Vol.2020-HPC-177 No.9
2020/12/21

大規模線形問題における代数的多重格子法の粗格子集約手法の有効性評価

藤井 昭宏[1,a)]　田中 輝雄[1]

概要：我々はこれまで，代数的多重格子法の粗格子集約の手法（CGA）として，並列度が適切に制御された粗いレベルを一度に生成する手法を研究してきた．この通信最適化の効果を検証するため，これまでは主にストロングスケーリングで評価をしてきたが，本研究では，自由度のサイズが 10^{10} 程度以上のより大規模な問題での CGA の効果を検証するため，ウィークスケーリングでの性能の評価を行った．SA-AMG 法における代表的な粗いレベルの生成手法の設定を作成し，CGA による性能改善効果を調べた．その結果，ウィークスケーリングテストにおけるすべての問題サイズで最も安定に高速な実行時間を示していた．これは少ないオーバーヘッドで粗いレベルの行列を集約でき，効率良く粗いレベルの処理が扱えたためと考えられる．またよく利用される代数的多重格子法のライブラリである PETSc-gamg ソルバともウイークスケーリングでの設定で性能比較を行い，本手法の有効性を確認した．

Evaluation of Coarse grid aggregation for Large Sized Problems

1. はじめに

代数的多重格子法は問題行列が与えられると，その情報からサイズの小さい行列を生成し利用して線形問題を解く手法であり，有効に機能する場合には，未知数の個数を n としたときに $O(n)$ の計算量で収束することが知られている．$O(n)$ の解法はそれほど多くないため，大規模な問題では特に重要な解法の 1 つになっている．

そのような代数的多重格子法だが，粗いレベルの問題はかなり小さくなり，高並列な環境では保持する並列度と未知数の個数とのバランスが成立しにくくなり，効率的な計算処理が難しくなる．ここで特に最も粗いレベルでは，1 プロセスにまとめて直接解法で解くことも多いが，数千プロセスから，いきなり 1 プロセスにまとめようとするとその際のコストも大きくなってくる．そこで段階的に並列度を集約することが考えられるが，幾何的多重格子法では，中島らの研究 [1][2] が知られている．代数的多重格子法については，一度，粗いレベルを生成してから再分散し直す手法が利用されることが多い [3]．その中で我々は，粗い行列を作成する前にレベル間演算子の行列 P を事前に調整することで，並列度が適切に制御された粗いレベルを生成する手法を提案してきた [4]．　この手法は，粗いレベルでの通信コストを低減する効果があるため，通信コストが性能に大きな影響を与えるストロングスケーリングの設定で手法の有効性を確認してきた．

一方で，代数的多重格子法は大規模な問題での性能が良いことが 1 つの特徴になっている，本研究では自由度の個数が 10^{10} 程度の大きさの問題も含めるため，ウィークスケーリングの設定で本研究の粗格子集約手法の効果を検証する．

2. 代数的多重格子法

本研究は SA-AMG 法 [5] の実装手法について考えている．ここでは簡単なアルゴリズムの概略と本研究で関連がある項目を中心に記述する．

2.1 アルゴリズム概略

SA-AMG 法は問題行列 A_1 が与えられると，そこからマルチレベルな行列の階層構造を Alg.1 のように作成する．この図では，レベル数は L としている．まずはじめに，2 行目の $aggregate$ 関数にて未知数の全体集合を関連の強い背反な未知数集合に分割し，予備的な補間行列 $\tilde{P}$ を生成す

[1] 工学院大学
Kogakuin University
[a)] fujii@cc.kogakuin.ac.jp

る．$\tilde{P}$ の行数は，未知数全体の個数に，列数は未知数全体を分割した未知数集合の個数にそれぞれ対応する．$\tilde{P}$ の各列は未知数集合 1 個に対応し，その集合に含まれる未知数に対応する要素にのみ 1 が入り，それ以外の未知数に対応する要素には 0 を入れる．$\tilde{P}$ をそのまま補間行列として利用することもできるが，問題行列の情報を使い，$\tilde{P}$ の各列の値を初期ベクトルとして，$A_1x = 0$ に対する減速ヤコビ法を適用する (3 行目の *smooth* 関数) ことで，精度を上げた，補間行列 P を計算する．最後に P^TAP を計算することで，1 段粗いレベルの行列を計算する．これを繰り返し，未知数の個数がより小さくなった問題行列を生成し，問題行列も合わせて L 個の行列，$L-1$ 個の補間行列 P を生成する．

