番号について連続に配置し、セル番号については不連続
  $[\rho_0,\rho_1,\rho_2,...,\ vx_0,vx_1,vx_2,...,\ vy_0,vy_1,vy_2,...,\ vz_0,vz_1,vz_2,...,\ P_0,P_1,P_2,...]$

考えるべきこと：

- 同時にアクセスする変数はメモリ上近くに配置されている方が良い。
  格子法の流体力学は両方のアクセスパターンを含む：
  reconstruction: 変数ごと ⇔ Riemann solver, 基本量-保存量変換: 全変数セット
- 多くの場合メモリには4変数ないし8変数単位でアクセスする → AoSは不利
- ベクトル（SIMD）化にはメモリ上連続な変数に対して同じ操作をするのが良い

AthenaはAoS → Athena++ではテストコードで両方を比較した上でSoAを採用

キャッシュ・メモリバンド幅

現代的な計算機ではキャッシュはコンパイラ・CPUが自動で使うもので、ユーザーが積極的に制御することは少ないが、メモリ律速コードでは性能に直結する。
コンパイラが理解しやすいようなデータ構造にすることが重要。
キャッシュサイズは32KB(L1)~1MB(L2)→倍精度なら数千変数は乗る
→3次元データ全ては乗らないが、1次元データなら乗る
全てのデータをキャッシュに乗るようにするのは難しいが、再利用される一時データなどは必要最小限の領域に確保する方が良い

Athena++での事例:
1. reconstructionしたleft/right stateを当初は2次元配列(NVAR, NX)に格納
2. 高次精度化の利便のため4次元データ(NVAR, NZ, NY, NX)に変更
3. 1ノードフルに利用した計算で大幅な(~50%)性能低下が発覚
   1コアでは性能低下がほぼなかったため気が付くのが遅れた
   (1コアではメモリやL3バンド幅に余裕がある。性能律速要因を知ることは重要)
4. 元の実装に戻した

注:キャッシュを作用させるためにデータを小さな箱に分割する「キャッシュブロッキング」という技術もあるが、コードが複雑になるためそこまではやっていない。

ループのベクトル化

ベクトル化するにはループがメモリ上連続な変数にアクセスすることが重要。
SoA(NVAR, NZ, NY, NX)方のデータ配置をしている場合、
ループは原則として外側からN（物理変数）, Z, Y, X方向の順にする。
（Z, Y, Xの３重ループ内で複数のNVARにアクセスするのは構わない。）

```
void MeshBlock::ProblemGenerator(ParameterInput *pin) {
  for (int k=ks; k<=ke; ++k) {
    for (int j=js; j<=je; ++j) {
      for (int i=is; i<=ie; ++i) {
```

#pragma omp simd
最近のコンパイラは最適化レベルが高ければ比較的良くベクトル化してくれるが、
Athena++では多くの最内ループに#pragma omp simdを付けて明示的に
ベクトル化させている。これはループがSIMD化可能であることを示す指示文。
コンパイラによって程度は異なるが、特にIntel Compilerでは非常に強力。
ただし副作用があるケースもあるので、ベクトル化できるかどうか確認した上で、
最適化レベルによって答えが（ほぼ）変わらないかどうか、きちんとテストすること。

コンパイラオプション

多くのコンパイラでは-Oオプションで最適化レベルを制御する。
-O0は最適化なし、-O3（コンパイラによっては-O5）だと最適化レベル最大。

実はこれ以上に性能を引き出せる最適化オプションがある（詳細はマニュアル参照）。
Athena++ではIntel Compilerの場合

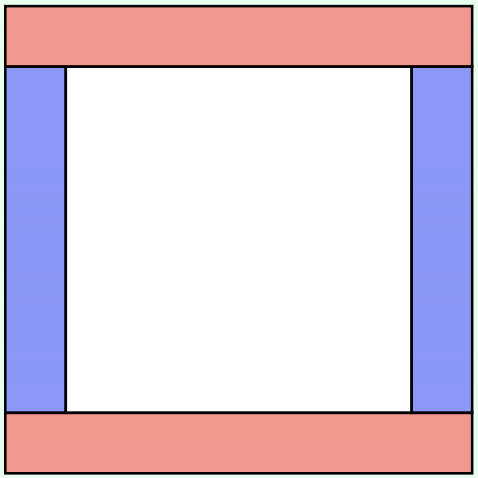
- **-ipo**（-fastに含まれる）：ファイル間最適化（関数のインライン展開を促進）
- -inline-forceinline： インライン展開を促進
- -qopenmp-simd： #pragma omp simdを有効にする
- -qopt-prefetch=4：キャッシュ利用の促進
- (-qoverride-limits: 時間やメモリがかかっても最適化を諦めない)

などを付けている。

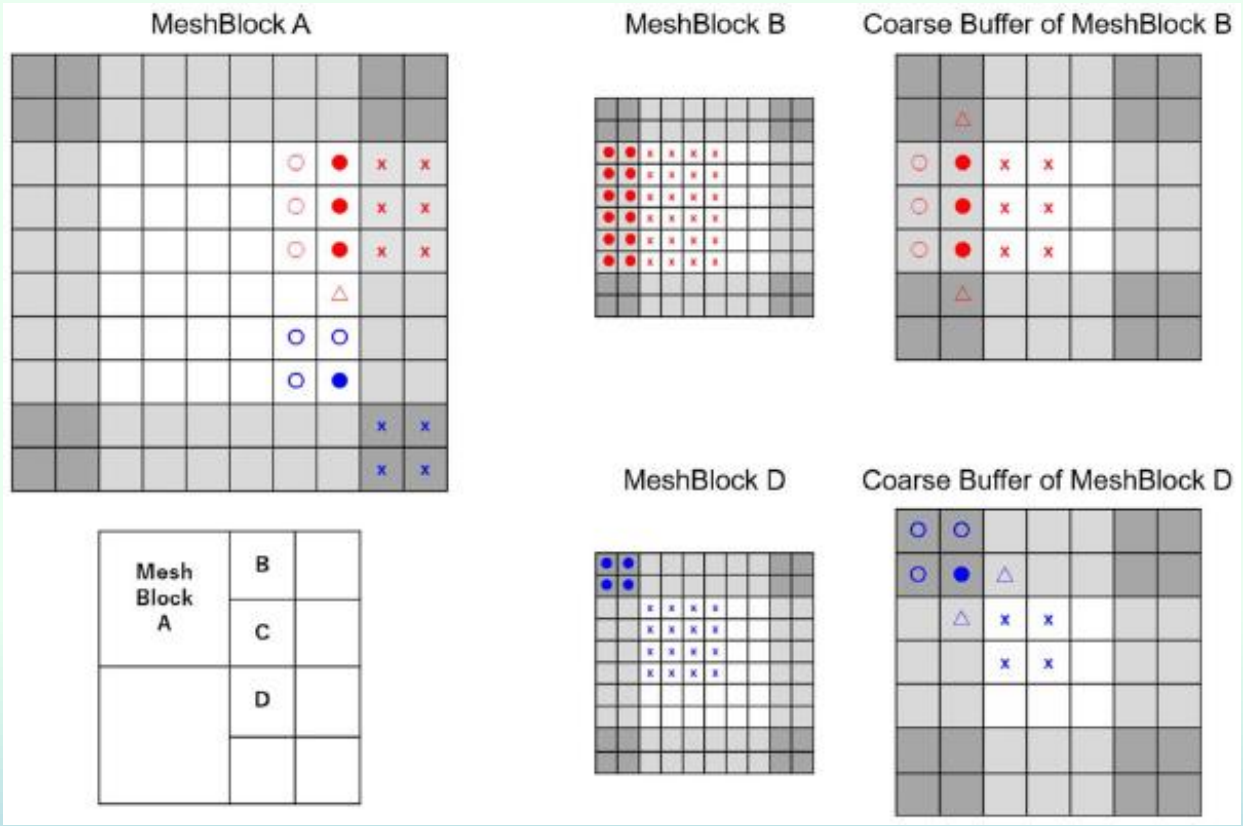
小さい関数（音速計算等）がインライン展開されないとベクトル化の支障となるため、
-ipo (Inter Procedure Optimization)は重要。
g++, Clangでは-flto（Link Time Optimization）。
ただしこれを付けるとコンパイル時間が大幅に伸びる。

通信パターン

分割したMeshBlock間では通信により「袖（ghost cells）」データを取得する。
Athena++ではこの通信パターンをAthenaから変更した。



↑Athenaではx→y→zの順に袖を通信していた。
依存関係が発生する上にAMRの実装は非常に複雑
→Athena++は全方向に対して独立・一斉に通信する。
⇒スケーラビリティの改善



