東京大学 近山・田浦研究室 M1 原健太朗

2009.12.10

発表の流れ

- 課題説明と基本方針
- 前処理と領域分割
- 解法のまとめ
- 分散共有メモリでどこまで性能が出るか

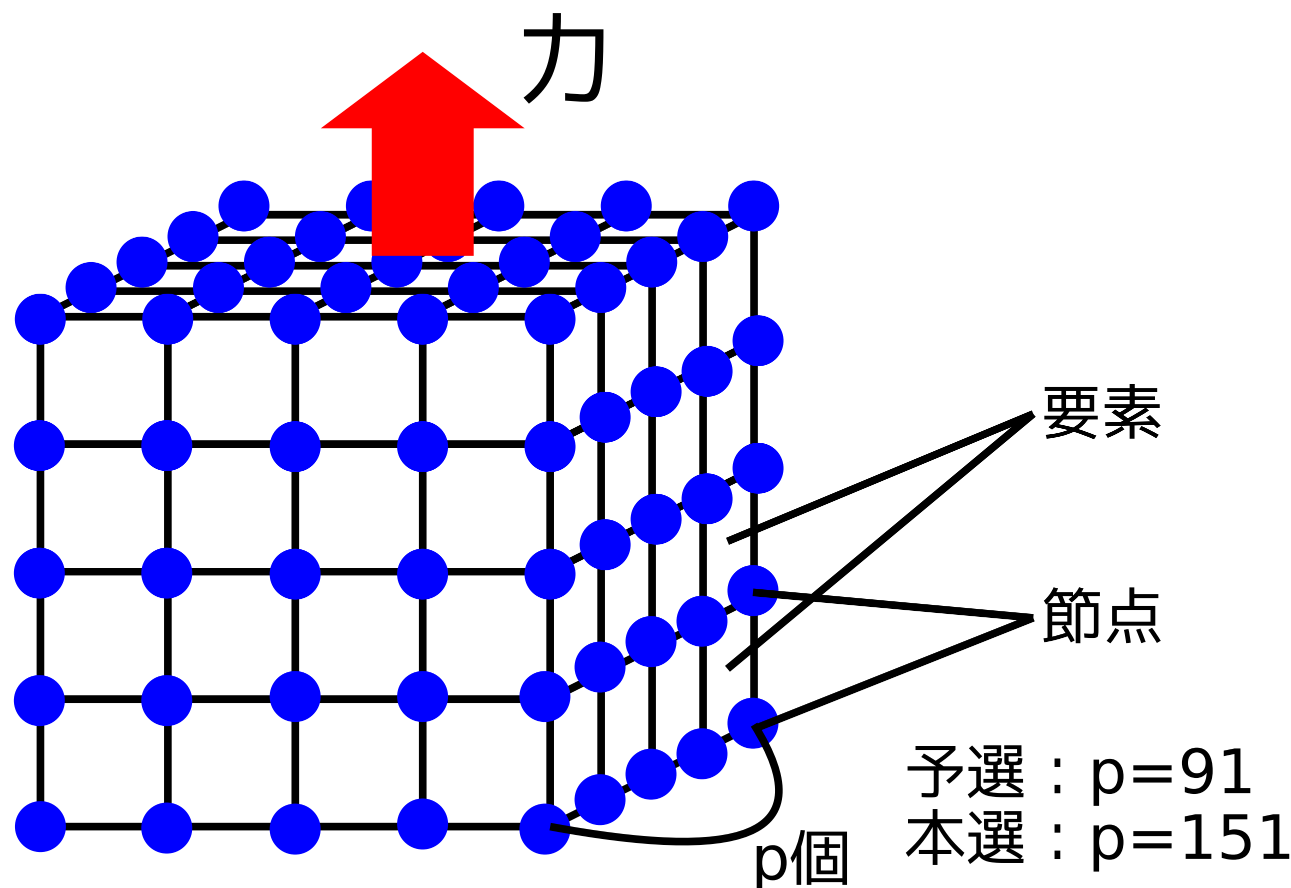
❖ 1. 課題説明と基本方針

コンテスト課題：$Ax = b$ を解く

- **問題：疎行列 A に関して $Ax = b$ を解く**
  - → **予選：未知数は約** 200 **万個，非零成分数は約** 1.8 **億個**
  - → **本選：未知数は約** 1000 **万個，非零成分数は約** 8.3 **億個**
- **実行環境：**
  - → **予選：**T2K **東大の** 8 **ノード** (128PE)
  - → **本選：**T2K **東大の** 32 **ノード** (512PE)

$Ax = b$ の由来：有限要素法

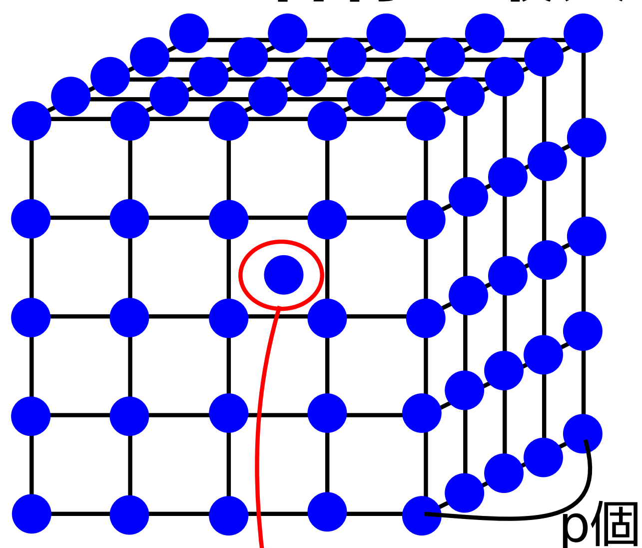
- かなり「性質が悪い」立方体物体に力を加えたときの変形を**有限要素法**で計算
- 節点の個数：
  - → 予選：91^3，本選：151^3

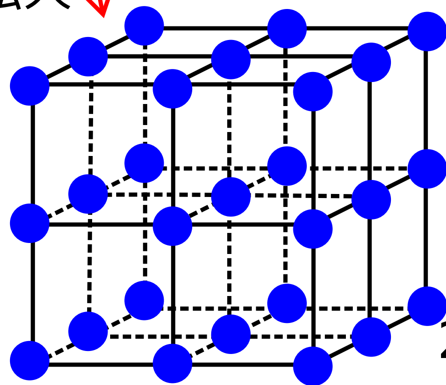


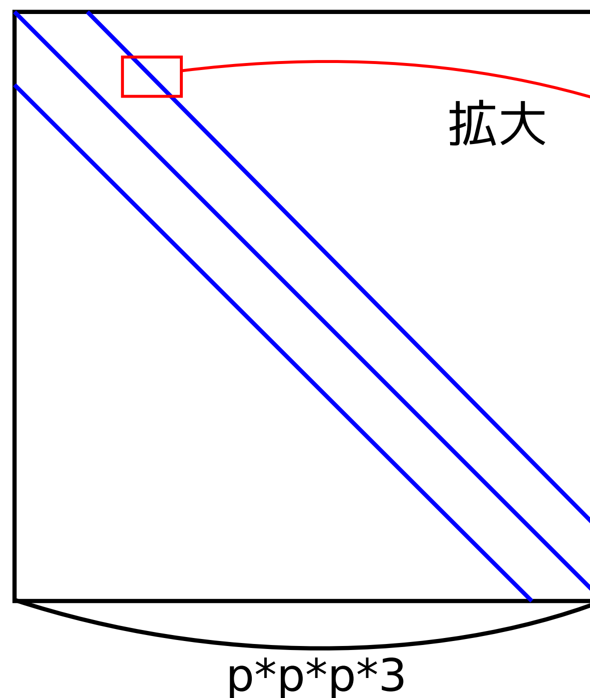
$Ax = b$ の正体

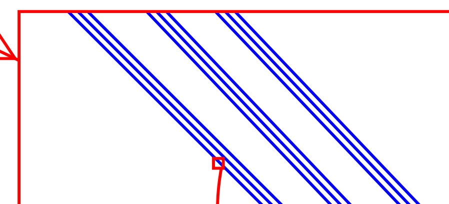
- **未知数 1 個は各節点の各変位 $(x, y, z$ 成分$)$ に対応**
- **疎行列 A は 27 点差分に基づく**

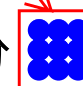
→ **よって各行は最大 81 個の非零成分を持つ**











定石：ブロック化

- **ブロック化**：x, y, z の 3 変位をまとめて (つまり節点単位で) 処理すること
  - → 疎行列 A の 3×3 の 9 成分を「1 成分」と見なして処理
  - → 各種計算に必要なメモリ量を大幅に削減可能

連立方程式をどう解くか

- 直接法：
  - → LU 分解 $+$ 前進後退代入
    - ♦ × 時間的・空間的に不可能
- 反復法：
  - → Jacobi 法，Gauss Sidel 法
    - ♦ × 収束が遅すぎる
  - → CG 系解法
    - ♦ ◎ これを使うが，疎行列 A の「性質が悪い」ため強力な前処理が必須

前処理付 CG 法とは

[Ax = bを解く]

前処理行列Mを求める
x = 0　：解ベクトル（求めたいもの）
p = 0　：修正ベクトル（どの方向にxを修正していくか）
r = b　：残差ベクトル（b-Axのこと）
while (|r|>ε) {
　Mz = rを解く ← 前処理（前進後退代入）
　β = 1/(r,z)
　p = p+βz ← 前処理後の残差ベクトルを基にして解ベクトルを修正する方向を計算
　q = Ap
　α = 1/(p,q)
　x = x + αp ← 解ベクトルを修正
　r = r - αq ← 残差ベクトルを更新
}

並列化すべきアルゴリズム

[Ax = bを解く]

```
前処理行列Mを求める    前処理行列の生成
x = 0
p = 0
r = b
while (|r|>ε) {
    Mz = rを解く       前進後退代入(前処理)
    β = 1/(r,z)        内積
    p = p+βz           DAXPY
    q = Ap             行列ベクトル積
    α = 1/(p,q)        内積
    x = x + αp         DAXPY
    r = r - αq         DAXPY
}
```