マルチレベルを生成した後，Alg.2 に記述されている V サイクルにより反復的に解を収束させる．本研究では，V-cycle 1 回の処理を CG 法の前処理として適用している．A_1, x_1, b_1 はそれぞれ，問題行列，初期解ベクトル，右辺ベクトルを表しており，$A_1x_1 = b_1$ を満たすように x_1 を繰り返し修正していくことになる．Alg.2 の最初の for 文は，ガウスザイデル法などの定常反復法を *smoother* 関数により適用し，残差ベクトルを計算しそれを P^T により粗いレベルに写像し，そのレベルの右辺ベクトルとしてセットする．この処理を一番粗いレベルまで繰り返し，一番粗いレベルでは未知数の個数も十分小さいため，6 行目の *solve* 関数に対応するが直接解法などである程度厳密に解く．その粗いレベルの解を 1 段上の未知数の多いレベルに写像し，足し込むことで，解が補正される．これは 8 行目の処理に対応する．その後，定常反復解法を適用することで，解を修正し，そのレベルでの解により近づける．これを一番未知数の多いレベルまで繰り返すことで，解ベクトルがマルチレベルな構造を使って効率良く補正されることになる．

Algorithm 1 Setup part, In: A_1, Out:$A_i, P_i, i = 2..L$

```
1: for i ← 2 to L do
2:     P̃_i ← aggregate(A_{i−1});
3:     P_i ← smooth(P̃_i);
4:     A_i ← P_i^T A_{i−1} P_i;
5: end for
```

Algorithm 2 V-cycle, In: A_1, x_1, b_1, Out:x

```
1: for i ← 1 to L − 1 do
2:     x_i ← smoother(A_i, x_i, b_i);
3:     b_{i+1} ← P_i^T (b_i − A_i x_i);
4:     x_{i+1} ← 0;
5: end for
6: x_L ← solve(A_L, x_L, b_L)
7: for i ← L − 1 to 1 do
8:     x_i ← x_i + P_{i+1} x_{i+1};
9:     x_i ← smoother(A_i, x_i, b_i);
10: end for
```

2.2 並列実装

Alg.1 のマルチレベルの行列生成部と Alg.2 の V-サイクルの処理があるが，*aggregate* 関数以外はガウスザイデル系緩和法であったり，疎行列疎行列積であるため，疎行列ベクトル積と同様な方法で並列計算ができる．詳細は論文 [6] の通りに作成しているが，ここでは並列実装の方針を簡単に説明する．まず疎行列のデータ構造としては，ブロック行分割にし，各プロセスのランク番号順にブロック行を割り当てるものとしている．このように未知数がプロセス番号に 1 次元的に連続して並んでいると仮定することで，プロセスごとの担当未知数の個数だけを記録すれば，未知数番号からどのプロセスが保持しているか，わかることになる．また各プロセスごとに担当する未知数とその境界領域（のりしろ）の未知数を合わせてローカルな未知数番号で計算をすすめることで，全体での未知数番号を意識することなく疎行列ベクトル積などができるようになる，一方，のりしろ部分の未知数は担当するプロセスから更新された新しい値をとってくる必要があるため，どのプロセスの何番目の未知数かテーブルとして保持する必要がある．これを通信テーブルと呼び，疎行列ベクトル積を行う前に，隣接通信を行い，のりしろ領域にある未知数の値を更新する必要がある．この未知数の接続のされ方は行列に依って当然異なり，通信パターンは変わってくる．Alg.1 からもわかるように，各レベルごとの行列 A_i とレベル i からレベル $i-1$ へと未知数を写像する行列 P_i があり，それぞれ固有に通信テーブルを設定している．一般的に，行列が効率的に分散されている時は，のりしろにある未知数の個数が最小化されており，隣接プロセス数も小さくなる．数値実験では各レベルの疎行列の隣接プロセス数とレベル間の補間行列 P の隣接プロセス数が記載されている．

aggregate 関数部分は未知数集合を背反する強連結している未知数集合に分けることになるが，各プロセスは領域分割により，自分の担当未知数を保持している．未知数集合をこの担当未知数内で作成することにして，領域境界には未知数集合を作らないことで各プロセスが担当領域を独立に分けることができ，並列に *aggregate* 関数の処理ができる．このアグリゲート生成戦略を decoupled[7] と呼ぶ．実装が単純になる反面，高並列になった場合に，プロセス並列度より小さいサイズの粗いレベルが生成できないため，問題になることが知られている．一方，領域境界の上に未知数集合を作っていく，coupled 戦略の場合，プロセス並列度より小さく粗くすることができるため，高並列時の粗いレベルの安定性につながることが知られている [7]．しかし，いずれにしても超高並列時には最も粗いレベルに近いところでは，各プロセスが未知数を数個しか担当しなくなり，ほとんどが通信時間になる，という問題が出てくると想定される．まとめると coupled と decoupled のアグリゲート戦略の有効性のイメージは表 1 となる．