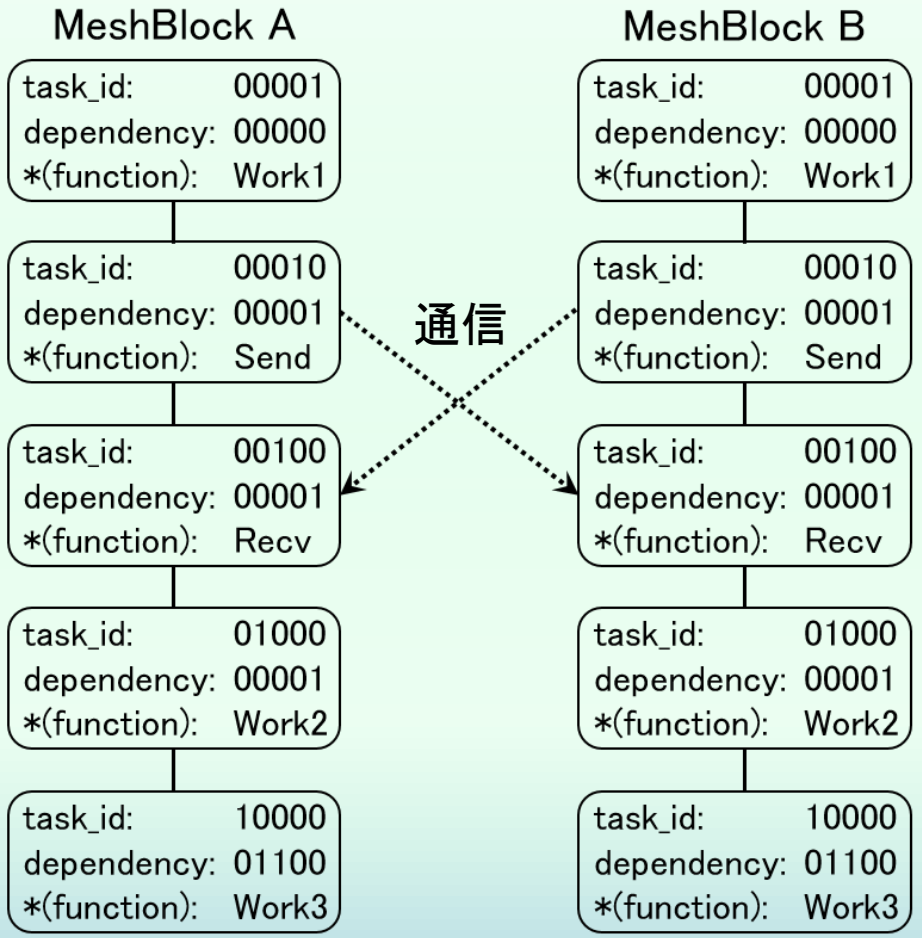
TaskList

従来のシミュレーションコードでは
各計算ステップの実行順序は固定

Athena++ではこれを小さなTaskに
分割し、その間の順序と依存関係
を記述するTaskListを導入

- TaskとTask間の依存関係を分離
→コードのモジュール化を促進
- Taskの実行順序は動的に決定
→通信と計算の同時実行が可能に

⇒開発の効率化・並列性能の向上
⇒領域・時間ごとに異なるTaskList
を適用することも可能



ハイブリッド並列化

講習会では扱わなかったが、Athena++ではOpenMPによるスレッド並列化、
及びMPIとOpenMPを組み合わせたハイブリッド並列化にも対応している。

MPIによる並列化は「分散メモリ並列化」と呼ばれ、各プロセスのメモリ空間は独立していて、明示的な通信でのみデータをやりとりする。
OpenMPは「共有メモリ並列化」であり、各スレッドがメモリを共有しており同時にアクセスすることができる。共通するデータを共有することでメモリの節約にもなる。

コード開発は単純な範囲では共有メモリ並列化の方が簡単（指示文を入れるだけ）だが、性能を出すのは実はこちらの方が難しい。
複数のスレッドがメモリの同じ領域に同時にアクセスする場合、結果を保証したり、アクセスの競合を回避する仕組みが必要であり、これらにコストがかかる。

これについて話し始めると数時間必要なのでここでは省略して、ポイントだけ紹介。

粗粒度スレッド並列化

典型的にはOpenMPによる並列化は多重ループの最も外側に対して行う:

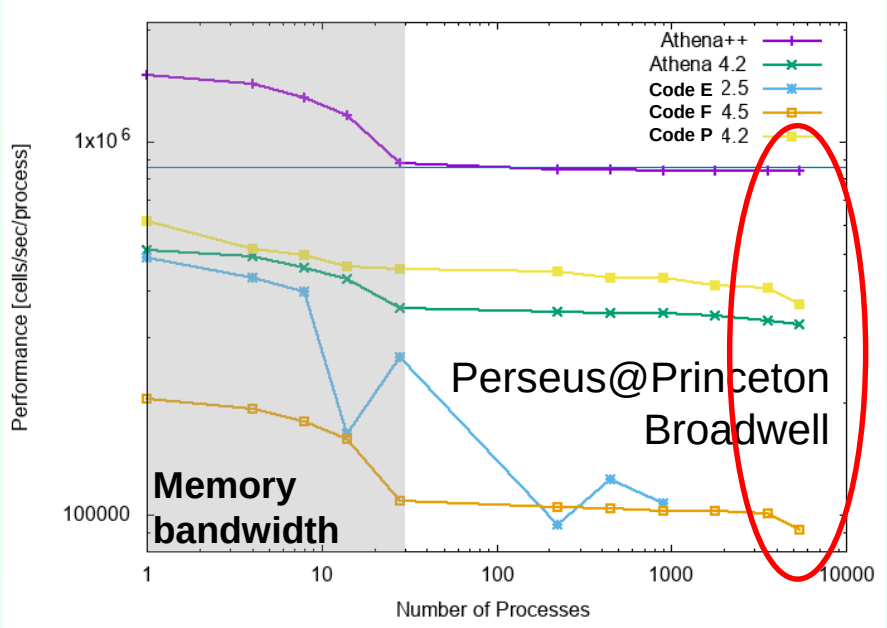
```
#pragma omp parallel for
for (int k = ks; k <= ke; ++k) {
  for (int j = js; j <= je; ++j) {
    #pragma omp simd
    for (int i = is; i <= ie; ++i) {
      ...
    }
  }
}
```

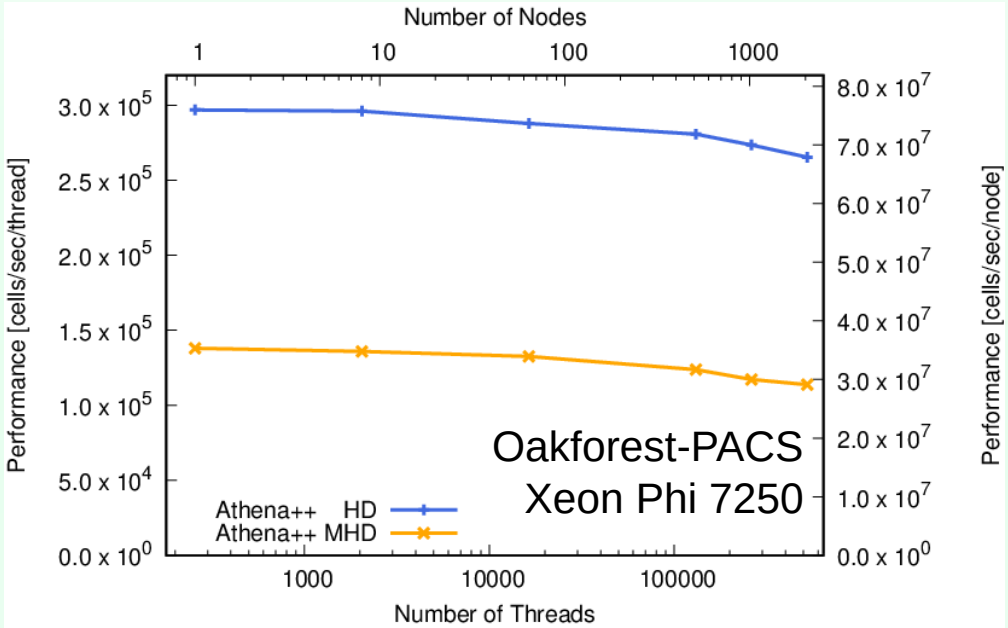
この方法だと・・・

- ループの最後で毎回スレッド間の同期が発生する(＝スレッド数が増えると遅い)。
- 全てのループに逐一指示文を入れる必要がある

```
80   while (nmb_left > 0) {↓
81 #pragma omp parallel for reduction(- : nmb_left) num_threads(nthreads) schedule(dynamic,1)↓
82     for (int i=0; i<nmb; ++i) {↓
83       if (DoAllAvailableTasks(pmesh->my_blocks(i), stage, pmesh->my_blocks(i)->tasks)↓
84           == TaskListStatus::complete) {↓
85         nmb_left--;↓
86       }↓
87     }↓
88   }↓
```

Athena++の性能





64^3 cells / core用いた理想磁気流体力学の弱スケーリングテスト

- ~10,000 並列でも>97%の弱スケーリング性能
- ~50万並列でも~85% という高い性能を発揮
- 公開されている宇宙物理学向けAMR MHDコードの中で最も高速
- 開発費用も非常に安い（Code EとFは大規模プロジェクト）