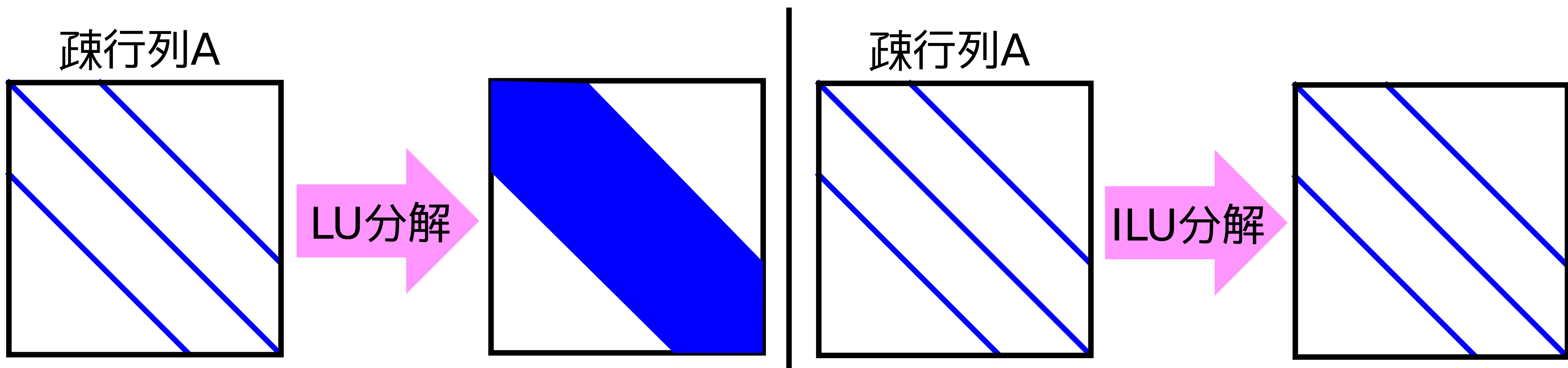
前処理の強力さが収束性を決める

最大のボトルネック

❖ 2.　前処理と領域分割

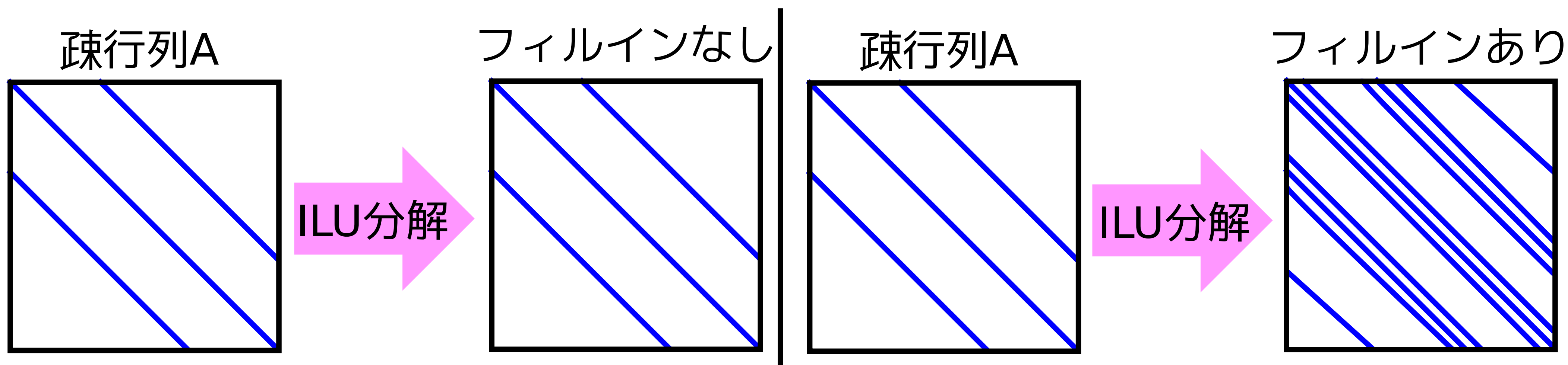
前処理行列 M をどう作るか：ILU 分解

- 前処理の目標：
  - → 疎行列 A に近い前処理行列 M を作って $Mz = r$ を解く
- $M = A$ とし，A の完全 LU 分解によって $Mz = r$ を解くのが究極の理想だが，時間的・空間的に不可能
  - → というか，それができれば最初から $Ax = b$ は解けている
- そこで，A の非零成分の位置だけを考えて「不完全に」LU 分解 (ILU 分解) したものを M とする



ILU 分解におけるフィルイン

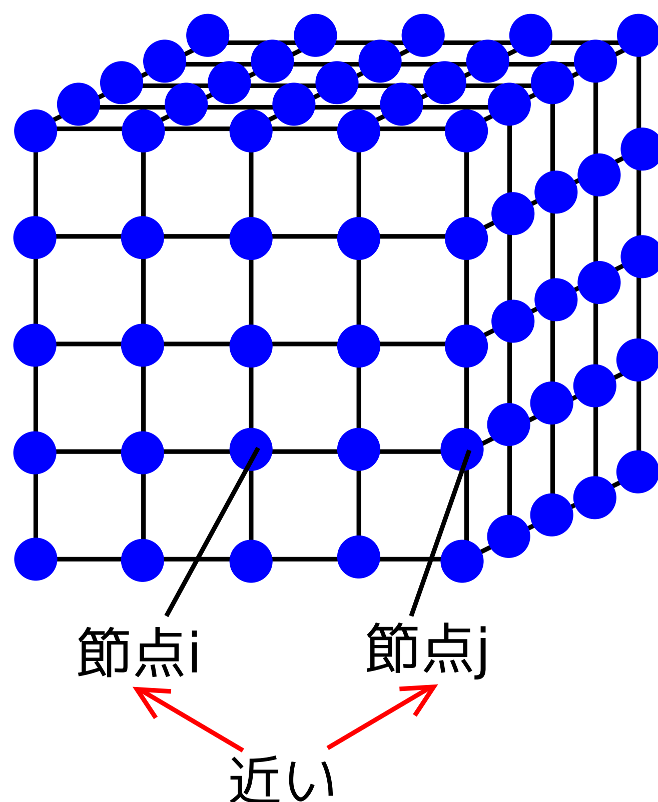
- **しかし，単純な** ILU **分解は「不完全すぎて」前処理として弱い**
- **そこで，** A **の非零成分以外の位置にも「ちょっとだけ」値を入れる (フィルイン)**

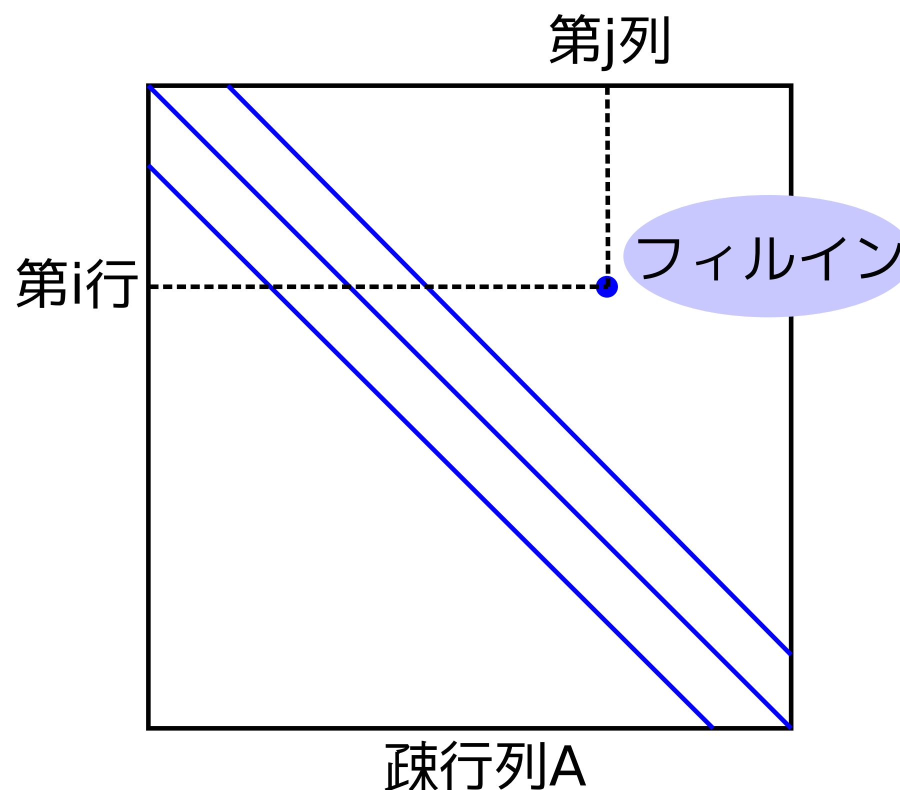


どの位置にフィルインさせるか

- 節点 i と節点 j が「近く」にあれば，第 i 行の第 j 列にフィルイン
- どのくらい「近く」まで考えるかの指標が**フィルインレベル**

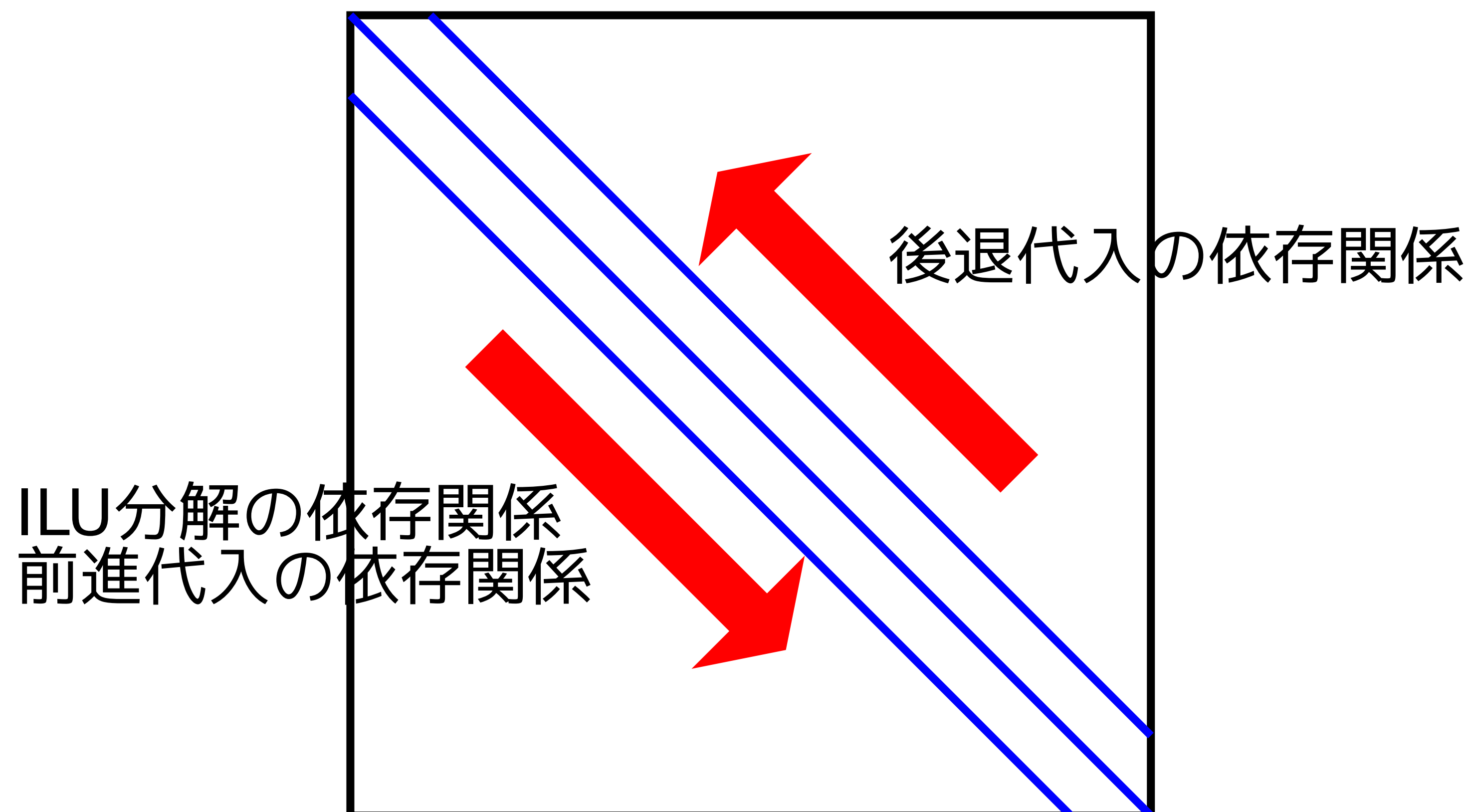
→ 予選：フィルインレベル 3，本選：フィルインレベル 4





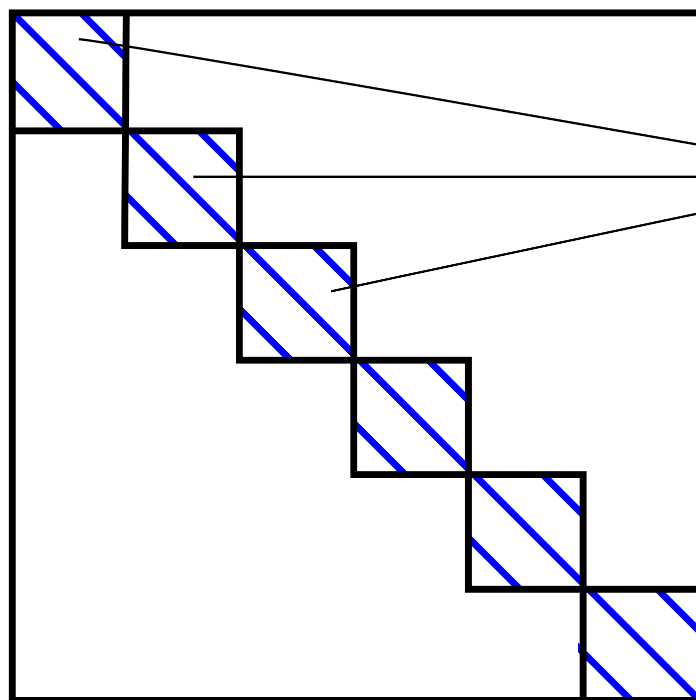
ILU 分解をどう並列化するか

- **以上の前処理行列 M の作り方では，ILU 分解と (反復処理における最大のボトルネックの) 前進後退代入 $Mz = r$ が並列化できない**
  - → **疎行列は各行各列の成分数が少ないので並列性を抽出しづらい**