表 1 Coupled and decoupled aggregation

parallelism	low	high	extraordinary high
coupled	○	○	△
decoupled	○	△	×

3. 粗格子集約手法とマルチレベル生成戦略

超高並列時には，どうやっても粗いレベルにおいて，1 プロセスの担当未知数の個数が小さくなるところが問題になる．

3.1 粗格子集約手法

そのため，我々は粗格子集約手法 [4] を研究してきた．PETSc の SA-AMG ソルバや Adams らの研究 [8]，Lin らの研究 [3] でも粗いレベルの行列生成後，再分散することで，並列度の集約を行った性能が報告されているが，本研究では粗いレベルを生成してから再分散するのではなく，並列度が適切に調整された粗いレベルを一度に作成する手法となっている．具体的には，Alg. 1 の行列 P の列番号を一度生成した後で，粗いレベルでの番号の付け替えをすることで，その列に相当する粗いレベルでの未知数の担当プロセス番号を変更することができ，4 行目の三つの行列の積の計算の結果から適切な分散が実現された行列を生成するように調整した [4].

補間行列 P の列番号の調整の仕方は，各プロセスが未知数集合を自分の担当未知数内に何個生成したかをプロセス間の隣接関係を含めてグラフ化する．つまり，Alg. 1 の 2 行目のレベル i 用の未知数としてアグリゲートを各プロセス領域内で独立に生成したのち，各プロセスを点として，レベル $i-1$ でのプロセス間の通信テーブルの隣接関係を枝とした重み付きグラフを作る．点には重みをつけ，その重みはそのプロセスの領域内で生成したアグリゲートの個数を設定している．このグラフを使い，1 プロセスあたりの想定未知数の個数になるように分割数を定め，ParMETIS[9] ライブラリを利用し点の重みの合計が均等になるようにこのグラフの点集合の分割を行う．これにより結合すると適切なアグリゲートの個数になるプロセス集合を生成することができる．この結果，粗いレベルでどのプロセス領域を結合するとよいかがわかるため，行列 P の列番号を，それを結合するプロセス間で連続にし，粗いレベルでの担当プロセスを決めてしまうことで，適切な行列 P が生成できる．これにより，下記のことが実現できる．

- 粗いレベルになっても，平均的に 1 プロセスあたり一定個数以上の未知数を保持するように適切に並列度を調整する．すなわち，未知数を保持しないプロセスを必要に応じて作成する．担当未知数がないプロセスはそのレベルでは計算に参加しないことを意味する．
- ParMETIS によりプロセス間隣接グラフ領域分割するが，それにより各領域内の点の重みの合計（粗いレベルの未知数の個数）を同程度になるようバランスさせながら，領域間で切断される枝の本数を最小化させている．つまり，粗いレベルにおけるプロセス間の隣接プロセス数が少なくなるようにしながら，合計の未知数のサイズは同程度になるように結合するプロセス集合を決めている．本研究では利用しなかったが，枝にも重みをつけ，プロセス間の隣接関係にも強い連結と弱い連結を区別できるようにすることで，異方性が強い問題にも対応することができるようになる．

ストロングスケーリング時のこの粗格子集約手法の効果は [4] にて調べられているが，代数的多重格子法は大規模問題でより重要になる側面もあるため，本研究では自由度の個数が 10^{10} 程度の大規模な問題での評価も含め，ウィークスケーリング時の効果について調べる．

3.2 マルチレベル生成戦略

ここでは本研究で性能比較の対象とするマルチレベル生成戦略を整理する． 2.2 節で aggregate 生成手法には各プロセス領域内で独立に未知数集合を生成する decoupled と，プロセス領域境界から未知数集合を生成する coupled があることを紹介した．前節での粗格子集約手法 (CGA) はプロセス領域を結合するため，プロセス領域が独立となる decoupled のみが対象となる．

本研究では，レベル数の上限を 10 とし，アグリゲート生成手法，最も粗いレベルの問題サイズの閾値，またその解き方（1 プロセスにまとめて直接解法か，並列反復解法か）により，表 2 のように 4 つの代表的なマルチレベル生成戦略を考え CGA の有効性の評価のため，性能比較することとした．

CGA_LU decoupled なアグリゲーションで CGA を組み合わせ最下層は 500 個の未知数以下になるまで粗くし，1 プロセスで直接解法を行う．

w/o_LU decoupled なアグリゲーションで最下層は 40000 個以下の未知数になるまで粗くし，そこではマルチカラーガウスザイデルを 20 回適用する

