Vol.2015-HPC-148 No.5
2015/3/2

動的負荷分散による階層型行列計算の並列化

棟形 克己[1,a)]　平石 拓[2,b)]　伊田 明弘[2,4]　岩下 武史[3,4]　中島 浩[2]

概要：階層型行列（$\mathcal{H}$ 行列）は，N 個の要素間の $N \times N$ 個の相互関係を表す密行列の圧縮表現の一つである．本研究では，$\mathcal{H}$ 行列の生成および $\mathcal{H}$ 行列ベクトル積の MPI/OpenMP ハイブリッド並列化を，動的負荷分散を用いて行った．$\mathcal{H}$ 行列の生成は，小行列（葉行列）への行列の区分けと，各葉行列の要素計算により行われる．後者は葉行列単位のタスク並列化が可能だが，各コアにタスク集合を静的に割り当てる実装では，各タスクの計算負荷を正確には見積もれないこと，および全体に対する負荷割合が大きなタスクの存在により十分な負荷均衡が得られない．そこで，MPI および OpenMP の 2 レベルの階層型マスタワーカ方式による動的タスク割り当てを行い，さらに OpenMP レベルでは大きなタスクにプロセス内の全スレッドを割り当てることで負荷を均衡化した． $\mathcal{H}$ 行列ベクトル積でも同様のタスク並列化が可能だが，タスクのプロセス間移動のコストが大きいため，MPI レベルでは葉行列生成を担当したプロセスに引き続きその葉行列に関する部分計算を静的に割り当て，OpenMP レベルでのみ生成処理と同様の動的タスク割り当てを行った．生成処理の負荷均衡化の結果，この方式でプロセス間においても良好な負荷均衡が得られる．表面電荷法による係数行列生成およびその行列に対する行列ベクトル積を例題として性能評価を行った結果，32 プロセス ×8 スレッドによる並列実行では，従来の負荷見積もりに基づく静的割当手法に対して，$\mathcal{H}$ 行列生成では 3.4 倍，$\mathcal{H}$ 行列ベクトル積では 2.5 倍の性能が得られた．

キーワード：階層型行列，近似行列，動的負荷分散，ハイブリッド並列

Parallel Hierarchical Matrix Arithmetics using Dynamic Load Balancing

Munakata Katsumi[1,a)]　Hiraishi Tasuku[2,b)]　Ida Akihiro[2,4]　Iwashita Takeshi[3,4]
Nakashima Hiroshi[2]

Abstract: Hierarchical matrix ($\mathcal{H}$-matrix) is an approximated form to represent $N \times N$ correlations of N objects, which usually requires a $N \times N$ huge dense matrix. This paper proposes hybrid MPI/OpenMP implementations of $\mathcal{H}$-matrix generation and $\mathcal{H}$-matrix-vector multiplication using dynamic load balancing. $\mathcal{H}$-matrix generation is done by partitioning a matrix into submatrices called leaf matrices, followed by calculating element values of the leaf matrices. We can apply task parallelism to the latter operation by treating each leaf matrix as a parallelization unit. However, we cannot achieve a good speedup when assigning a set of tasks to each processor core statically, because (1) we cannot predict the computational amount of each task precisely and (2) there exist tasks whose ratios of the computational amounts to the total amount are too large. We solved these problems by (1) dynamic task assignment based on the hierarchical master-worker method with the MPI and OpenMP levels, and (2) dividing a large task and executing it in parallel using all threads in an MPI process. We can apply the same parallelization strategy to $\mathcal{H}$-matrix-vector multiplication. However, because the task migration cost among processes is too high, we reused the same task assignment as in $\mathcal{H}$-matrix generation on the MPI level, and performed dynamic task assignment only on the OpenMP level. We can get better load balance even among processes due to the dynamic load balancing used in $\mathcal{H}$-matrix generation. We evaluated the performances of our implementations when generating a coefficient matrix used in the surface charge method as an $\mathcal{H}$-matrix and multiplying the $\mathcal{H}$-matrix by a vector. As a result, in an execution with 32 processes × 8 threads, we achieved a 3.4 times and 2.5 times better performance in $\mathcal{H}$-matrix generation and $\mathcal{H}$-matrix-vector multiplication respectively, than the existing implementations that perform static task assignment based on estimated computational amounts of tasks.