富岳への対応状況

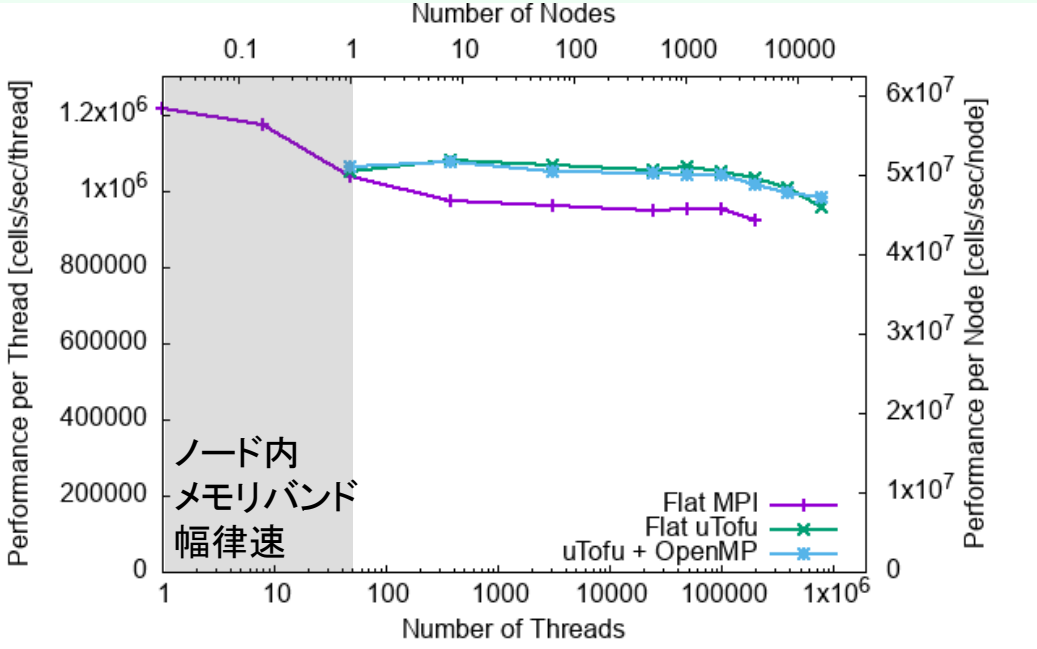
C++コンパイラのClangモードを利用
MHD弱スケーリング性能の測定→

- Intel Skylakeに対してコア当たり
  ほぼ同等の性能を実現
- HBMのバンド幅はDDR4の4倍ある
  はずなので、これでは正直不満足

→ A64fxの設計が複雑な計算に不向
（レジスタ不足、レイテンシ大）

- コンパイラの最適化性能が悪い

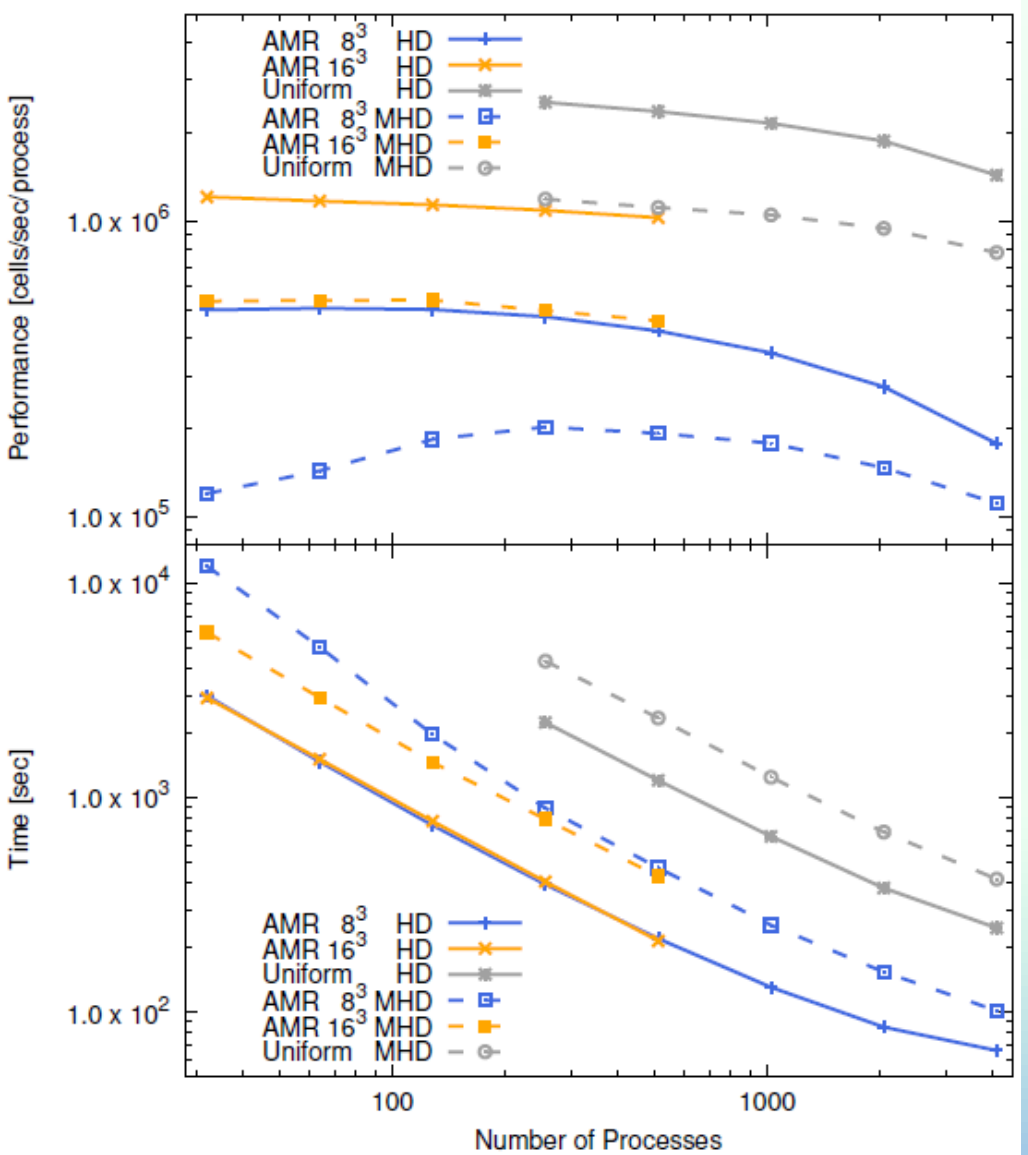
→手動でインライン展開
＋ ベクトル化するようループ変形



- Flat MPIでは大規模並列に対応できないが、富岳では
  MPI_THREAD_MULTIPLEがサポートされていない

→ uTofuによるハイブリッド並列化でさらなる大規模並列計算が可能に

AMRの性能



HD/MHD Blast wave tests
Uniform: 512^3
AMR: 3 levels with 8^3 and
16^3 MeshBlocks
2nd-order PLM,
HLLE for HD, HLLD for MHD
Cray XC50 @ NAOJ, Skylake
ストロングスケーリング

（計算時間短縮効果の半分は高温領域の分解能が下がったことで時間刻みが伸びたため）

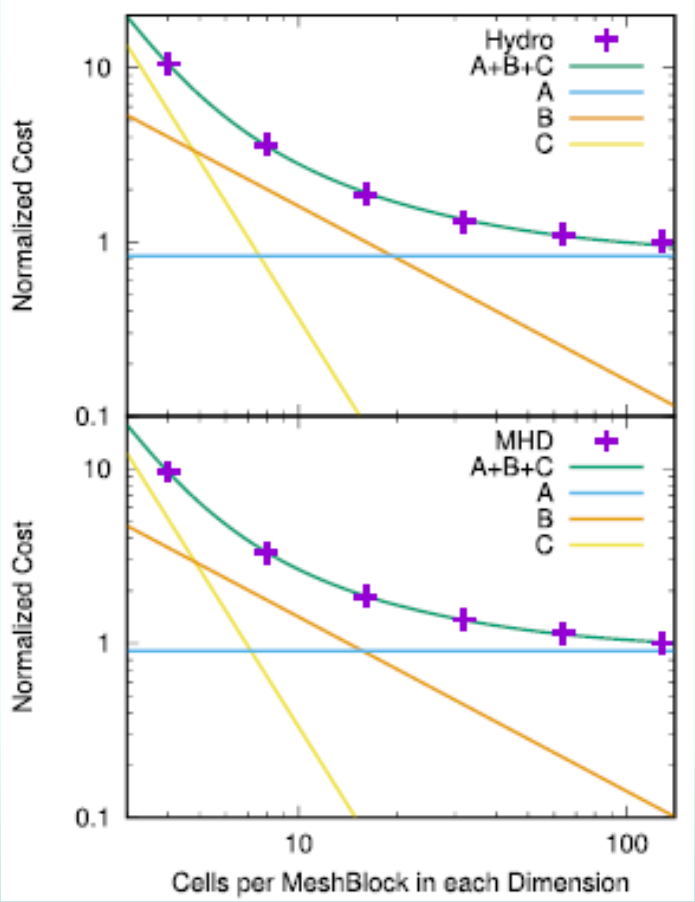
MeshBlockサイズの選び方

AMRで柔軟な格子を生成したい場合や、
使用するコア数を増やしたい場合など、
MeshBlockのサイズを小さく取りたいケースがある。
一方、MeshBlockが大きい方が性能は出しやすい。

右図：3次元計算でMeshBlockのサイズを変えた
　　時の1セル当たりの相対的計算コストの変化
A: 実際の計算に必要なコスト
B: 表面積に比例する袖通信や境界条件のコスト
C: ブロックの個数に比例するコスト