LU decoupled なアグリゲーションで CGA を適用せず，500 個以下になるまで，もしくはレベル数上限まで粗いレベルの生成を行い，最後は 1 プロセスにまとめて LU 分解を行う．

coupled_LU coupled なアグリゲーションで粗格子を作成し，500 個以下になったら，1 プロセスにまとめて LU 分解で処理する

マルチレベル生成戦略という言葉を使っているが，その手法により Alg. 1 の行列 P の値，非ゼロ構造が変化し，それにより生成される粗いレベルの行列も変化し，収束性から粗いレベルの行列の分散状況まで決まってくる．

表 2　マルチレベル生成戦略
Table 2　Multi-level setup strategies

	CGA_LU	w/o_LU
Aggre.	decoupled	decoupled
Coarsest DOF	500	40000
Coarsest.sol.	LU	MCSGS(20iter)
	LU	coupled_LU
Aggre.	decoupled	coupled
Coarsest DOF	500	500
Coarsest.sol.	LU	LU

表 3　Oakbridge-CX 1 ノード概略
Table 3　Oakbridge-CX 1 node

System name	Oakbridge-CX
CPU name	Intel Xeon Platinum 8280
number of cores/socket	28
number of sockets/node	2
frequency	2.7GHz
memory	192GiB(DDR4)
compiler	Intel Parallel Studio

4. 数値実験

4.1 問題設定と目的

ウィークスケーリングの環境において大規模問題における，粗格子集約手法の効果を調べることを目的として，4.2 節では代表的なマルチレベル生成戦略を作成し性能の比較を行い，さらに 4.3 節では PETSc の SA-AMG ソルバのデフォルトのパラメタ設定でのときのソルバを基準として，現状の我々のソルバの性能を分析した．問題として，三次元立方体形状の有限体積法に基づくポアソン方程式 $\frac{\partial}{\partial x}(\lambda\frac{\partial T}{\partial x})+\frac{\partial}{\partial y}(\lambda\frac{\partial T}{\partial y})+\frac{\partial}{\partial z}(\lambda\frac{\partial T}{\partial z})=f$ を対象とした．T が未知数であり，右辺の f は定数である．拡散係数 λ が不連続に変化する問題で，最大 1.0×10^{5} から $1.0\times10^{-}5$ まで変化する問題である．ガウスザイデル前処理付き CG 法などでは収束しない問題となっている．

4.2 節では, Oakbridge-CX を最大 256 ノード利用し MPI-openMP のハイブリッド環境での実験となっており，4.3 節では PETSc ライブラリとの比較のため, 最大 32 ノードを利用し，flat MPI にて性能を比較している．1 ノードの構成は表 3 のようになり，28 コア CPU が二つ利用可能である．

本実験で使用した我々のソルバのコードは [10] に公開されている．

4.2 粗いレベル生成手法間での比較

本節の実験では，Oakbridge-CX 1 ノードあたり，18 プロセス 3 スレッドを起動し，54CPU コアを利用する形で実行した．1 プロセスあたり $135\times135\times135$ の未知数を割り当て，1 ノードあたり，4.4×10^{7} 個の未知数を割り当てている．

ソルバ比較対象としては，3.2 節で挙げた 4 種 CGA_LU，w/o_LU，LU，coupled_LU である．CGA_LU，w/o_LU，LU の中では，CGA がある場合とない場合の比較になるため，CGA の処理のオーバーヘッド，効果をよく見れる分析となる．coupled_LU は プロセス領域境界を超えた未知数集合を生成していく手法で，CGA をせずともプロセス境界を越えて未知数の個数を減らしていけるメリットがあるが，1 プロセスあたりの未知数の個数を調整してプロセス領域を集約していく手法ではない．

CGA の設定としては，各プロセスが平均的に 500 個の未知数を保持できるように並列度を集約する．各レベルでのスムーザの設定はプロセス領域間の依存関係を無視した，対称マルチカラーガウスザイデル 1 回となっており，安定性を高めるため 0.8 倍で減速させている．強連結成分の閾値は 0.1 とし，レベル生成するごとにこの閾値も 0.8 倍して小さくしている．

その結果，この図 1 のようになった．4 通りの粗いレベル生成の設定を比較しており，横軸が問題サイズ，縦軸が収束に要する時間となっている．点線と実線があり，点線はマルチレベルの生成に必要となる時間を表す．その上に反復解法部の時間をのせ，実線は問題を 1 回解くための総実行時間を表している．最大規模の問題では 256 ノード ($\text{ノードあたり } 16\,processes\times 3\,threads \text{ で } 54cores$ となり 256 ノードで 13824cores) 利用しているが，2 番目のサイズの問題では，64 ノード利用している．各点には数字が書かれているが，その数字は収束までに要した AMG-CG 法の反復回数を表している．1 プロセスあたりのスレッド数は自由に変えられるが，1 万以上のプロセスを立てると通信に関連するオーバーヘッドもでてくる懸念もあったため，最大問題サイズのときにプロセス数が数千プロセスの範囲に収まるように 3 スレッドと設定した．