ブロック ILU 分解

- **ブロック ILU 分解**：行列を PE 数個分の正方形対角「ブロック」に分割し，それ以外の成分は零とする (=成分を「無視」する)
  - → ILU 分解も前進後退代入も，PE 間で完全に独立
- 強力な前処理を行うには，できるだけ「無視」が少ないように「ブロック」を構成することが重要



「ブロック」の物理的意味：領域分割

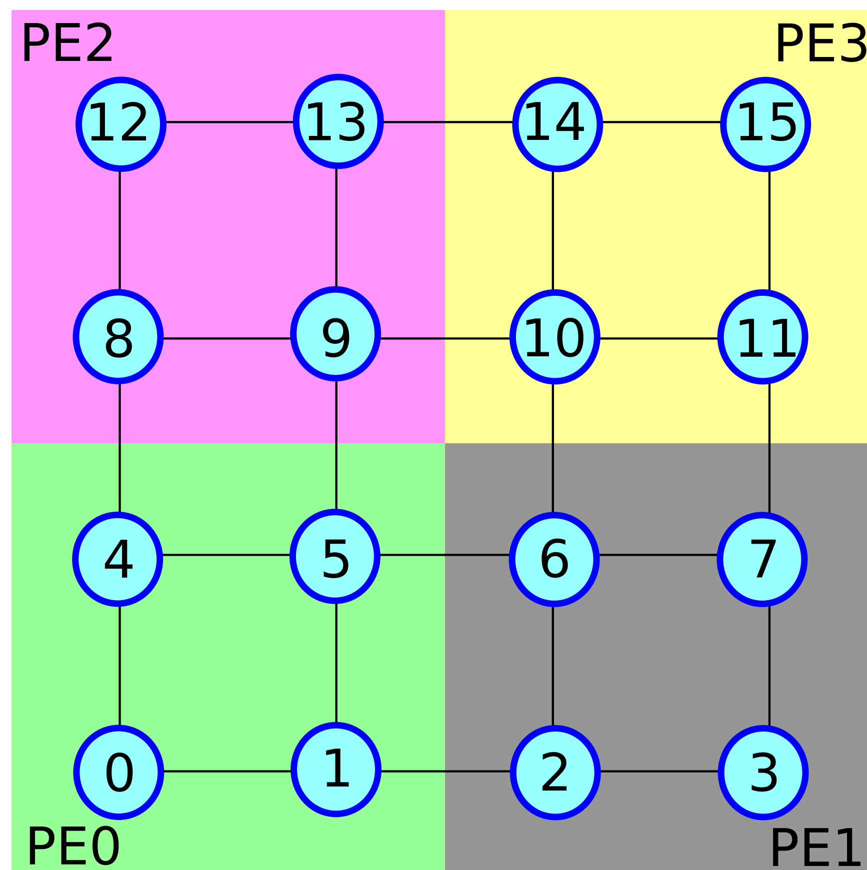
- M における「ブロック」：立方体物体において各 PE が担当する節点集合に対応
- M をどう「ブロック」に分けるか＝全節点をどう**領域分割** (＝**データ分散**) するか

→ 以降，どういう領域分割が効率的かを考えていく

領域分割

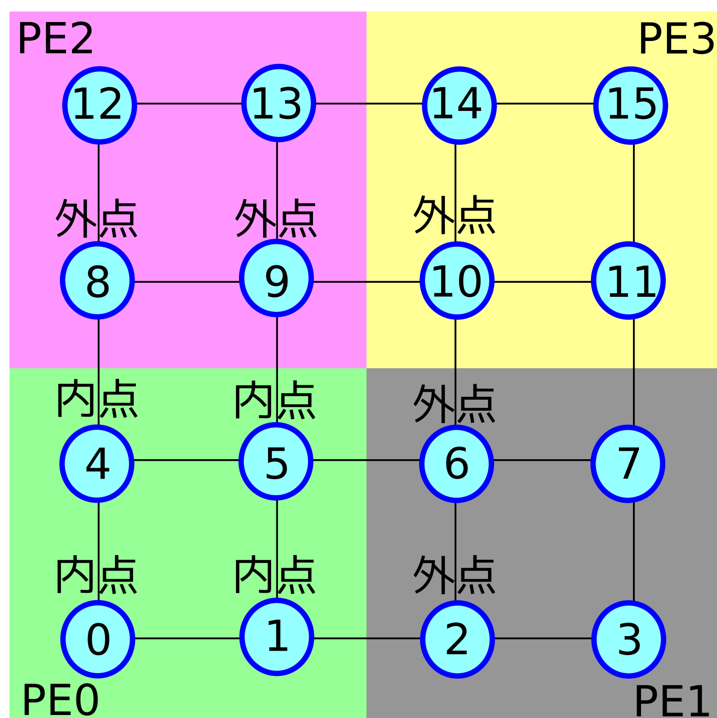
領域分割のモデルを簡単化

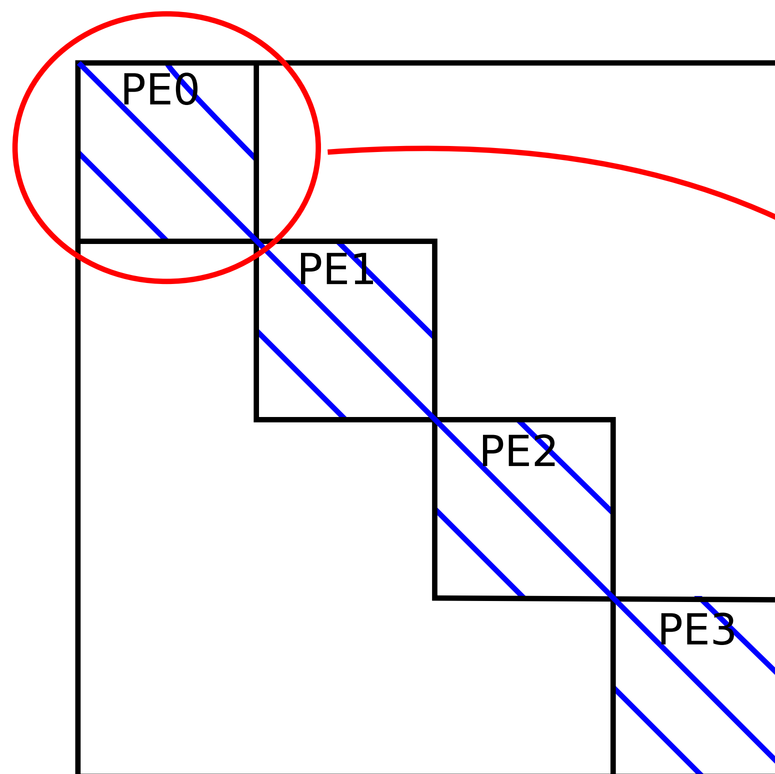
- 3 次元立方体だと面倒なので，以下のモデルを考える：
  - → 4×4 の 2 次元領域
  - → 4PE で領域分割
  - → PE0 の挙動に着眼

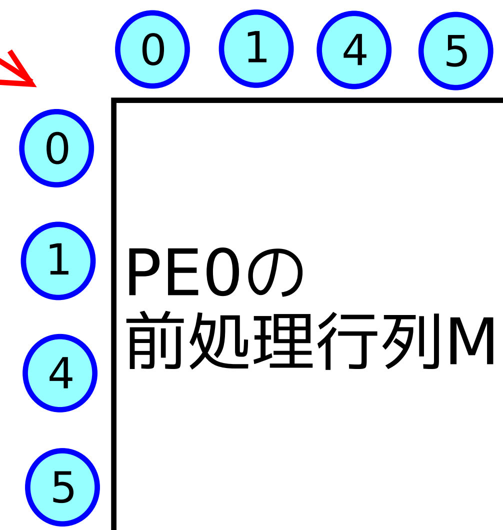


ブロック ILU 分解の物理的意味

- ブロック ILU 分解の物理的意味：
  - → **外点**の影響を完全に無視し**内点**だけを考えて前処理することに相当

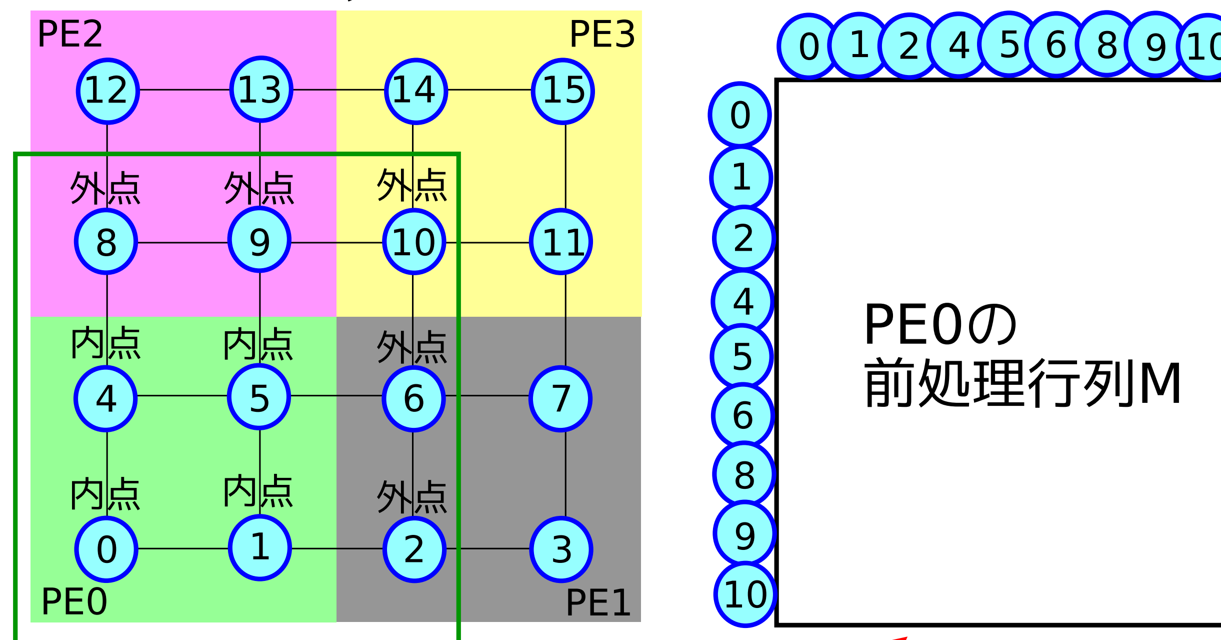






領域間オーバーラップ

- **領域間オーバーラップ：**
  - → 「近く」にある外点の影響も考慮して前処理すること
- 各 PE は内点に関する r しか知らないのに，どう $Mz = r$ を解くか?
  - → Additive Schwarz 法，Restricted Additive Schwarz 法，…