⇒ 一様格子なら32^3以上、AMRでは（柔軟性を確保するために）8^3-16^3程度を目安にするのが良い。

ただし使用している物理や計算機によることに注意。



どこを読めばいいか

流体ソルバを理解したい → hydro/
磁場ソルバを理解したい → field/, hydro/rsolvers/mhd/
グリッドの構造を理解したい → mesh/
座標系の取り扱いを理解したい → coordinates/
状態方程式を理解したい → eos/
計算の流れを理解したい → main.cpp, task_list/
並列化を理解したい* → mesh/, bvals/（*: 魔境です）

Athena++の「メインループ」: TaskList::DoTaskListの以下の部分。

```
80   while (nmb_left > 0) {
81 #pragma omp parallel for reduction(- : nmb_left) num_threads(nthreads) schedule(dynamic,1)
82     for (int i=0; i<nmb; ++i) {
83       if (DoAllAvailableTasks(pmesh->my_blocks(i), stage, pmesh->my_blocks(i)->tasks)
84           == TaskListStatus::complete) {
85         nmb_left--;
86       }
87     }
88   }
```

コードの品質向上とツールの利用

→Webのチュートリアル参照

「良い」シミュレーションのために

あるいは「シミュレーションのダークサイド」

良いシミュレーションとは？

ここまで体験してもらったように、シミュレーションコードを使えばそれらしい答えは割と簡単に出る。しかし、結果が出てもそれが正しいという保証はないし、公開コードを使っていても結果が正しいことを証明するのはユーザーの責任。

では、シミュレーション結果が正しいことはどう証明すれば良いだろうか？

- 少なくとも理想化された状況について解析解を再現することを確認する

→ 解析解がない複雑な問題も多々ある（解がわかっているならそもそもシミュレーションは不要）
→ Kelvin Helmholtz不安定性等の単純な問題なら線形解析で分散関係がわかる

- 分解能を上げた時に同じ結果が得られること（＝収束性）を確認する
- 解が満たすべき物理的性質を持っていることを確認する

→ 対称性 - ただしシミュレーションでは数値誤差のため対称性が破れることがある
→ （非自明な）保存量 - 保存形では$\rho, \rho\boldsymbol{v}, E$は自明に保存するのでそれ以外

- 解の振る舞いが少なくとも定性的に物理に基づいて理解できる

→ どの物理過程が支配的か？依存性は説明できるか？
→ 既知の現象（不安定性・安定性）との関係？

⇒ Enterキーを押して出てきた結果を垂れ流すのは良いシミュレーションではない。

シミュレーションのダークサイド

離散化された方程式を解く数値シミュレーションは何かしらの意味で
須らく「近似」であることにまず注意してほしい。

数値シミュレーションは実験が不可能な宇宙物理学分野において
複雑で非線形な問題を解く事実上唯一の手段ではあるが、
その道には様々な罠や落とし穴がある。
中には、皆が平然と渡っているが実は危ない橋まである。

ここではそれら例を紹介して、シミュレーション結果を盲信せずに
きちんと自分で物理に基づいて理解することの重要性を語りつつ、
ここまでで扱いきれなかった微妙なノウハウなどを共有したい。

宇宙物理学の特殊性

流体力学シミュレーションを使う分野は工学から理学まで非常に幅広いが、
その中でも宇宙物理学は他の分野と比べて非常に特殊である。

- 超音速・圧縮性・高レイノルズ数
→ 多くの分野では非圧縮・低レイノルズ数流体の方が多い。
- 対象の幾何学的形状が単純なのでメッシュ形状に対するこだわりが薄い
→ 工学分野ではメッシュ生成自体が研究対象になる
- 解の誤差や収束性に対するこだわりが薄い
→ 衝撃波では1次精度しか出ない、そもそも観測の精度が低いから？
- 計算している物理過程のバリエーションが多い
→ 基本は共通でも天体ごとに違う物理過程が重要になる

他分野のシミュレーション研究とは主流の手法・アルゴリズムが異なる。
新しいアルゴリズムを考えてみたら、他の分野では既に知られていた、
ということもあるので幅広く情報収集することは重要。

収束性

適切なアルゴリズムを使えば、分解能を上げれば離散化した系も
元の方程式の正しい近似解に十分近づくことが期待される。

では、「十分近い」ことはどうやって確かめればいいのだろうか？

- 数値的に誤差を評価してそれが十分（例えば0.001%以下）小さいことを示す

→ 厳密だが、宇宙物理では衝撃波が多数現れ、不連続面では一次精度しか出ない（←Godunovの定理）ため、この意味での誤差を十分小さくするのは難しい。

- 興味ある現象の物理的なスケールを十分分解する

→ 重力ならジーンズ波長、放射冷却なら冷却長、線形不安定なら不安定波長
→ 乱流等、物理過程によっては無限に小さい構造を生じ得る

- 分解能に対して解の振舞を調べ、その影響を予測する（cf. リチャードソン補外）
- 少なくとも興味ある物理量が分解能に大きく依存しないことを確認する

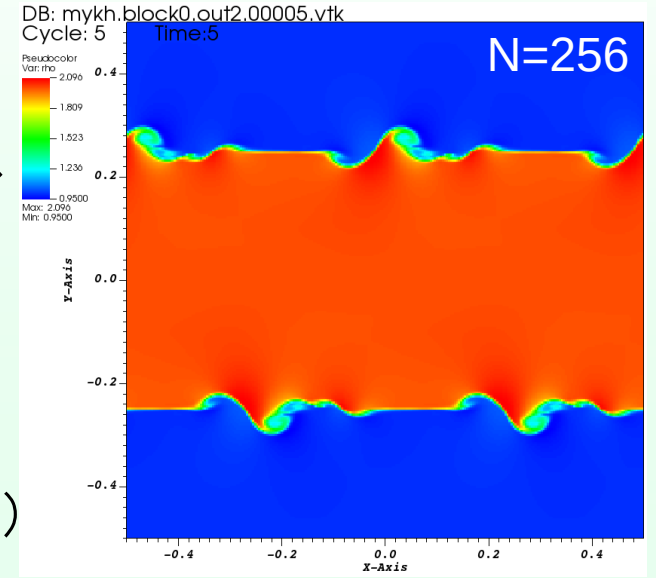
→ 小スケールの乱流は分解できなくとも平均的・統計的性質は収束することがある
→ 小さい構造を全て分解することが系の理解に必要とは限らない

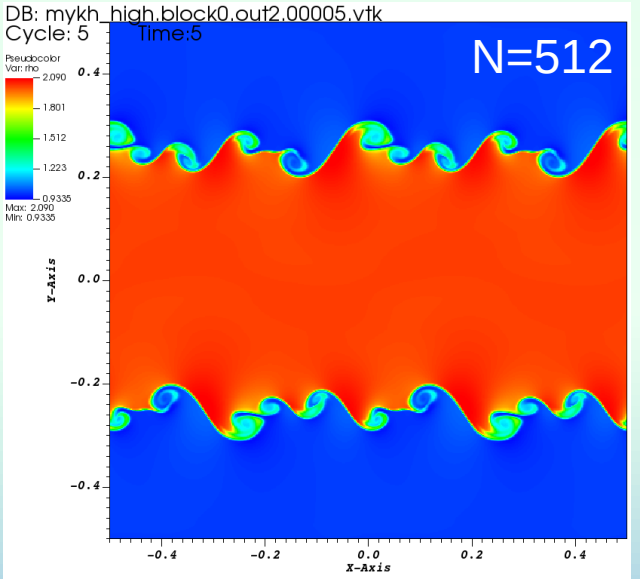
⇒ 良いシミュレーションには対象の深い物理的理解が不可欠。

収束性に関する注意

物理的な散逸のない系では最小スケールはセルサイズ
→ 分解能を上げるとより細かい構造が発生して、
どこまで行っても厳密には収束しない場合がある。

- 無限小の波長が不安定な場合
  例：境界層の厚みがゼロのKH不安定（→右図）
- 粘性なしの乱流（カスケードにより小さい構造ができる）
- 熱伝導なし・温度ゼロまで到達する冷却不安定





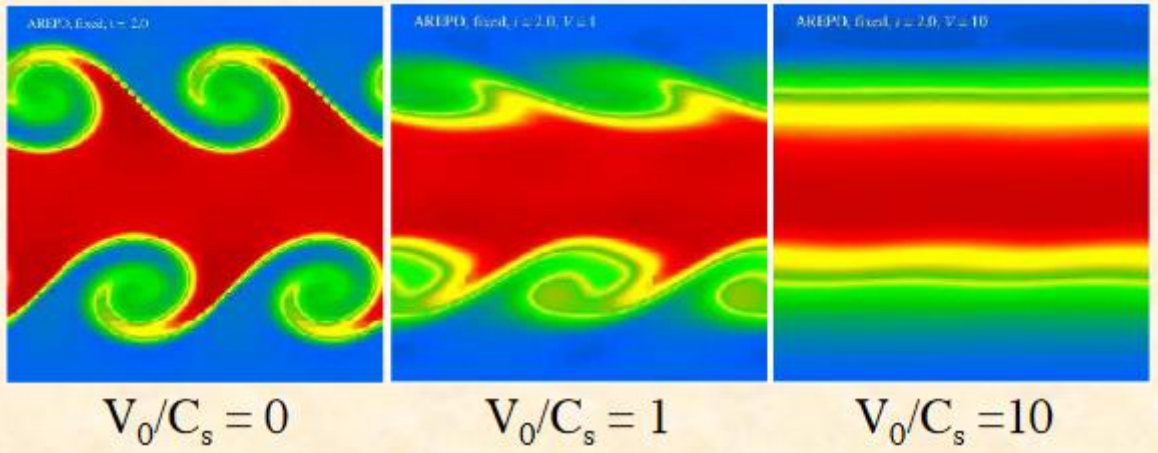
厳密な意味での収束には、小スケールの構造を
抑制する（可能なら物理的な）何らかの機構を導入し、
その機構で決まる最小スケールを分解する必要がある。