まず全体として，CGA を組み合わせた解法 (CGA_LU) はほとんどすべてのサイズで，最適か最適に近い性能となっていることがわかる．最大サイズの問題においては，総実行時間では，CGA を用いない最も早かった設定より 15 %近く改善させていることがわかった．次にマルチレベル生成のためのコストに注目すると，ウィークスケーリングで順調に性能が出せているところでは，30 回の反復でおよそ 30 秒であり，データ生成部がおよそ 10 秒であるため，AMG-CG 反復 10 回分のコストでマルチレベルが生成されていることがわかる．これは最大規模の問題においてもほぼ成立していた．

今回問題サイズが増大するにつれて，収束に要する反復回数も増大してしまい，実行時間も増加する形になっているが，これは問題に対して，スムーザの設定に改善の余地があることを示唆している．マルチカラー対称 GS 法を 1

回のみスムーザとして利用していた．スムーザの適用反復回数をマルチカラー SGS2 回にするなど，すると問題サイズの増加による反復回数の増加もより抑えられたものになると予測される．

次にそれぞれのソルバ設定での性能差を考える．それぞれのソルバ設定 (CGA_LU,LU ,w/o_LU, coupled_LU) での最大サイズの問題での状況について表 4，表 5，表 6，表 7 に示す．

表は各レベルの行列の状況が書かれており，疎行列の情報，分散された疎行列の隣接プロセス数の情報，V サイクルでの所要時間，の三つのパートに分かれている．まず最初のパートは，DOF，# of P, ave. nonz, はそれぞれ行列の行数，分散されているプロセス数，各行あたりの平均非ゼロ要素数である．二つ目のパートには，各レベル，とレベル間での行列の隣接関係が最大何個あるかが書かれている．どのテーブルもレベル 1 の max neib は 26 個の隣接プロセスを持っているが，これは三次元直方体形状に区切っているため，隣接 26 個の領域とデータのやりとりが発生していることを示す．bet levs はレベル間演算子の分散疎行列の隣接関係を表している．ここでレベル 2 に書かれている値はレベル 2，1 間の行列の情報が書かれているため，レベル 1 の所は空白になっている．最後に三つ目の実行時間の部分は，max time は各レベルでのスムーザの処理時間の累積の時間，bet lev[s] はレベル間演算子の処理の累積時間を表している．それぞれ全プロセスの中での最大値を示している．レベル間演算子はレベル i のところはレベル $i, i-1$ の間の時間を表すためレベル 1 のところは空白になっている．

次に各表について簡単に説明する．まず表 4 は, CGA_LU で最大並列度 ($4608\,process \times 3\,thread\,on\,256\,nodes$) の時の状況を表している．表からわかるように，$10^{10}$ 程度の未知数がある問題に対して 6 回粗くすることで，37 個の未知数の問題まで自由度を落としている．それに伴って，2 列目の並列度も落ちていることがわかる．今回の設定では，1 プロセスあたり平均的に 500 程度の未知数が保持できるように並列度を落とす設定にしているが，レベル 4 のプロセス数と自由度のバランスを見てもおよそそのようになっていることがわかる．三列目の行あたり平均非ゼロ要素数は粗くなればなるほど増え，最下層部ではほぼ密行列の状態になっている．4 列目は各プロセスが保持する疎行列ののりしろ関係を表している．レベル 4 で最大 33 個の隣接プロセスがある場合があるが，CGA により並列度を適切に落としているため，1 行あたりの非ゼロ要素数が増えても，粗いレベルでの隣接プロセス数は抑えられていることがわかる．ただ，レベル間演算子の通信テーブルが増大しており，レベル 5 では隣接プロセス数が 124 になるものもでてきていることがわかる．レベルごとのスムーザの処理時間とレベル間の移動のための処理時間が 6，7 列目に記述されている．各レベルでの処理時間を見ると，粗いレベルに行くほど減っている．レベル間演算子の時間はレベル 3，4 間での時間が上のレベルに比べて増えているが，総じて時間を減らすことができており粗いレベルがボトルネックにはなっていないことがわかる．