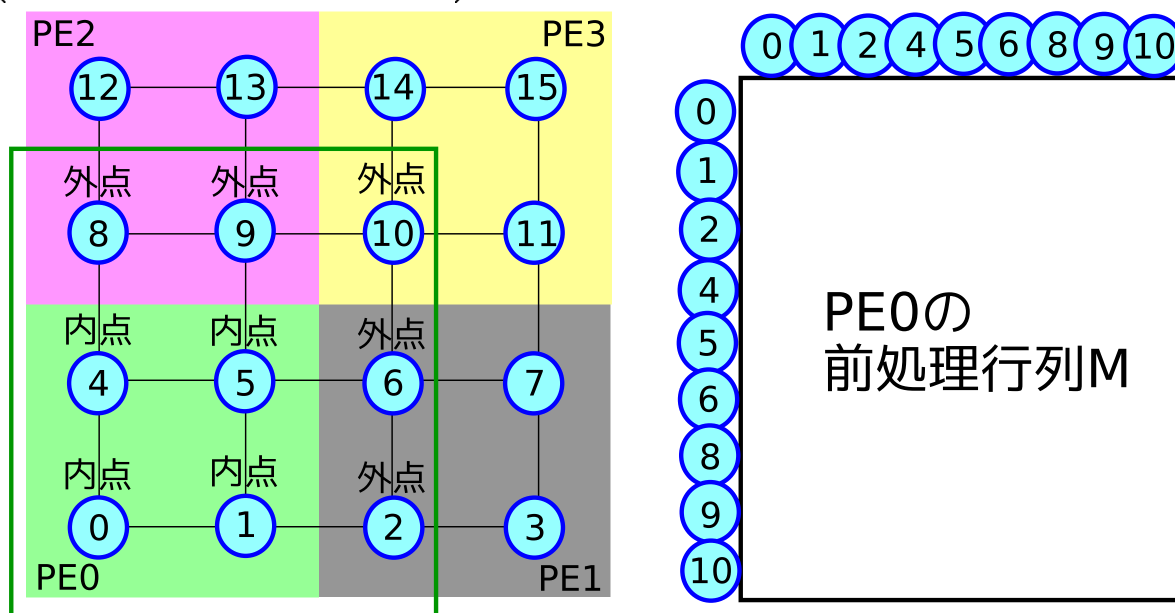


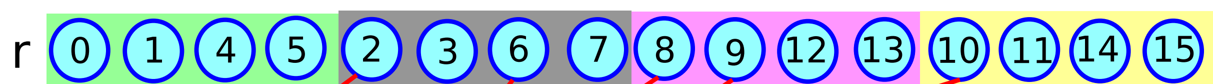
深さ l の Restricted Additive Schwarz 法 (1)

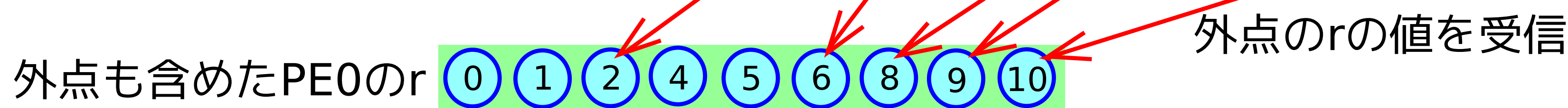
(1) **周囲 l までの外点を考えて M を作る**

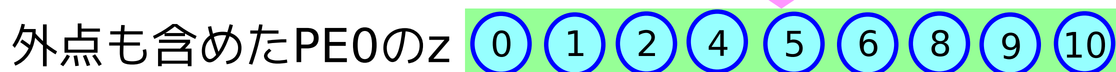
(2) **考えている外点に対応する r の値を受信**

(3) $Mz = r$ **を解く (前進後退代入)**



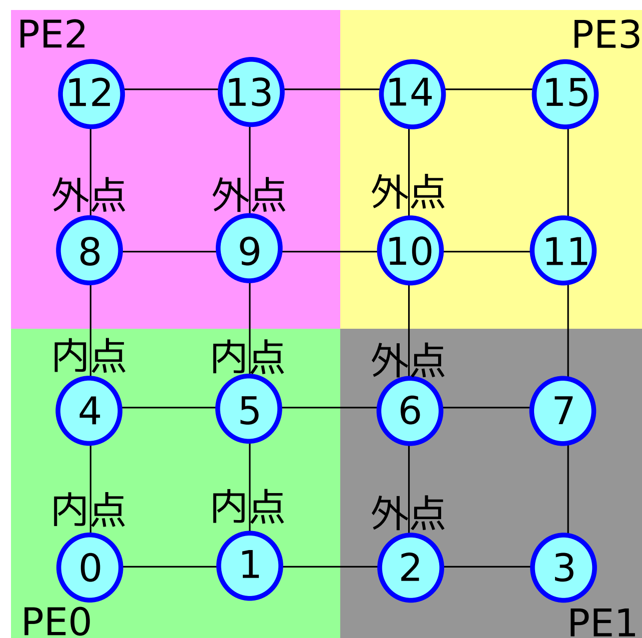


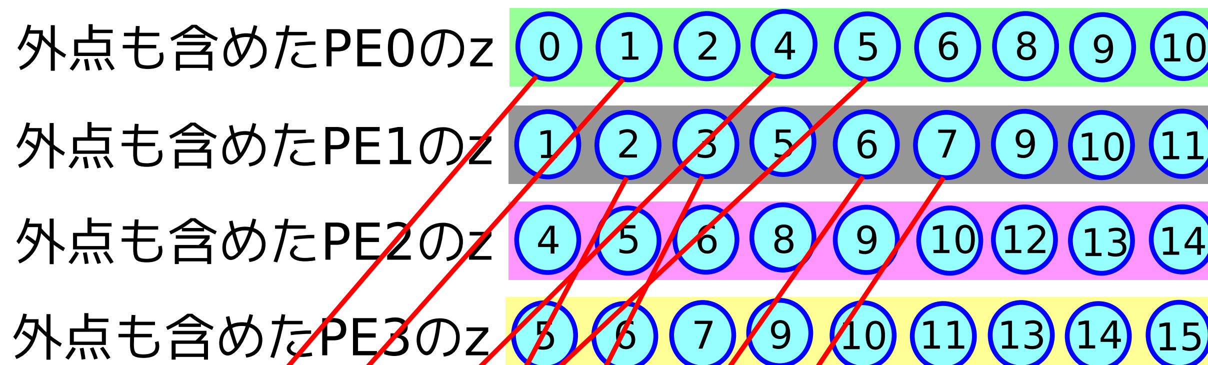


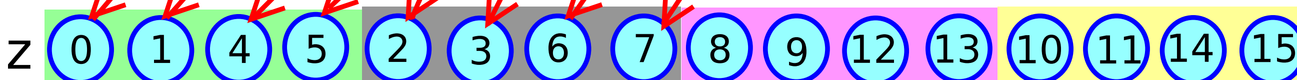


深さ l の Restricted Additive Schwarz 法 (2)

(4) 外点に関して出てきた z の値は捨てる



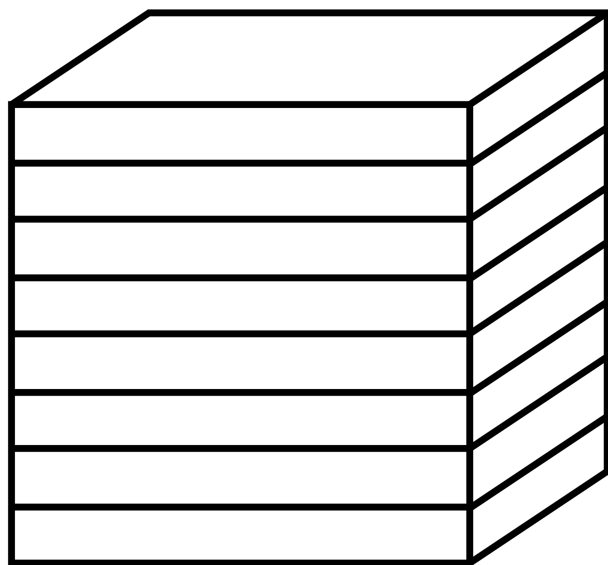




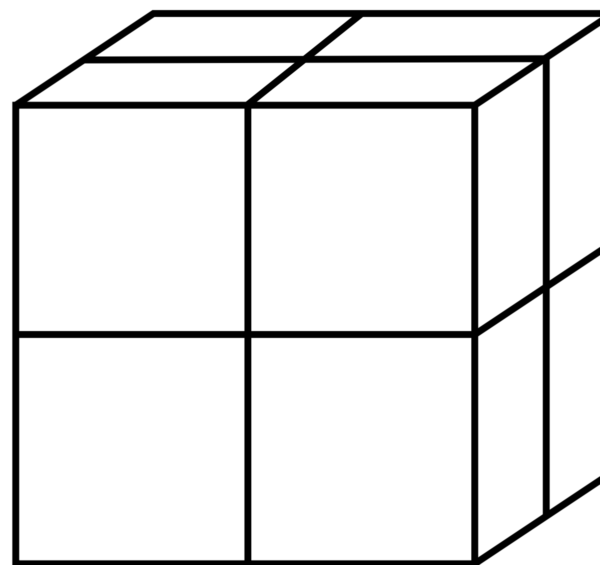
➤ 予選：深さ $l = 3$，本選：深さ $l = 4$

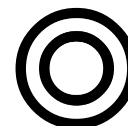
では，良い領域分割とは何なのか

- 外点が少ない（＝各領域が立方体に近い）ことが重要
  - → 内点/外点の比率が高いほど領域間オーバーラップの効果が強力なため
  - → 通信量が減るため
- 領域間の負荷均衡のために微調整できれば，なお良い



×



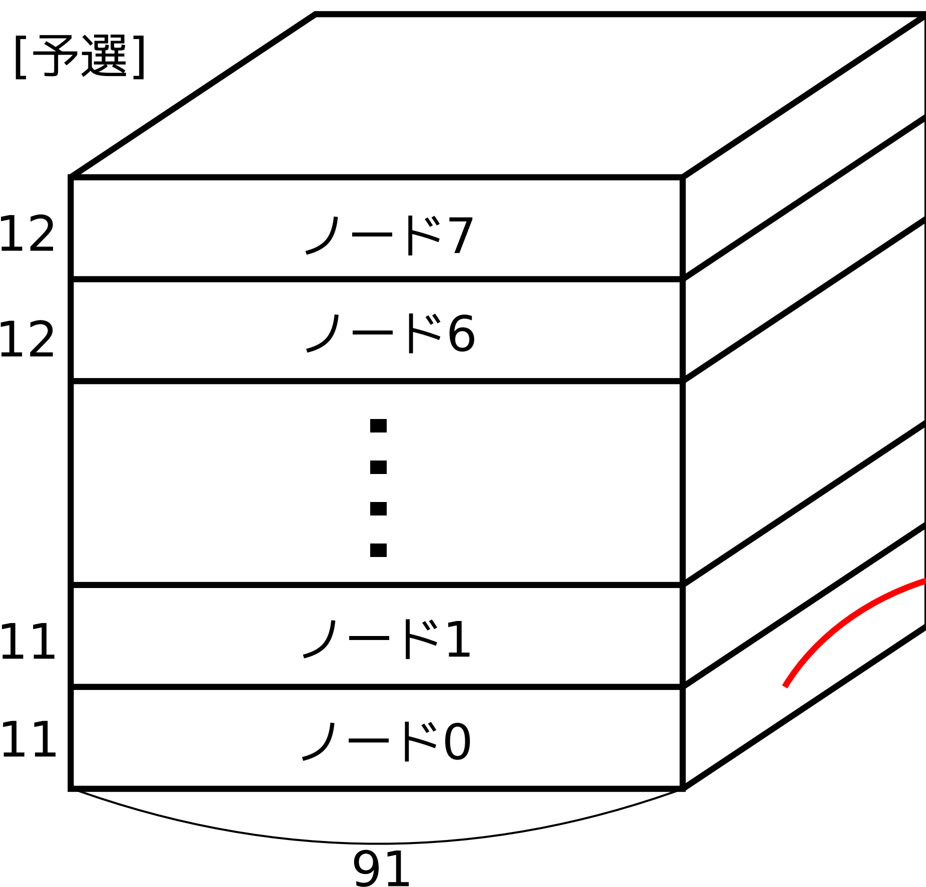


私の勘違い

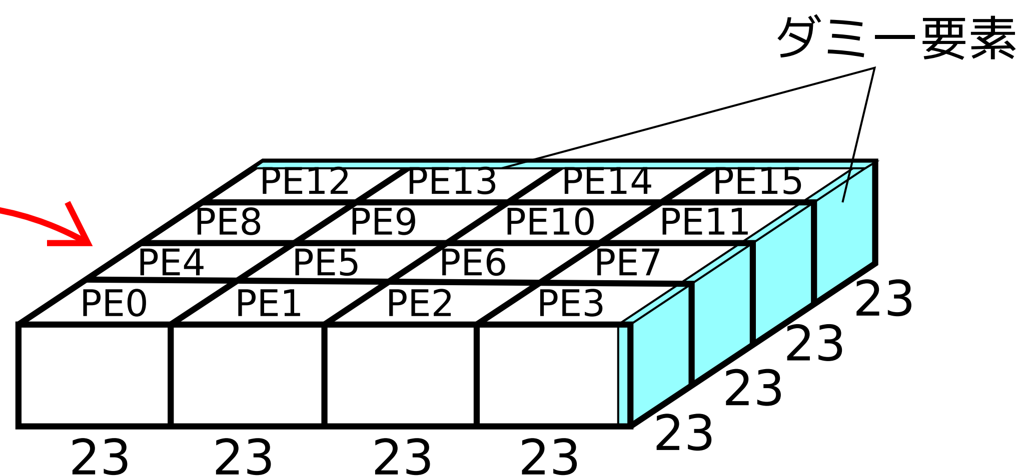
- ところが，私はこう誤解していた：
  - 外点の少なさは通信量の削減のみに効いて，前処理の強さには影響しない
  - だとすれば，外点を少なくして通信量を減らすために複雑な領域分割を行うよりも，通信量は少し多くても，通信形態やメモリアクセスを単純化できるような領域分割の方が高速なはず