- KH → 有限厚さの境界層では短波長は安定化する
（ただし、一般に長時間非線形発展を追うにはやはり粘性等が必要）
- 乱流 → 粘性を入れて小スケールの渦を抑制する
- 冷却 → 熱伝導を入れて小スケールを安定化する

⇒ただし宇宙物理では粘性や熱伝導が効くスケールは
非常に小さく、現実のスケールを分解したシミュレーションは困難 → 注意が必要

Eulerianコードはガリレイ不変？

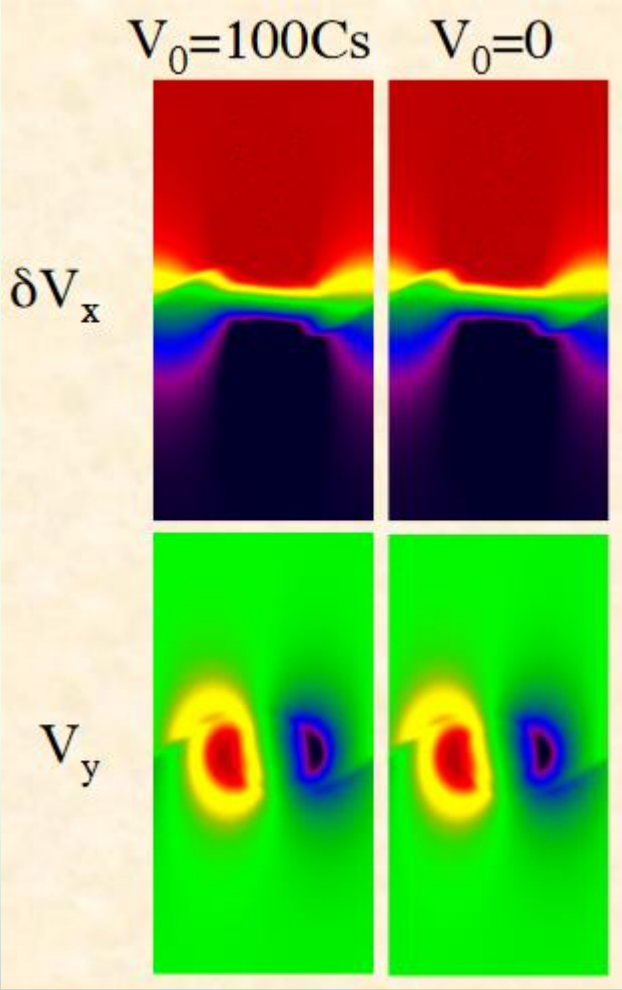
Lagrange法コード（SPH, moving mesh等）からの良くある批判：
「オイラー法＋格子法のコードはガリレイ変換に対して不変ではない」



(↑ Jim Stoneの講義ノートより)

Kelvin Helmholtz不安定性をX方向にブーストした系で解くと、
ブースト速度が超音速の時著しい精度低下を起こし、真の解に収束しない？
⇒ オイラー法＋格子法のコードで超音速流を計算してはいけない？？？
（超音速の流れに対してRiemann Solverは常に風上側の流束を返す）

打切り誤差はガリレイ不変でない



(↑ Jim Stoneの講義ノートより)

実はこれはコードの本質的問題ではない（ほっ）。
前頁の問題は初期に不連続（分解できていない）境界面を置いたKelvin Helmholtz不安定性問題。

このような不連続面での打ち切り誤差は分解能に依存するだけでなく、実はガリレイ不変でない。
その結果ブーストした系で誤差が卓越する。

← 十分に分解された解に対しては打切り誤差はブーストした系でも十分小さくなる。

大事なことは「**必要な構造を十分分解すること**」

注：十分分解能を確保できるかどうかは別問題。
また、このような計算は非常に高コスト。

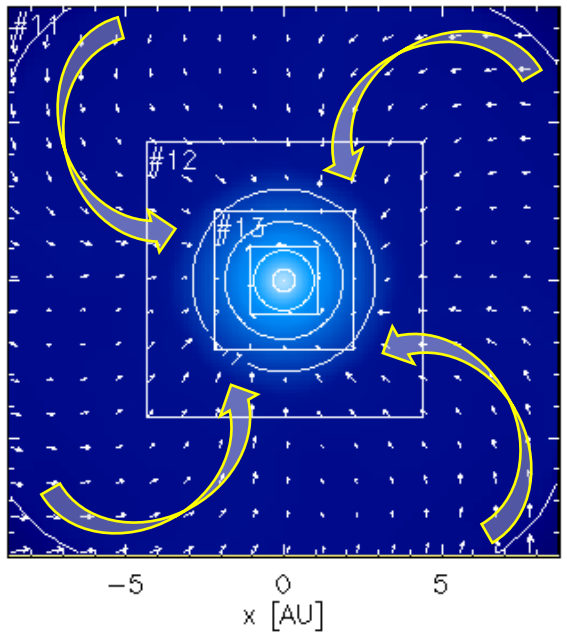
初期摂動 vs グリッドノイズ

数値シミュレーションでは計算領域を格子に離散化して計算する。
→ セルスケールでの「ノイズ」が必ず発生してしまう。

特に（線形）不安定な現象は格子の構造に起因するノイズからでも成長する。
⇒適当な初期条件を入れると摂動の強さ→結果が分解能に依存してしまう。

例：デカルト座標系で丸いものを解くと
格子に起因するm=4モードが卓越することがある。
（右図：Tomida et al. 2013, 初期摂動はm=2のみ）

初期条件や解像度に非物理的に依存しないよう、
格子ノイズよりも振幅が大きく、かつ波長が長い
摂動を入れるか、格子ノイズが結果に影響していない
ことを確かめる必要がある。



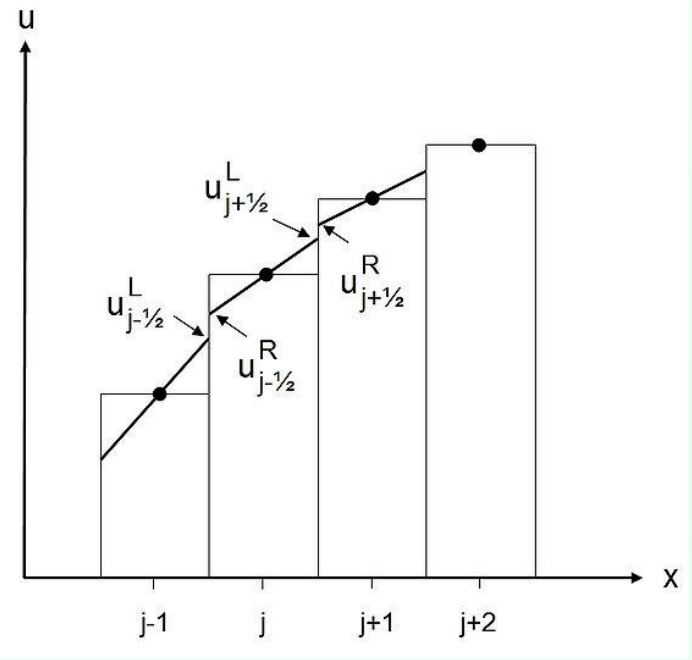
補間の話

流体計算ではあらゆる局面で物理量の補間を行う。
しかし、実はこの時変数間の物理的な関係を全て
同時に満たすことはできない。
例:
基本量 ρ, v を使って空間的な線形補間を行う。

$$\rho(x+\Delta x) \sim \rho_L + \frac{d\rho}{dx}\Delta x$$

$$v(x+\Delta x) \sim v + \frac{dv}{dx}\Delta x$$

$$\rho v(x+\Delta x) \sim \left(\rho + \frac{d\rho}{dx}\Delta x\right)\left(v + \frac{dv}{dx}\Delta x\right)$$



(図:Wikipedia)

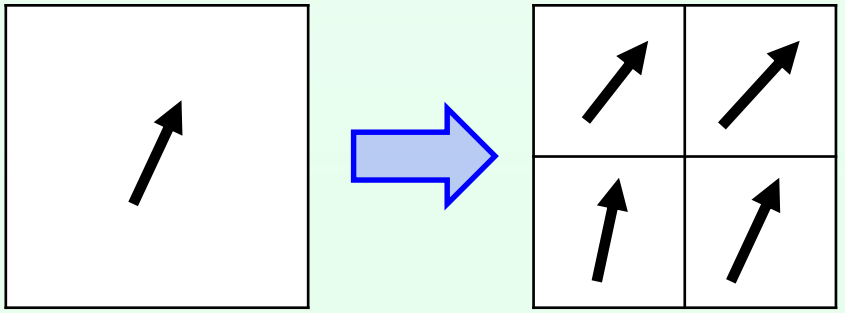
非線形な分布となり、必ずしも元の物理量の分布の性質を反映しない可能性がある。
例えばこの例ではセル中心の物理量に「質量流束一定の分布」を与えたとしても、
セル境界で計算される質量流束は(Δx の高次の項では)一定ではないかもしれない。
(注:分布が滑らかかつ十分高分解能であればこの違いは十分小さくなるはず。一方不連続面
ではリミッタが効いて勾配がゼロになるため、やはりこの差は消える。なので大丈夫・・・?)