表 5 では，LU のときで最大問題サイズ 10^9 程度（$1152\,process \times 3\,thread\,on\,64\,nodes$）のときの状況を示している．decoupled アグリゲーションのため各プロセス領域に 1 要素のみは残るため，最下層レベルの未知数の基準の個数である 500 個まで達することができず，レベル 5 から上限のレベル 10 まで適切に未知数の個数を小さくできていない．レベル 10 では，1 プロセスに行列を集約して，逆行列を適用している．この表 1 列目からレベル 8 以降の行列は平均非ゼロ要素数が行数と同等になっており，密行列となっていることがわかる．そのため，表の 4 列目の隣接プロセス数も，レベル 7 くらいから全プロセスと通信をするプロセスがでてきており，非常に通信のオーバーヘッドが高くなる状況になっていることがわかる．表 5 の 6，7 列目を見ても無駄に粗いレベルを作成しその部分の計算コストが全体に大きな悪影響を与えていることがわかる．

一方，表 6 では decoupled アグリゲーションであるが，最下層レベルの未知数の基準の個数を 40000 個以下と設定したときの最大サイズの問題に対する解法の様子である．未知数の個数が数万個残っている状態で粗いレベルの生成を止めることで並列性が保てないサイズの粗いレベルは生成されなくなる．また最も粗いレベルでも 1 プロセスにまとめる処理もいれず，並列反復解法を適用をしている．これにより，すべてのレベルで集約する構造はなくなり，完全に並列に動作させることができる．粗いレベルの計算による効率の劣化を抑え，通信テーブルサイズも適切に抑えられており，実行時間をみても粗くなるごとに適切にへっていっており，ある程度うまく機能していることがわかる．ただ，図 1 に示されているように，複数の問題サイズで収束に要する反復回数が大きく増大していた．これは最下層レベルの未知数の個数が大きいまま残しているため，最下層レベルの処理の MCSGS の 20 反復で解ききれていないことが原因であるように見える．

表 7 を見る．Coupled aggregation で，最大問題サイズの時の状況を表している．この場合，領域境界を越えた未知数集合を作成できるため，プロセス数に阻まれることなく粗いレベルサイズを小さくしていくことができる．例えばレベル 6 では 161 プロセスに 1101 行の行列が分散されていることがわかる．但し，ほとんど密につながっているため，最後のレベルではそれが 1 つの未知数にまとまってしまっており，最も粗いレベルでの補正がしっかりできていないように見え，そのため，必要な反復回数も増大しているように見えている．CGA と比較して，1 プロセスあたりの平均問題サイズを保持するようにはできないため，1

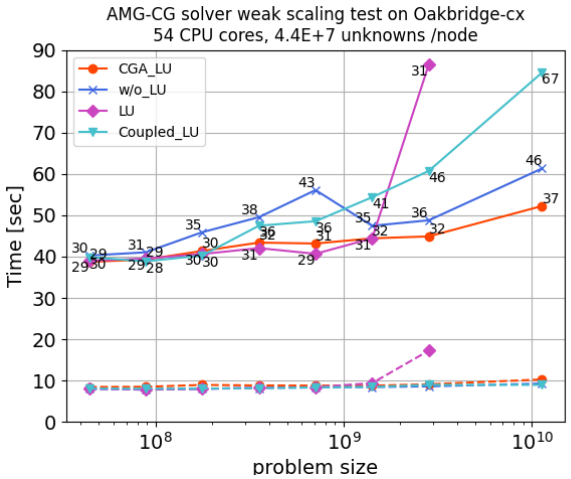

図 1　粗格子生成戦略とウィークスケーリング性能
Fig. 1　Weak scaling performance and multi-level setup strategies

表 4　CGA_LU での階層構造と各レベル，レベル間での処理時間
Table 4　CGA_LU's hierarchical structure and each level and between levels calculation time

Lev.	DOF	# of P	ave. nonz	max neib	bet levs	max time[s]	bet lev[s]
1	11337408000	4608	26.90	26		31.70000	
2	416679696	4608	69.70	26	26	5.14000	3.09
3	11919993	4608	92.89	26	26	2.05000	1.29
4	522267	1045	116.73	33	65	1.51000	2.84
5	25947	52	137.30	26	124	0.08900	0.318
6	1062	3	121.00	2	40	0.05300	0.073
7	37	1	35.40	0	2	0.00163	0.0037

表 5　LU での階層構造と各レベル，レベル間での処理時間
Table 5　LU's hierarchical structure and each level and between levels calculation time

Lev.	DOF	# of P	ave. nonz	max neib	bet levs	max time[s]	bet lev[s]
1	2834352000	1152	26.90	26		26.500	
2	104157013	1152	69.40	26	26	3.020	2.1
3	2978157	1152	92.03	26	26	0.254	0.251
4	129966	1152	113.38	26	26	0.107	0.083
5	8453	1152	143.57	80	26	0.060	0.032
6	1317	1152	254.63	435	26	1.800	0.019
7	1251	1152	941.20	1151	79	10.500	2.277
8	1244	1152	1244.00	1151	404	12.700	6.6
9	1240	1152	1240.00	1151	785	5.700	6
10	1229	1152	1229.00	0	1151	0.030	7.32