私の領域分割 (1)

- MPI 8 プロセス ×16 pthread で**ハイブリッド並列化**
- x, y 方向の端に「ダミー節点」を追加し，x, y 方向の領域の大きさを均等化



8ノード×16スレッド



私の領域分割 (2)

- 特長：**オーバーヘッドは非常に小さい**
  - → x, y 方向の外点の処理は**共有メモリ上の処理で完結**
  - → 通信が生じるのは隣接ノードとだけ
  - → 分割面が非常に単純で規則的なので，複雑なメモリアクセスや通信時のメモリコピーなどが不要

[予選]

12	ノード7
12	ノード6
	⋮
11	ノード1
11	ノード0

91

8ノード×16スレッド

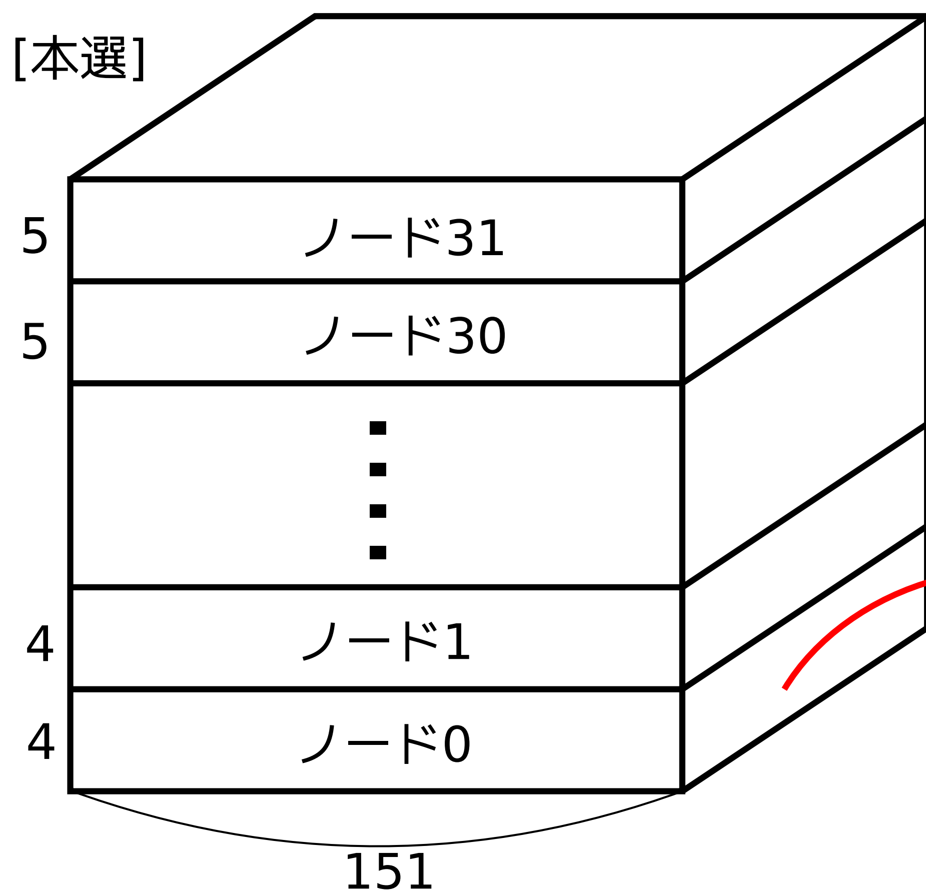
ダミー要素

PE12	PE13	PE14	PE15
PE8	PE9	PE10	PE11
PE4	PE5	PE6	PE7
PE0	PE1	PE2	PE3
23	23	23	23

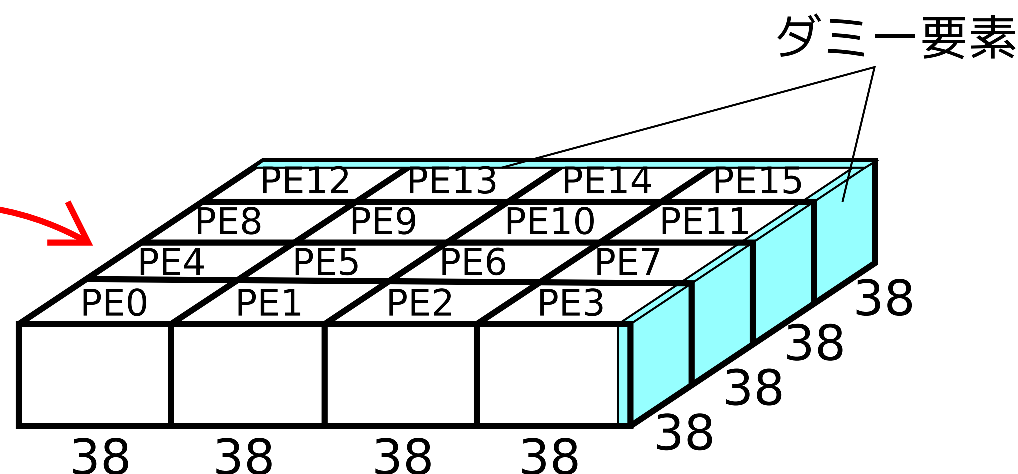
奥行き方向：23, 23, 23, 23

そして，本選問題で困った

- 予選：128PE で 91^3 節点，本選：512PE で 151^3 節点
- 本選だと領域が「薄く」なりすぎるため，前処理が弱すぎて収束しない恐れ
- 352PE が限界だと試算し，352PE で実行した



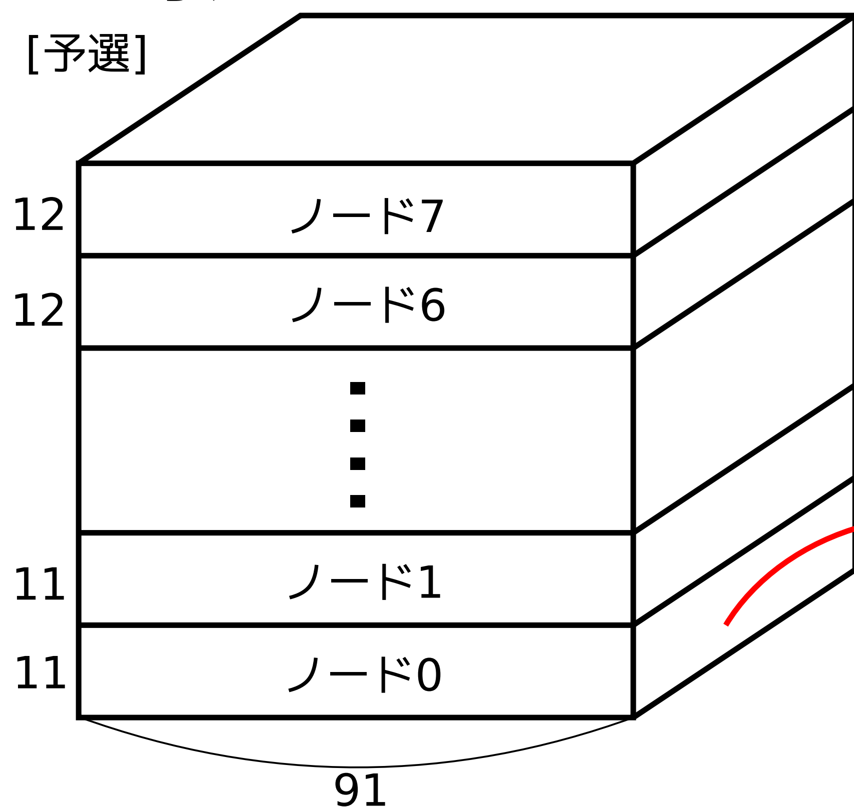
32ノード×16スレッド???



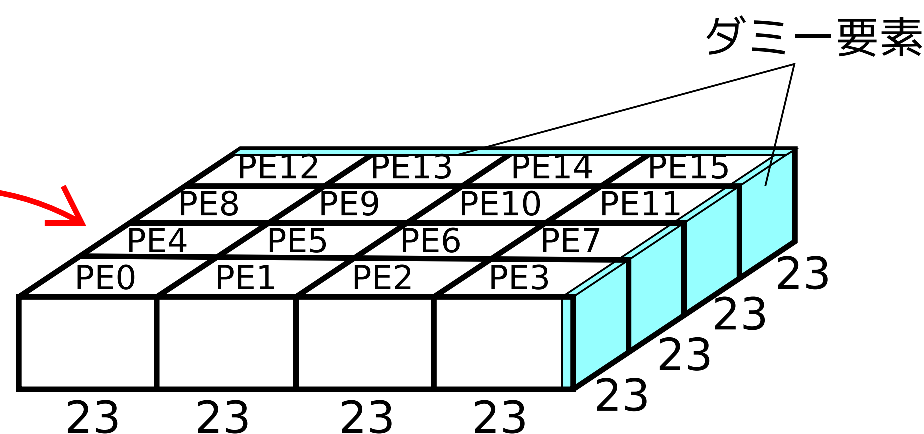
❖ 3. 解法のまとめ

主な解法のまとめ

- フィルイン付きブロック ILU 分解で前処理行列を構成
- MPI と pthread でハイブリッド並列化
- オーバーヘッドが非常に小さい領域分割
- Restricted Additive Schwarz 法に基づき領域間オーバーラップを考慮



8ノード×16スレッド



その他の工夫 (詳細略)

- 反復法として (CG 法ではなく)BiCGSTAB 法を利用
  - → Restricted Additive Schwarz 法が前処理行列 M の対称性を崩すため
  - → 収束安定化のために初期シャドウ残差ベクトルを工夫
- NUMA の First Touch
- double 型と float 型の使い分け
- キャッシュを意識したデータ構造
- 適度なループアンローリング
- コンパイラは gcc -O3 を使用