AMRにおける補間

補間に関する非物理的な振る舞いはAMRで格子を生成・破壊する時に顕著になる。
AMRでは高解像度の格子を生成する際は通常保存量を保存するように補間を行う。

この時、右図のように生成された格子の内部には有限の速度分散が発生する。
そこでその全運動エネルギーを計算すると

$$E_{\mathrm{kin}} = \sum \frac{1}{2}\rho(\boldsymbol{v}+\delta\boldsymbol{v})^2\Delta V_{\mathrm{f}} > \frac{1}{2}\rho\boldsymbol{v}^2\Delta V_{\mathrm{c}}$$



（ΔVはセルの体積で$\Delta V_{\mathrm{c}}=8\Delta V_{\mathrm{f}}$）となり、低解像度の格子より大きくなってしまう。

この状況で全エネルギーを保存すると、その皺寄せは全てガスの内部エネルギーに行く。つまり、AMRで格子を生成すると温度（エントロピー）が減少してしまう。
（格子を破壊する時は逆）これは段数の多いAMR計算で問題になることがある。

これは勾配を使った補間は「無から有を生み出す」非物理的操作であるため生じる。
基本量または温度で補間すれば回避できるが、保存則を満たせなくなる。
空間補間をせず１次精度で計算すると今度はノイズが大きくなってしまう。

⇒ これはAMRの本質的問題かもしれない（？）

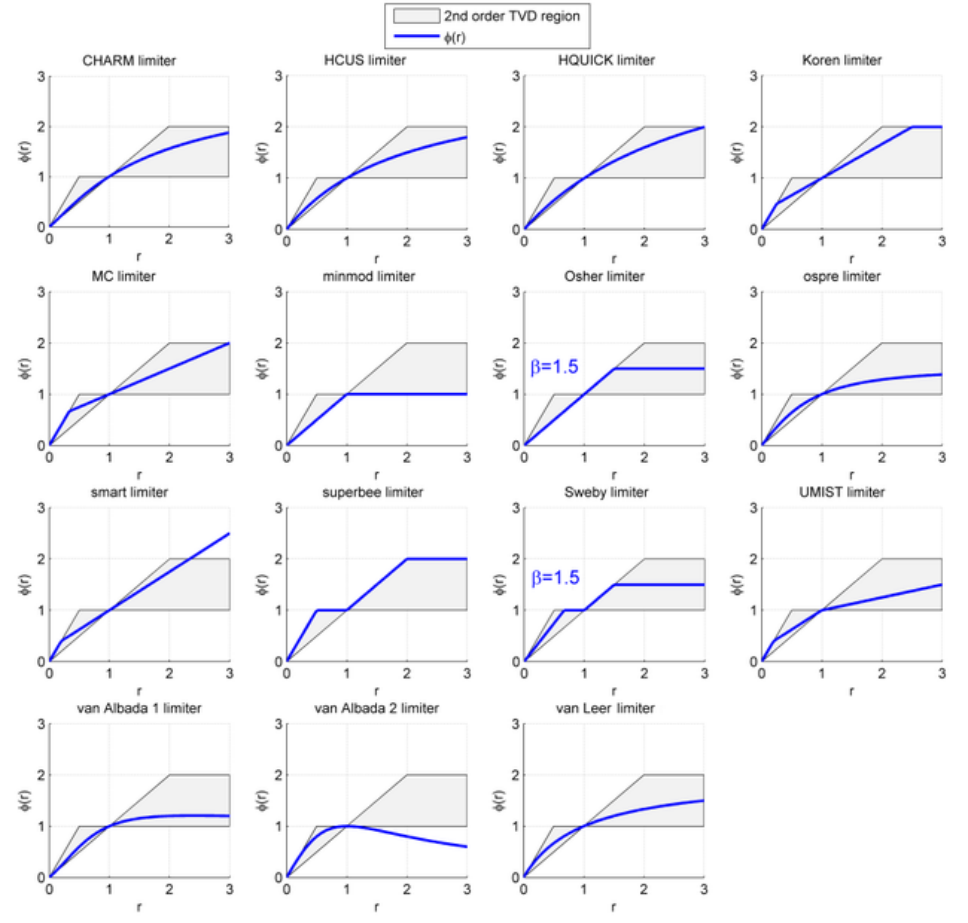
Slope Limiter

高次精度スキームでは不連続面等
での安定性を確保するために
Slope limiter（勾配制限関数）を使う。

これには様々なバリエーションがある
→ Sweby plot（灰色の領域が安定）
- 左右対称
- TVD条件を全域で満たす

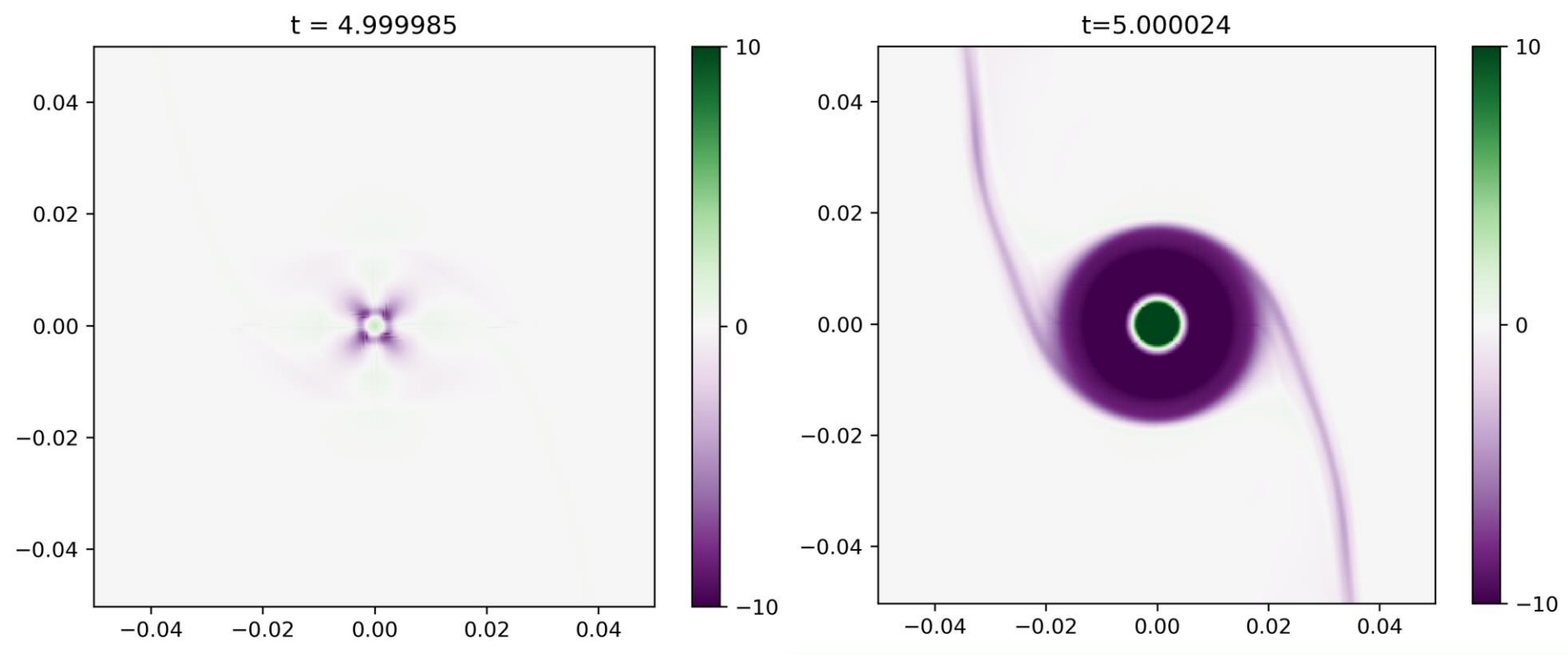
ものが良い。

Athena++ではvan Leerが標準。
数値不安定を抑制するためには
拡散強めのminmodに替えると良い。
他にはMCが良く使われる。
必要ならreconstructionを書き換える。