プロセスあたりの未知数の個数は数個になることもあり，そのため，レベル 6 での最大隣接プロセス数も 230 と，かなり多い形になっていた．

最後に，図 1 を見直すと，今回の実験では問題サイズで 10^8 程度，並列度で $144\,process \times 3\,threads$ くらいから性能差が出始めたが，すべての問題サイズを見て，CGA が入っているソルバ (CGA_LU) が最も安定して高速なソルバ設定になっていた．並列度の小さい問題では，CGA を行わなくても収束するが，そこでの性能差から CGA のオーバーヘッドが小さいことが分かり，さらに問題サイズが変化しても適切に並列度が調整されるため，収束性もより安定していた．

表 6　w/o_LU での階層構造と各レベル，レベル間での処理時間
Table 6　w/o_LU's hierarchical structure and each level and between levels calculation time

Lev.	DOF	# of P	ave. nonz	max neib	bet levs	max time[s]	bet lev[s]
1	11337408000	4608	26.90	26		39.30	
2	416679696	4608	69.70	26	26	6.13	1.77
3	11919993	4608	92.89	26	26	2.59	0.093
4	522267	4608	116.73	26	26	1.75	0.028
5	34057	4608	155.60	81	26	0.69	0.016

表 7　coupled_LU での階層構造と各レベル，レベル間での処理時間
Table 7　coupled_LU's hierarchical structure and each level and between levels calculation time

Lev.	DOF	# of P	ave. nonz	max neib	bet levs	max time[s]	bet lev[s]
1	11337408000	4608	26.90	26		57.500	
2	408543049	4608	78.10	26	26	9.200	5.67
3	11597241	4608	90.20	26	26	3.650	2.48
4	453623	4608	107.60	26	26	2.730	2.17
5	17309	4608	113.45	105	27	0.493	0.641
6	1101	161	130.22	230	88	0.120	0.223
7	1	1	1.00	0	161	0.007	0.046

4.3　PETSc-gamg との性能比較

最後に PETSc ライブラリ [11] に含まれる SA-AMG ソルバである gamg(以後 petsc-gamg と記述) との性能比較を行なった．Oakbridge-CX を最大 32 ノード利用し実験を行った．petsc-gamg との比較のため，フラット MPI で，1 ノードあたり 54 プロセス立ち上げ，1 プロセスあたり $45 \times 135 \times 135$ の未知数を割り当て実行している．petsc-gamg との比較のため未知数のサイズは 32 ビット整数型の範囲を超えないように設定している．最大 32 ノードを使用し，DOF が 6.29×10^8 のサイズの問題を解いている．

また，解法としてはどちらも AMG 前処理付き CG 法とし，petsc-gamg は flat-mpi の実装のみであるため，我々のソルバも flat-mpi で同じデータ分散にして評価した．petsc-gamg はデフォルトのパラメタ設定，我々のソルバも前節の CGA_LU と同じパラメタ設定にしているが flat mpi であるため，緩和法としてはマルチカラー化されていない対称ガウスザイデルを 1 回適用した．

Petsc は計算環境にデフォルトでインストールされているものを利用したが，バージョンは 3.11.2 であり，petsc-gamg のパラメタ設定は，KSPSetFromOptions 関数により下記オプションを実行時引数にをつけて設定した．これらのオプションにより Petsc 側で実行時間をステージごとに分類して表示するが，GAMG Solve と MG Apply のステージの合計時間を petsc-gamg の実行時間としてしている．

```
-ksp_type "cg" -pc_type "gamg" -ksp_rtol 2.0E-7
-log_view -pc_mg_log -ksp_monitor
```

図 2 に実行結果を示す．これまでと同様に横軸が問題サイズ，縦軸が実問題を解くための実行時間を示している．青の実線は petsc-gamg の実行時間を示し，赤の実線は CGA_LU での実行時間を示す．赤の点線部分はマルチレベル生成のための時間を示している．また実線上に書かれている数字は，収束に要した AMG-CG 法の反復回数を表している．図からまず，petsc-gamg の実行時間より我々