結果：本選問題における収束性

➤ 収束が非常に不安定

residual= r / b

1e+02
1e+00
1e-02
1e-04
1e-06
1e-08
1e-10
1e-12
1e-14

0 500 1000 1500 2000 2500 3000

# of iterations

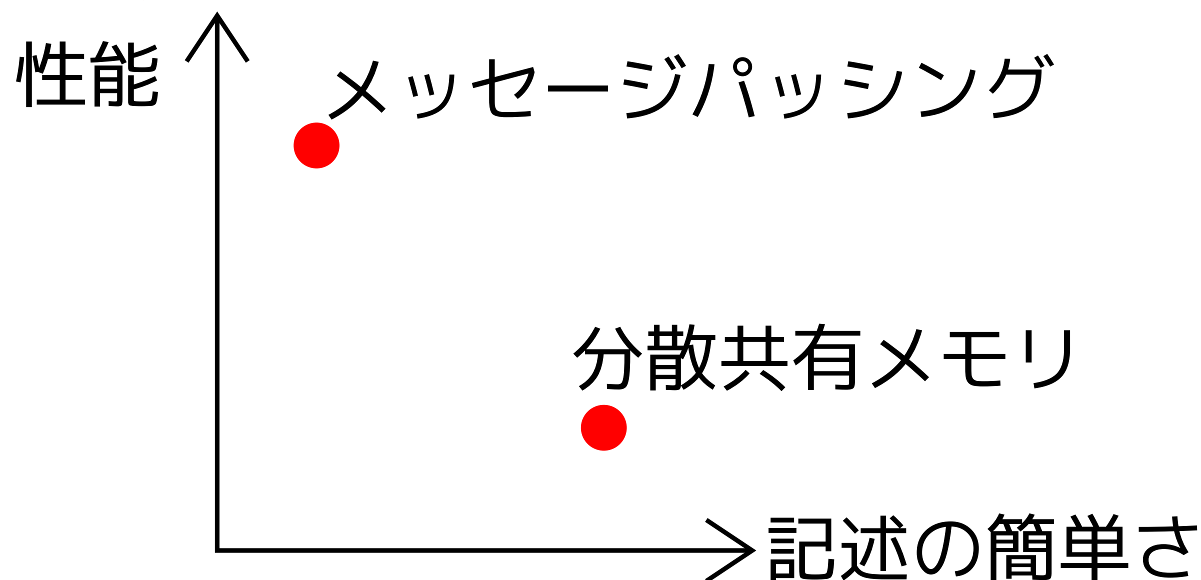
敗因

- 領域分割が不適切だった
  - → オーバーヘッドの小ささだけを意識して，前処理としての強力さを考慮しなかった
  - → 有限要素法としてのアルゴリズムよりも，細かい実装上の最適化に走りすぎた
- (今回の問題では)BiCGSTAB 法の収束性が悪かった
  - → BiCGSafe 法の方が安定する
- 節点のオーダリングを考慮しなかった (詳細は中島君の発表)

❖ 4. 分散共有メモリでどこまで性能が出るか

動機付け

- 有限要素法の通信形態は非定型で複雑
  - → メッセージパッシングで記述するのは大変
  - → **分散共有メモリで見通し良く記述**できたらうれしい
- しかし，容易にプログラムするだけでは分散共有メモリはまず性能が出ない
  - → では，**徹底的に最適化したら分散共有メモリでどこまでMPIに近づけるか**?



DMI：Distributed Memory Interface

特長 1： ノードの動的な参加/脱退に対応した分散共有メモリ

→ ノードの増減を越えて並列計算を継続可能

特長 2： 既存の共有メモリプログラミングとの乖離が少ない

→ pthread プログラムからのほぼ機械的な変換作業で DMI のコードが得られる

特長 3： (OS のメモリ保護機構を利用せずに) ユーザレベルでコンシステンシ管理を行うことで，柔軟な通信チューニングのための手段を数多く提供

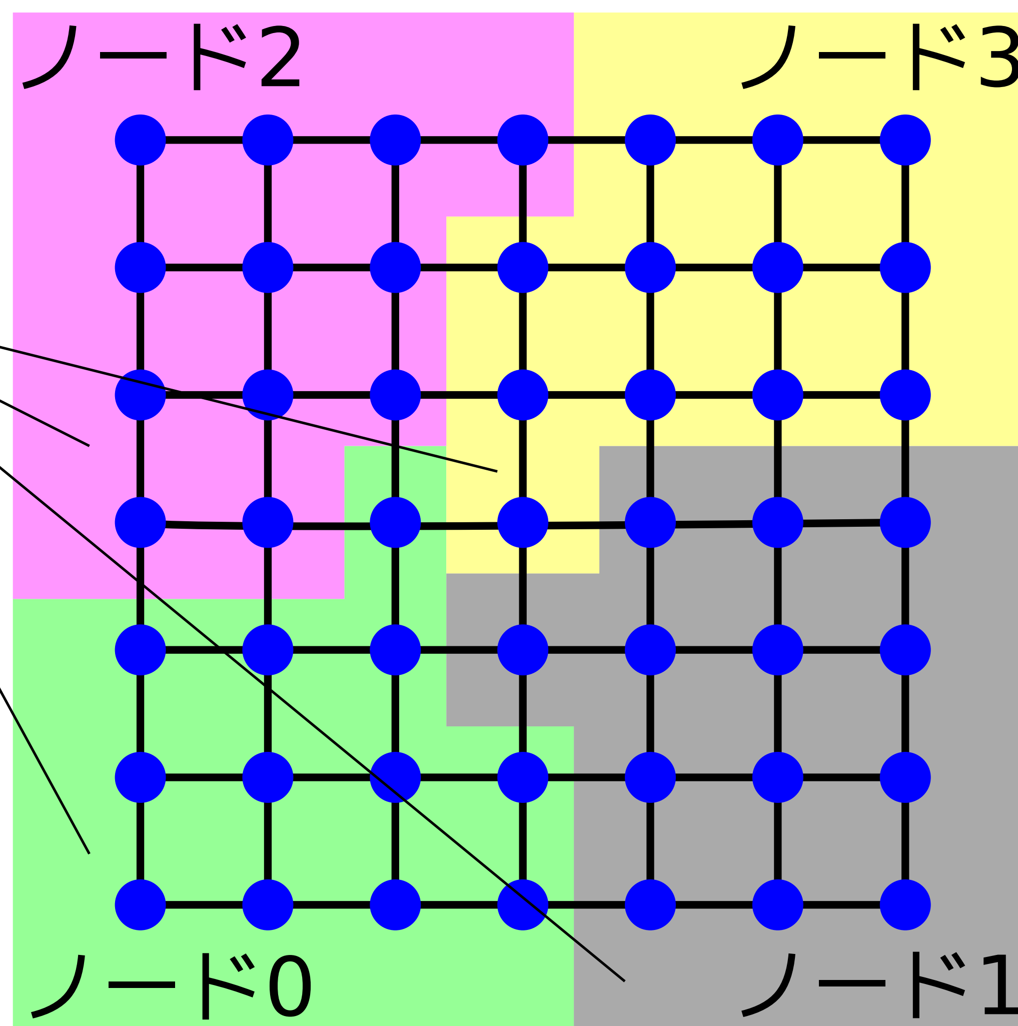
➤ 原健太朗，田浦健次朗，近山隆．計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース．情報処理学会論文誌（プログラミング）．2009/10

任意粒度でのコンシステンシ維持

- (OS のページサイズに関係なく) コンシステンシ維持の粒度を任意に設定可能

コンシステンシ粒度

領域分割の各領域をコンシステンシ維持の粒度に設定できる

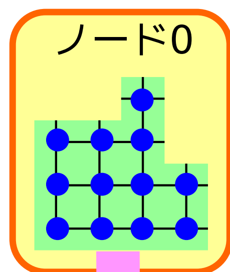


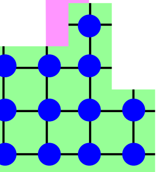
マルチモード read/write

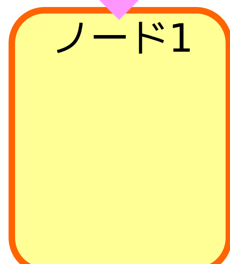
- read/write の粒度でデータをどう移動させるかを指示可能
  → 特に PUT/GET が可能

readの選択肢

オーナーノード

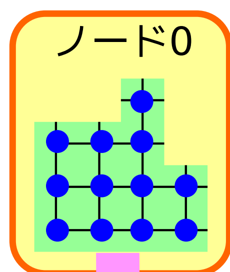


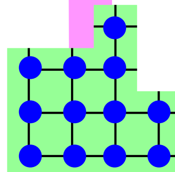


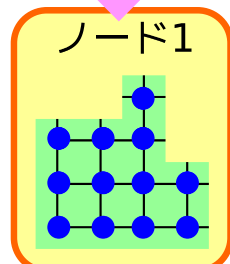


今の1回だけ
read (GET)

オーナーノード



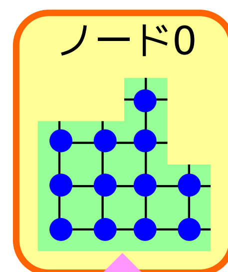


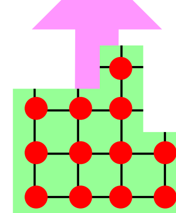


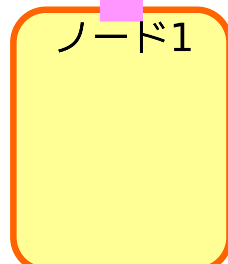
readした後
キャッシュする
(invalidate型
or update型)

writeの選択肢

オーナーノード

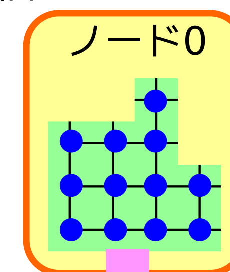


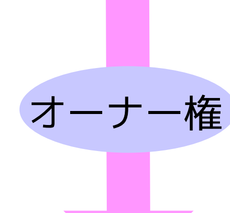




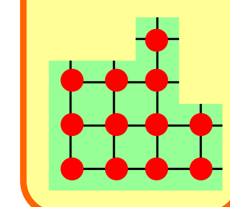
オーナーにデータ
を送ってwrite
してもらう(PUT)

旧オーナーノード





新オーナーノード

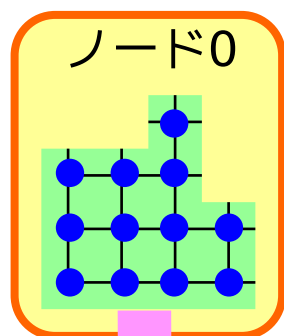


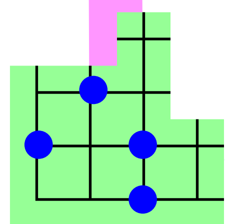
オーナー権を
奪った後で
自分でwriteする

離散アクセスのグルーピング

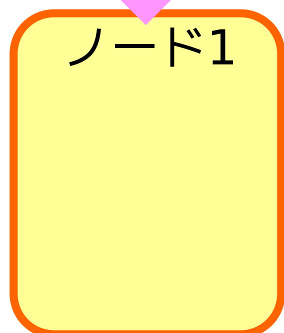
- コンシステンシ維持の粒度に関係なく，**必要な部分だけ一括して PUT/GET** できる