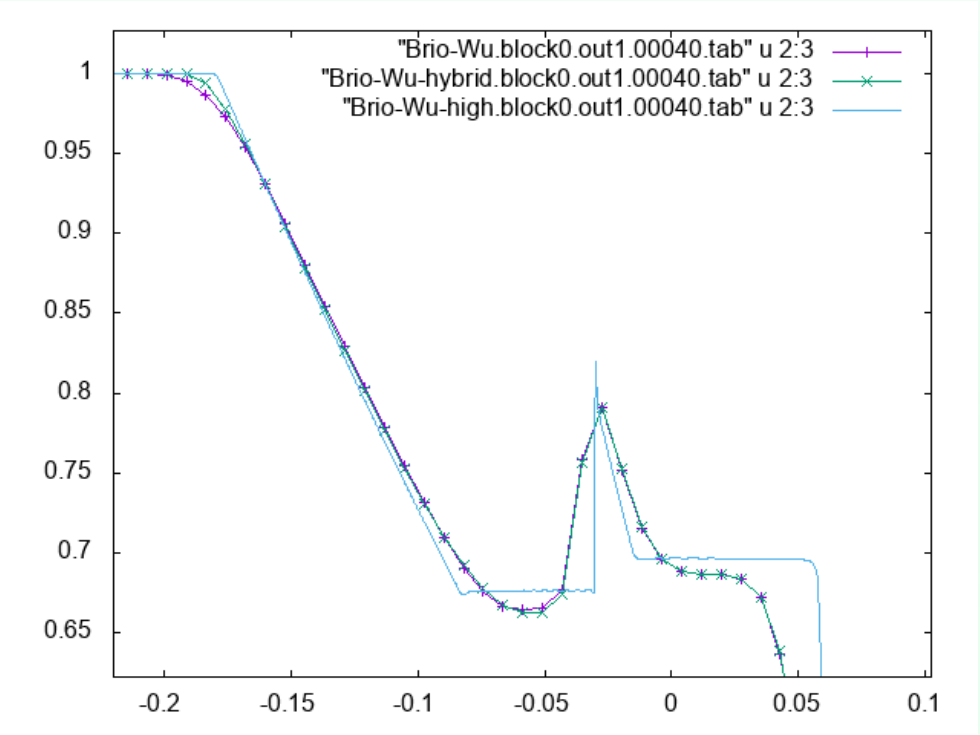
（図：Wikipedia https://en.wikipedia.org/wiki/Flux_limiter）

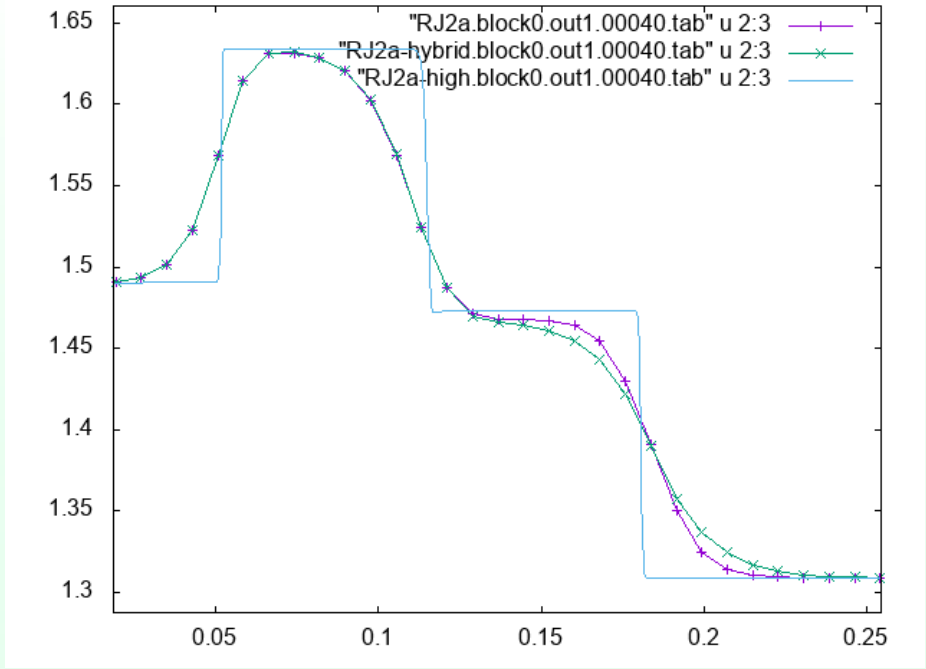
Slope Limiterで結果が変わる例

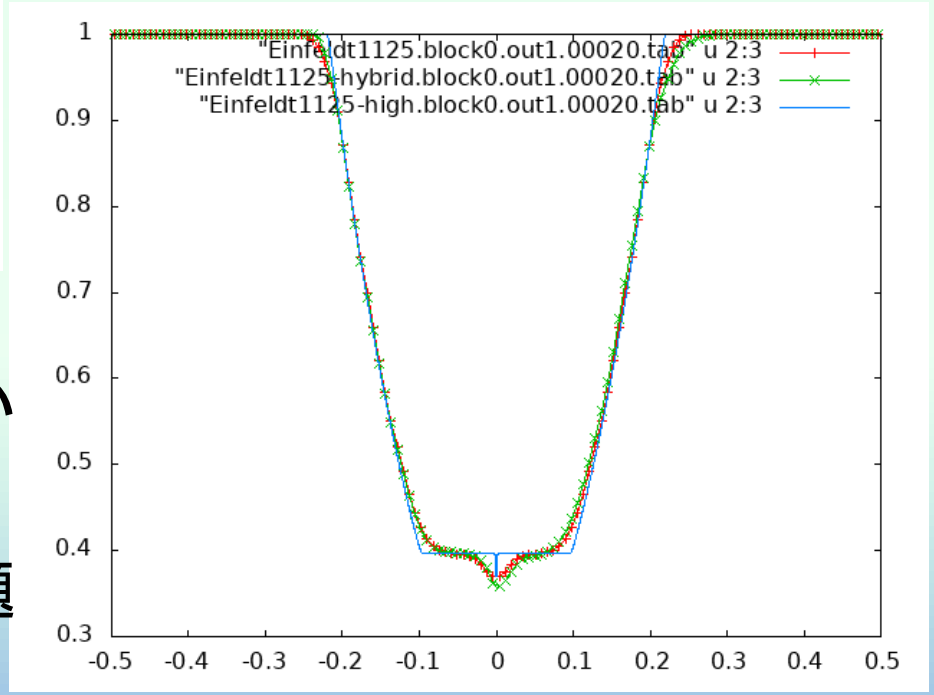


2D等温非粘性円盤中の惑星の局所計算。Vortensityはゼロのはず。
左: PLUTO - 密度はMC, 圧力はminmod, それ以外はvan-Leer
右: Athena++ - 全変数に対してvan-Leer
どちらも2次精度だが、limiterの選び方で顕著に結果が変わるケース

Limiterの比較







問題によってlimiterの適不適は違う
Brio-Wuはhybrid limiterがやや良い
RJ2a・Einfeldt1125ではVLが良い

→どれを標準にするかは難しい問題

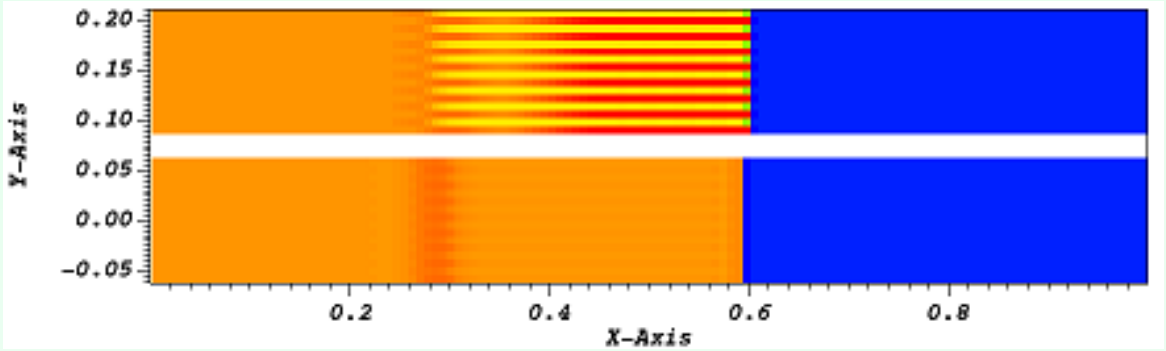
Carbuncle Phenomenon

基本的には数値計算手法は散逸が少ない、より物理を正しく含むものを選ぶべき。
流体力学ならHLLEよりHLLC（接触不連続面を分解できる）
磁気流体力学ならHLLEよりHLLD（Alfven波・接触不連続面を分解できる）