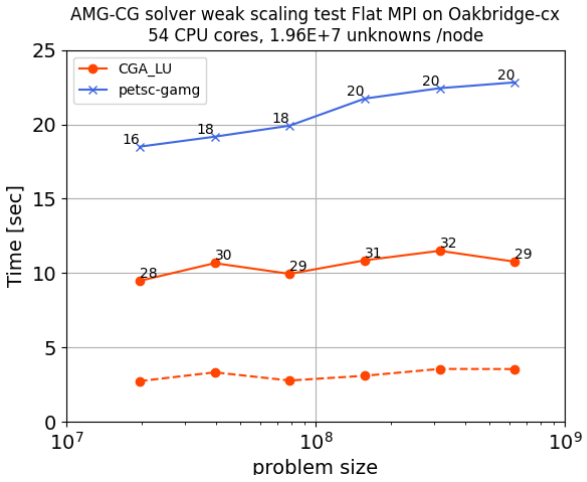

図 2　PETSc-gamg との性能比較
Fig. 2　Performance comparison with PETSc-gamg

のソルバの方が 2 倍近く高速に収束していることがわかる．petsc-gamg のソルバはデフォルトのパラメタ設定であり，パラメタ最適化の余地が残されている．ここでは性能のベースラインとして参照している．

また収束に要する反復回数を見てみると，petsc-gamg の方が 20 回前後，我々のソルバが 30 回前後と petsc-gamg の方が反復回数をより少なく収束している．これは，各レベルで適用される緩和法が petsc-gamg で適用されるものの方が強力であることを示唆している．問題サイズが増大しても我々のソルバの方が反復回数の増大を抑えられており，スケーラビリティが高いことを示していた．また問題サイズによらず一定の実行時間で収束していたことは，CGA が適正に機能し，粗いレベルでプロセス数が無駄に多い場合に発生するオーバーヘッドを適切に抑えているためと考えられる．

5. おわりに

粗格子集約手法（CGA）の評価を大規模な問題で評価するため，ウィークスケーリングの設定で CGA なしの手法とありの手法で実行時間を計測し分析した．結果，CGA を用いることで，最も粗いレベルのサイズを並列度に依存せず小さくできるため，問題サイズによらず収束性を高く保ちやすいこと，並列度を集約するコストも生成部の時間や反復解法部の時間からはみられないほど小さいことを確認した．また代表的なライブラリとの比較として，PETSc-gamg と性能比較を行った．PETSc-gamg のデフォルトのパラメタ設定を用いたものと比較して，CGA を組み込んだ解法は 2 倍程度高速に収束させていた．また問題サイズを大きくしたときの実行時間の増加率もより小さくなっており，CGA が有効に機能していると考えられる．

謝辞　本研究を進めるにあたり，東京大学の中島研吾先生にテスト問題生成ルーチンの提供をいただき助言をいただいた．ここに深謝する．本研究は学際大規模情報基盤共同利用・共同研究拠点，および，革新的ハイパフォーマンス・コンピューティング・インフラの支援による (課題番号: jh200041-NAH)．

参考文献

[1] Nakajima, K.: OpenMP/MPI Hybrid Parallel Multigrid Method on Fujitsu FX10 Supercomputer System, *Cluster Computing Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on*, pp. 199–206 (online), DOI: 10.1109/ClusterW.2012.35 (2012).
[2] Nakajima, K.: Optimization of serial and parallel communications for parallel geometric multigrid method., *ICPADS*, pp. 25–32 (2014).
[3] Lin, P. T.: Improving multigrid performance for unstructured mesh drift–diffusion simulations on 147,000 cores, *International Journal for Numerical Methods in Engineering*, Vol. 91, No. 9, pp. 971–989 (2012).
[4] Nomura, N., Fujii, A., Tanaka, T., Marques, O. and Nakajima, K.: Algebraic Multigrid Solver Using Coarse Grid Aggregation with Independent Aggregation, *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1104–1112 (online), DOI: 10.1109/IPDPSW.2018.00170 (2018).
[5] Vanek, P., Mandel, J. and Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems, *Computing*, Vol. 56, No. 3, pp. 179–196 (online), DOI: 10.1007/BF02238511 (1996).
[6] 藤井昭宏, 西田　晃, 小柳義夫：領域分割による並列 AMG アルゴリズム，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 44, No. SIG06(ACS1), pp. 9–17 (2003).
[7] Tuminaro, R. and Tong, C.: Parallel Smoothed Aggregation Multigrid : Aggregation Strategies on Massively Parallel Machines, *Supercomputing, ACM/IEEE 2000 Conference*, pp. 5–5 (online), DOI: 10.1109/SC.2000.10008 (2000).
[8] Adams, M., Bayraktar, H., Keaveny, T. and Papadopoulos, P.: Ultrascalable Implicit Finite Element Analyses in Solid Mechanics with over a Half a Billion Degrees of Freedom, *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pp. 34–34 (online), DOI: 10.1109/SC.2004.62 (2004).
[9] Karypis, G. and Kumar, V.: MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0, http://www.cs.umn.edu/m̃etis (2009).
[10] Fujii, A.: AMGS, Version 1.10, http://hpcl.info.kogakuin.ac.jp/docs/amgs (2020).
[11] Balay, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F. and Zhang, H.: PETSc Users Manual, Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory (2013).