オーナーノード

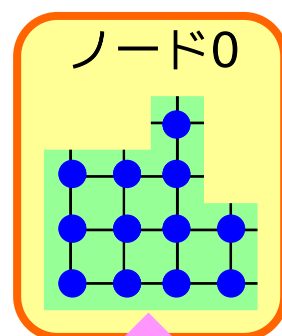


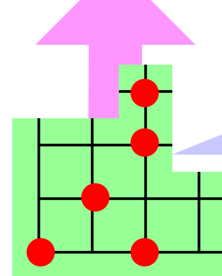


「ここ」と「そこ」だけreadする

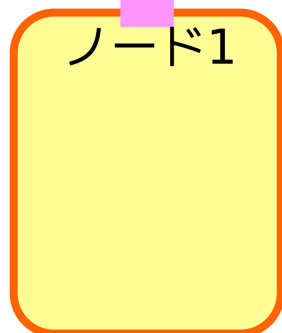


オーナーノード





「ここ」と「そこ」だけwriteする



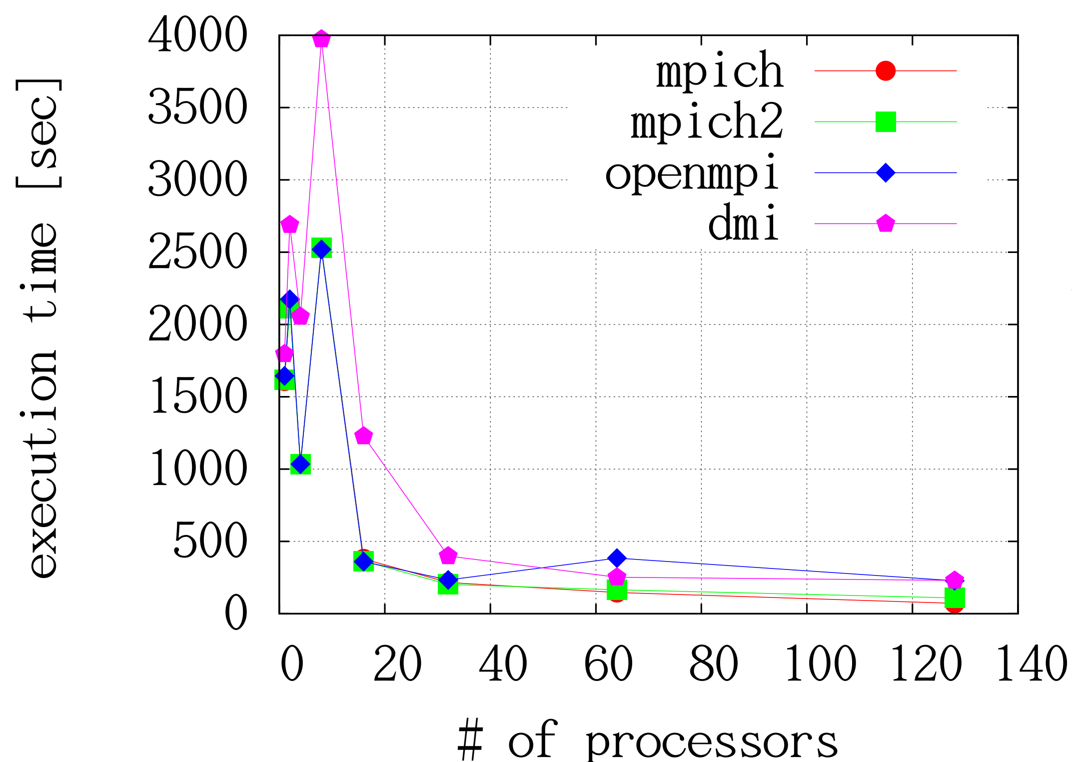
ハイブリッド並列

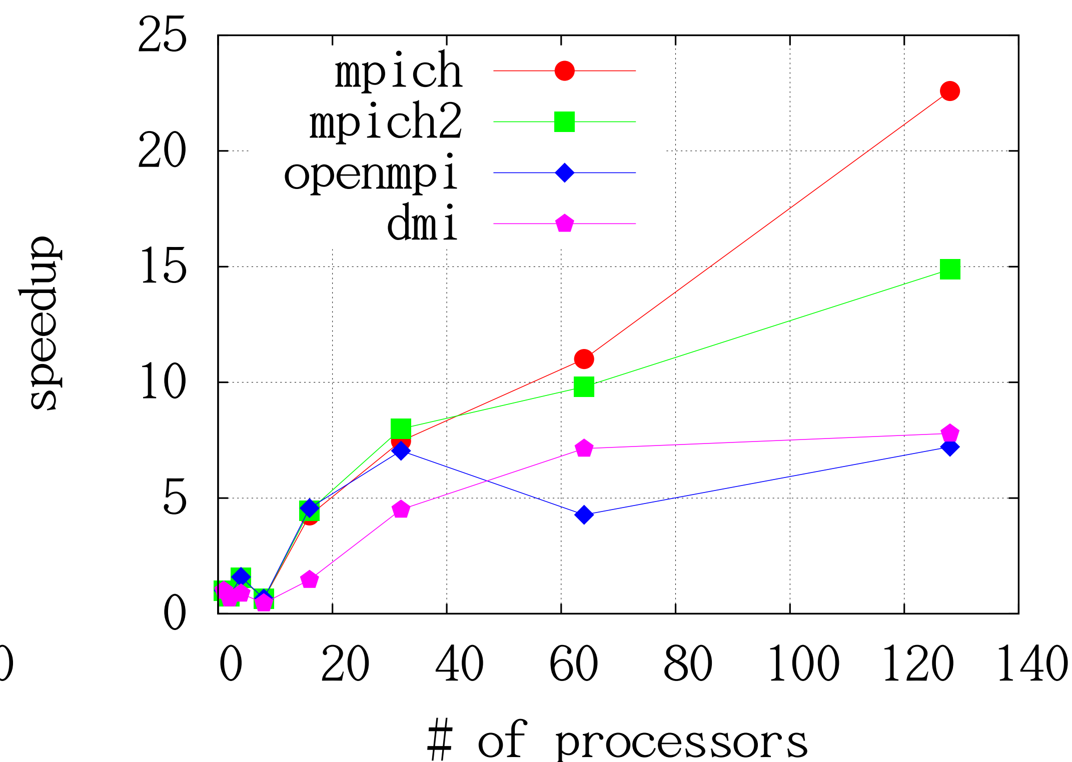
- **処理系が自動的にハイブリッド並列化してくれる**
  - → **ノード内は** pthread **並列**

実験環境

- 以上の最適化手段を駆使した DMI プログラムと各種 MPI を比較
- 予選問題を使用
- 1 位の中島君のプログラムを改変して利用
- コモディティクラスタ：InTrigger の kyutech クラスタ
  - Xeon 2.33GHz，8 コア ×16 ノード
  - 1GbitE×2
- スパコン：T2K 東大
  - Opteron 2.3GHz，16 コア ×8 ノード
  - MPI では Myrinet-10G×4 を使用
  - DMI では 1GbitE を使用

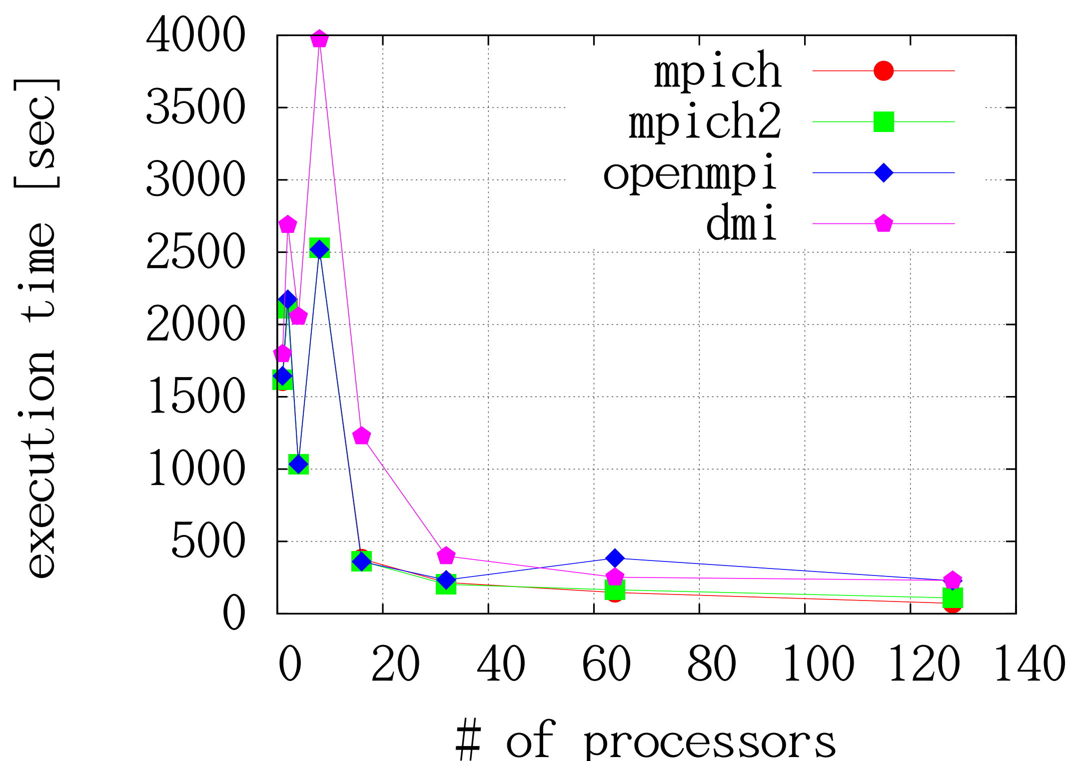
実験結果＋考察：kyutech(1)

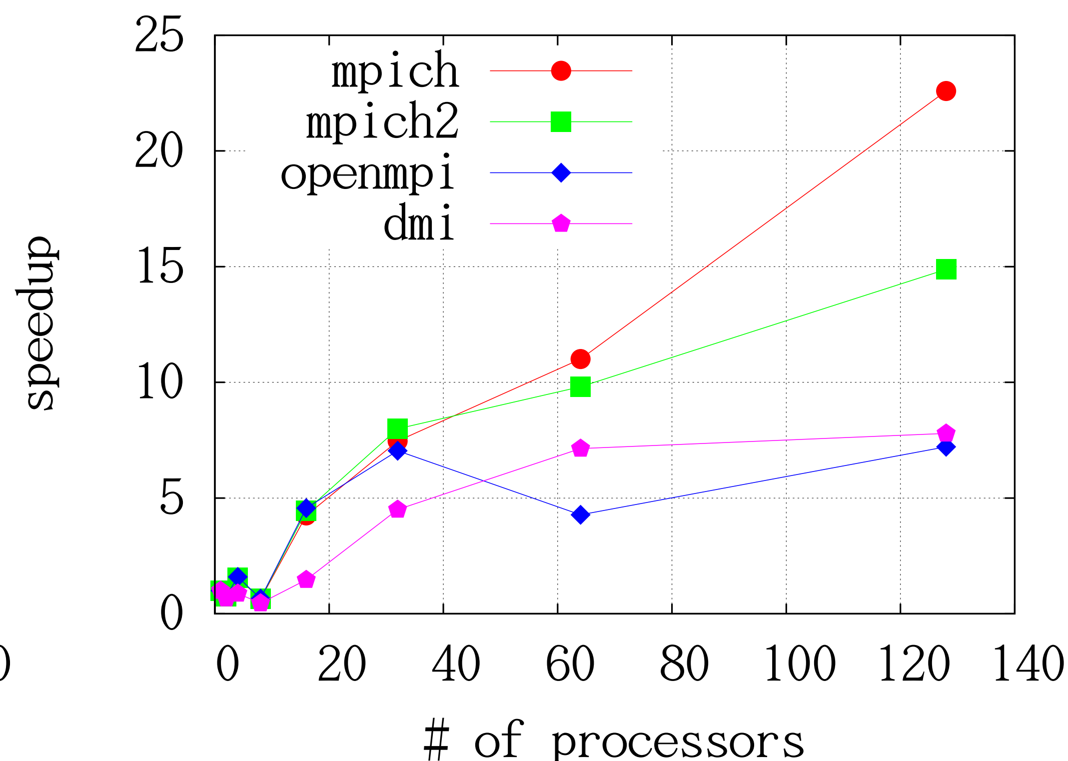




- 128PE 時の速度向上度：
  - ➔ mpich>mpich2>dmi≈openmpi
- ただし mpich は非同期通信などにバグあり

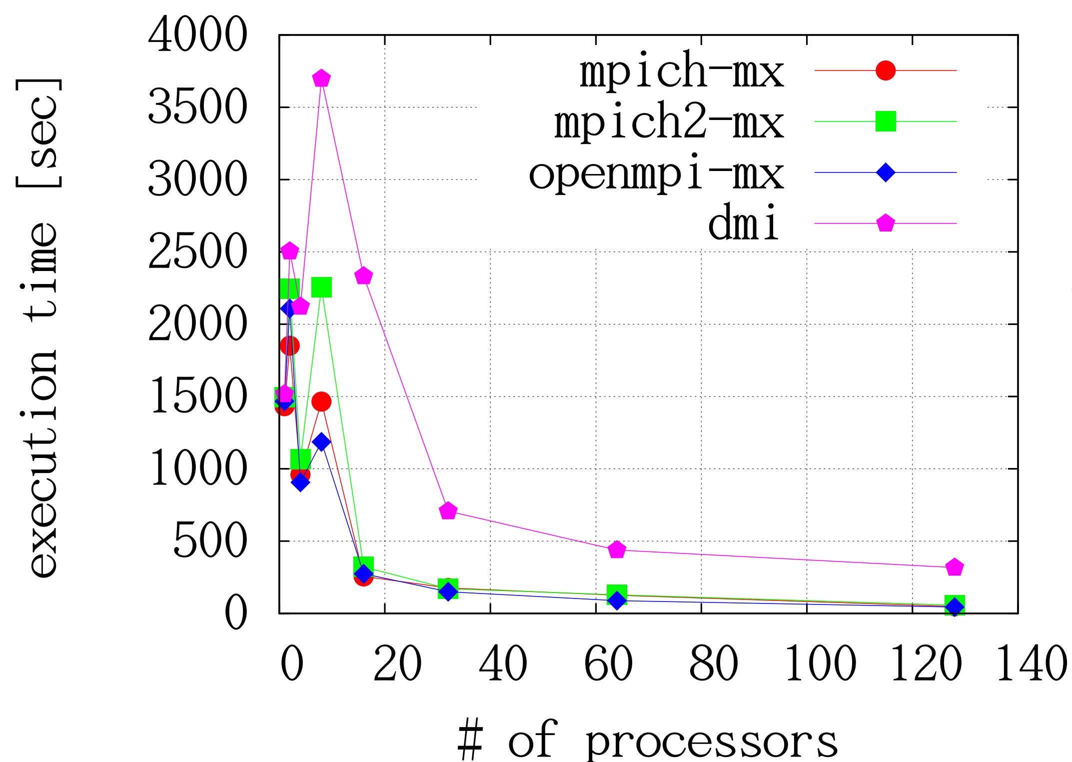
実験結果＋考察：kyutech(2)