ところが、低散逸・高解像度スキームの方が悪くなるケースが知られている。

HLLD →

HLLE →



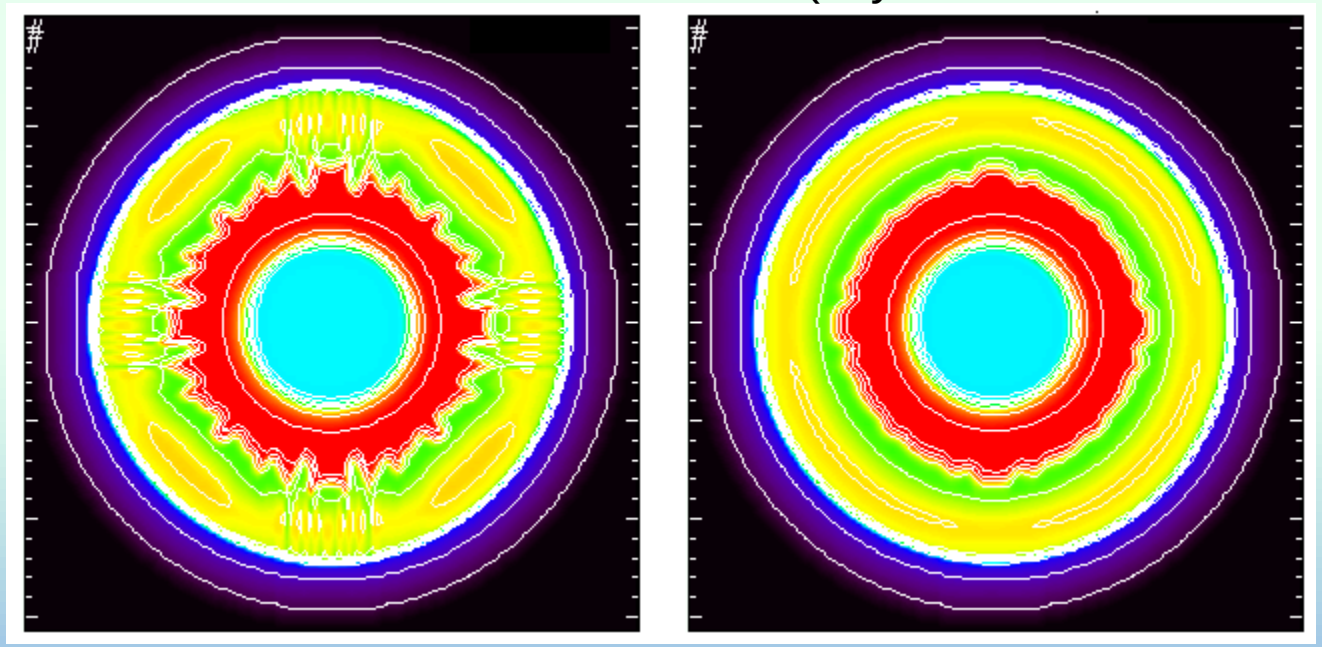
「**カーバンクル現象**」（J. Quirk 1994 の磁場有版）:
Rankine-Hugoniot関係を満たす衝撃波に5%のodd-even 摂動を入れる。
これは物理的には安定であるはずだが、HLLDやHLLC、Roe法は増幅してしまう。
これは衝撃波面がグリッドに揃っている時のみ起こる数値的問題と考えられている。
良いはずのスキームが簡単な状況で失敗する、ある意味皮肉な問題。

カーバンクル現象対策

カーバンクル現象は高解像度リーマンソルバの返す流束に
シア方向（vy, vz）の数値粘性が不足しているために起こると考えられている。
そこで、衝撃波を検出してその周辺にだけ追加の粘性を入れれば良い。
Athena++ではLHLLD(Minoshima & Miyoshi 2021)を実装している。

HLLD

衝撃波検出(Hanawa et al. 2008)＋
HLLD- (Miyoshi & Kusano 2007)



(Tomida et al. 2013)

Operator Splittingの話

拡散方程式の所で見たように、複雑な物理過程を導入する際には
operator splitting（演算子分離法）が良く用いられる。
これは時間精度は低下するものの、実装が簡便なので実用上重要。
特に流体部よりも速い（時間刻みが短い）物理は分離して解きたい。

そのような「速い」物理過程が複数ある時はOperator Splittingは要注意。
それらが独立（異なる変数しか使用しない）ならば問題ないが、
同一変数を扱う場合には分離すると問題を起こすことがある。
可換ではない演算子を分離すると、計算順序によって結果が変わってしまう。

例：輻射輸送と輻射-ガス結合を陰解法で解く（次頁）
これらはどちらも流体力学部よりも圧倒的に速いプロセス。

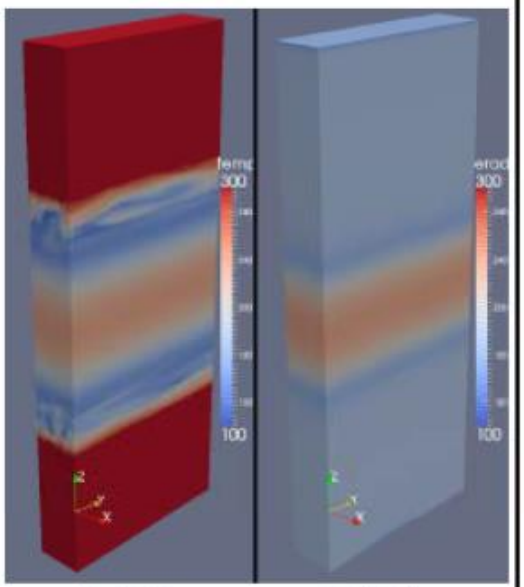
Operator Splittingが不適切な例

廣瀬さん@JAMSTEC提供

同時に解いた場合の解

$$\frac{\partial e}{\partial t} = -(4\pi B - cE)\kappa\rho$$

$$\frac{\partial E}{\partial t} = (4\pi B - cE)\kappa\rho - \nabla \cdot \boldsymbol{F}$$



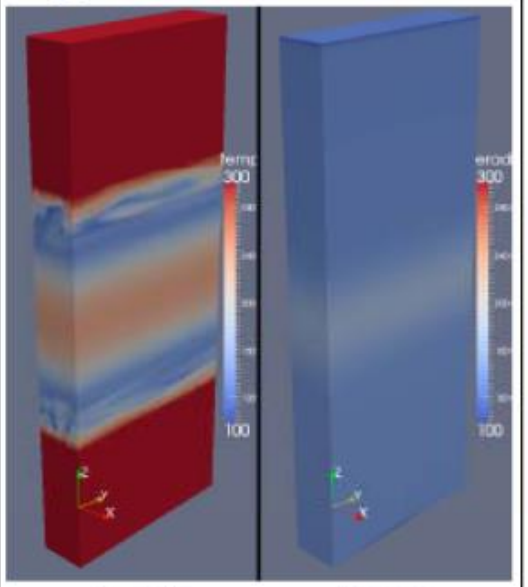
① 輻射-ガス相互作用

$$\frac{\partial e}{\partial t} = -(4\pi B - cE)\kappa\rho$$

$$\frac{\partial E}{\partial t} = (4\pi B - cE)\kappa\rho$$

② 輻射拡散

$$\frac{\partial E}{\partial t} = -\nabla \cdot \boldsymbol{F}$$



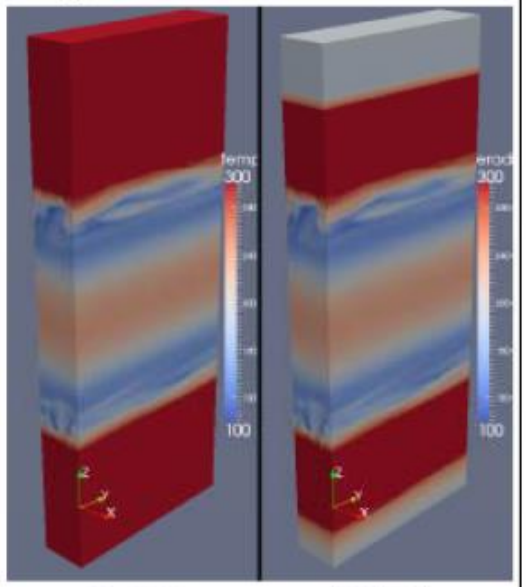
① 輻射拡散

$$\frac{\partial E}{\partial t} = -\nabla \cdot \boldsymbol{F}$$

② 輻射-ガス相互作用

$$\frac{\partial e}{\partial t} = -(4\pi B - cE)\kappa\rho$$

$$\frac{\partial E}{\partial t} = (4\pi B - cE)\kappa\rho$$



（このような分離は昔はよく行われていた。e.g. ZEUS-FLD, Turner & Stone 2001）

原因と対策

Operator splittingしないシステムでは
「ガスエネルギーが輻射エネルギーに変換されて、輸送される」
過程が全て整合的に計算される。
（ホースで繋いで水を流すイメージ）

Operator splittingしたシステムでは
「ガスエネルギーを輻射に渡す」（熱平衡Tg=Tr になる分しか渡せない！）
「輻射エネルギーを輸送する」
のように分離して解くため、非物理的にエネルギー輸送が律速されてしまう。
（コップで一杯ずつ水を汲みだしては流すようなイメージ）

時間刻みを変えたり、演算順序を変えた時に結果が大幅に変わる場合、
そのようなoperator splittingは適切でない。
→時間刻みを十分小さく取るか、ひとまとめのシステムとして解く必要がある。
（輻射-ガス相互作用が十分速いならTg=Trを仮定した方が良い場合もある）

まとめのかわりに

とりとめもなく色々怖い話をしましたが、
「だからシミュレーションは辞めましょう」という話ではありません。

コード・スキームの性質と限界を理解した上で適切に選択し、
シミュレーション結果の物理的妥当性をきちんと確認しましょう

というお話です。そしてもう一つ、論文を書く時は

使用したテクニックを（フロアなど怪しいことまで含めて）
きちんと説明して、再現できるようにしてください。

限界を理解して誠実に議論すれば、rejectされることは稀です。

シミュレーションコードは便利ですが、万能ではありません。
上手に使って皆さんの研究に活かして頂ければと思います。