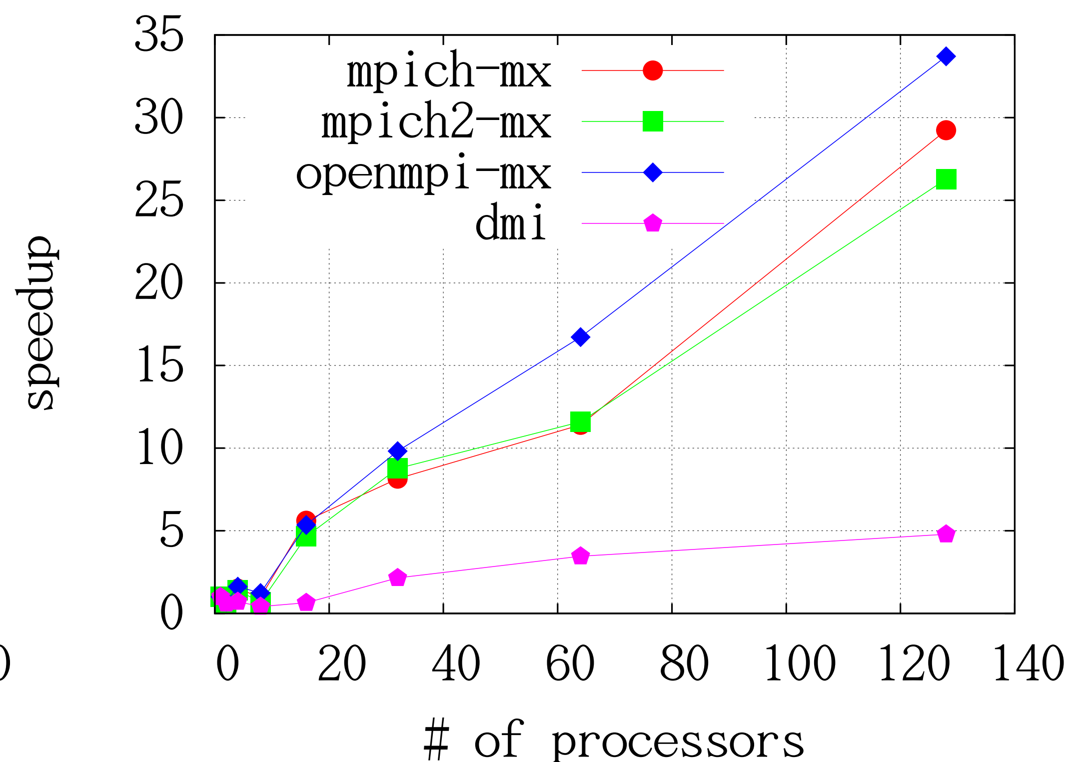




- (一般的に分散共有メモリは性能が出にくいが) **DMI で徹底的に最適化すれば 128PE で OpenMPI 並の性能が出せた**
- スケーラビリティの曲線が揺れるのは，BiCGSTAB 法の収束性が (その PE 数における) 領域分割に大きく左右されるため

実験結果 + 考察：T2K 東大





- 128PE 時の速度向上度：

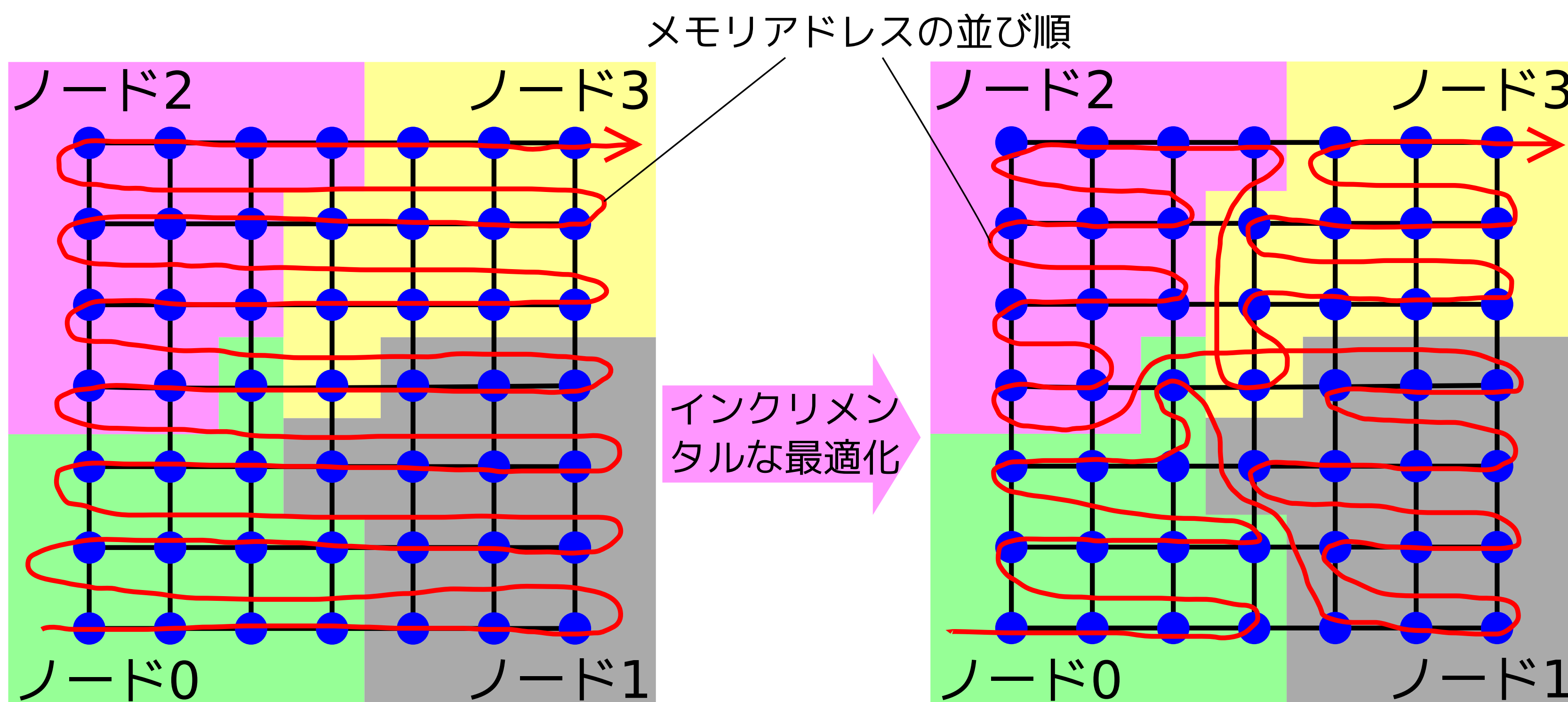
  → openmpi-mx>mpich-mx>mpich2-mx≫dmi

- やはり Myrinet(用の MPI) は速い!

今回わかったこと：DMI のインパクト

- 実験結果：
  - → 簡単なコードを書いたら当然遅かったが，
  - → 徹底的に最適化したところ，
  - → 結果的に MPI 並に煩雑なコードになってしまったものの，
  - → kyutech では **128PE で OpenMPI と同等の性能**が出た
- DMI のインパクト：**プログラム開発の見通しが良い**
  - → read/write ベースで記述できるので，**「とりあえず動く」プログラムを書くのは MPI よりもはるかに簡単**
  - → 柔軟な通信チューニングの API により，**インクリメンタルに最適化**可能

一般の分散共有メモリに対して言えること

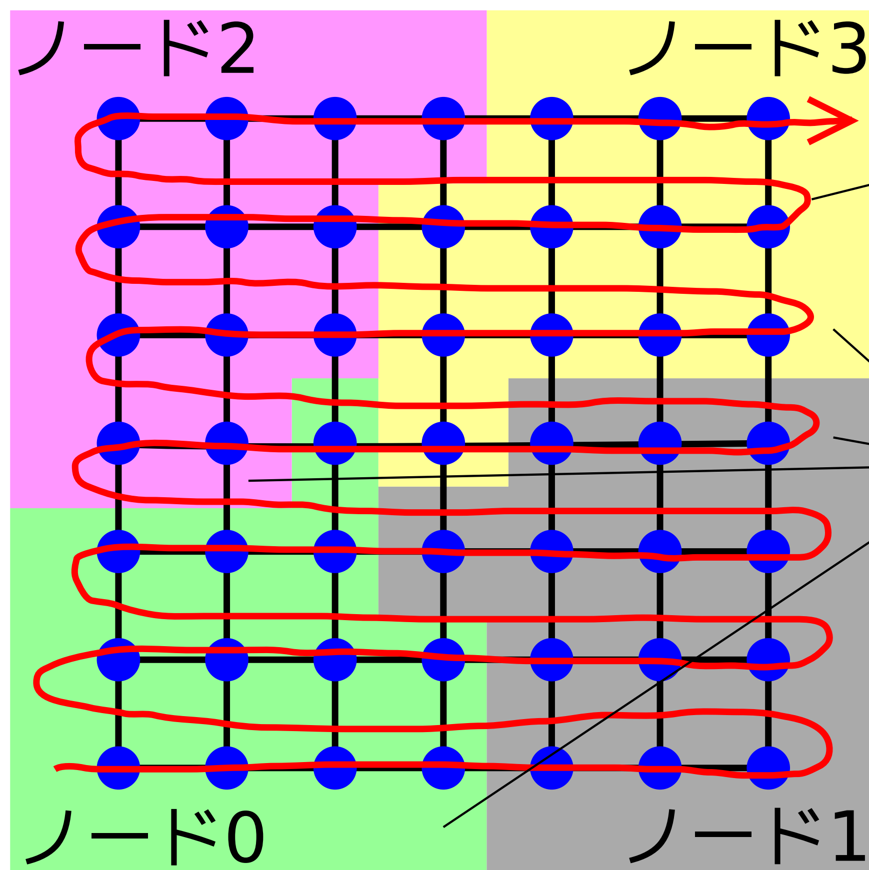


記述は簡単で「とりあえず動く」が，コンシステンシ維持の粒度をどう選んでも性能が出ない

記述は複雑だが，データの分散配置とコンシステンシ維持の粒度が一致するので性能が出る

- **記述の簡単さを決めるのは，（プログラマから見える）メモリアドレスの並び順が規則的かどうか**
- **性能を決めるのは，データの分散配置とコンシステンシ維持の粒度が一致しているかどうか**

記述の簡単さと性能を両立させるには



(1)メモリアドレスは規則的に並んでいるようにプログラマに見せつつも，

(2)処理系としては，データの分散配置に「ほぼ」従った粒度でコンシステンシ維持を行えば，

(3)記述の簡単さと性能を両立できる

- 有限要素法の**データの分散配置はかなり不規則**
  - → ブロックサイクリックなどの典型パターンでは対処できないため，こういう機能がぜひほしい
- 現在，**この連立方程式ソルバを題材として**いろいろと模索中...

感想

- 難しかった
- (DMI のような) 分散処理系を設計していく上で**実用的なアプリを知る**とても良い機会だった
  - → 今まで，データアクセスが定型的な単純なベンチマークしか見ていなかったことに気づいた
- 実行委員会の皆様および中島君に心より感謝申し上げます！