Keywords: Hierarchical Matrices, Approximated Matrices, Dynamic Load Balancing, Hybrid Parallelism

1. はじめに

境界要素解析や多体問題シミュレーションでは，物理要素間の相互関係を表す密行列を係数行列とする連立一次方程式を解くことが必要になることが多い．しかし，N 個の物理要素の全相互関係を表す密行列の要素数は N^2 となり，これを直接扱う計算手法では必要なメモリ容量や計算時間が N を増やすにつれて急激に増大してしまう．そこで，そのような密行列の近似圧縮表現である $\mathcal{H}$ 行列 [1][2][3] を用いる手法が提案されている．$\mathcal{H}$ 行列とは**図 1** のように区分けされた行列の各ブロック（葉行列）を，積がその葉行列の近似となるような 2 つの低ランク行列で表現したものである（ただし図中では “full” と区分けされている一部の小さな葉行列は全要素の値により直接表現される）．$\mathcal{H}$ 行列は N 個の物理要素データから密行列の直接表現を経由することなく生成可能で，そのデータサイズは $\mathcal{O}(NK\log N)$ に抑えられる．ただし K は要求する近似精度に応じて設定されるパラメータであり，通常 N より十分小さい．

$\mathcal{H}$ 行列を利用するアプリケーション全体の高速化のためには，$\mathcal{H}$ 行列ベクトル積のような $\mathcal{H}$ 行列に対する演算だけでなく，$\mathcal{H}$ 行列の生成そのものの高速化・並列化も重要である．しかしながら，葉行列を並列化単位として静的なタスク割当を行う従来の並列化では，負荷の不均衡により並列計算性能が抑えられるという問題があった．負荷を均衡させるのが困難である理由として，各タスクの計算量が事前予測困難であることがあげられる．既存の実装である $\mathcal{H}$ACApK [4][5] では，計算量を経験則に基づいて見積もる試みを行っているが，それでも対象によっては十分な負荷均衡が得られないことがある．また，行列の性質によっては，行列全体に対するサイズが非常に大きい葉行列によって並列性能が律速されることもある．

そこで本研究では，$\mathcal{H}$ 行列生成を動的負荷分散を用いて並列化することで負荷均衡を達成することを試みた．さらに，大きな葉行列を MPI プロセス内の全 OpenMP スレッドで並列処理することで，より良好な負荷均衡を得る方法を考案した．また，同様の理由により従来実装では十分な負荷均衡が得られていなかった $\mathcal{H}$ 行列ベクトル積に対しても，$\mathcal{H}$ 行列生成に対して適用した戦略を用いることで，良好な負荷均衡を得ることを試みた．

本論文の構成は以下の通りである．まず，2 章で $\mathcal{H}$ 行列について説明を行い，本研究で並列化対象となる，$\mathcal{H}$ 行列生成および $\mathcal{H}$ 行列ベクトル積のアルゴリズムについて論じる．次に 3 章と 4 章では，$\mathcal{H}$ 行列生成と $\mathcal{H}$ 行列ベクトル積の従来実装および提案実装について述べ，5 章でそれらの実装に関する性能評価結果を示す．続いて 6 章で関連研究について述べ，また 7 章で今後の課題を示した後，8 章で本論文のまとめを行う．

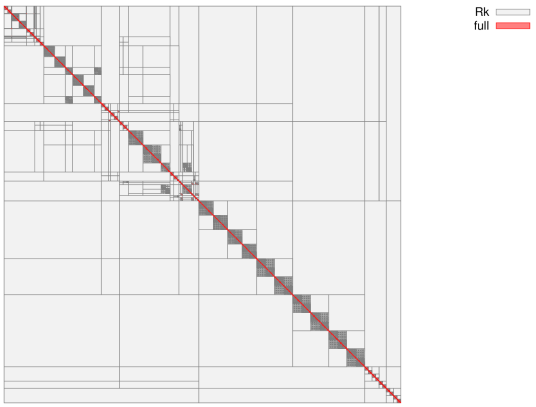

図 1 $\mathcal{H}$ 行列

2. $\mathcal{H}$ 行列法

本章では，本研究で動的負荷分散を用いた並列化の対象とする，$\mathcal{H}$ 行列法について説明する．まず，2.1 節では密行列の近似圧縮表現である $\mathcal{H}$ 行列の概要を説明する．次に 2.2 節では，ACA+法 [1] に基づく $\mathcal{H}$ 行列の生成法を概説する．最後に，2.3 節で $\mathcal{H}$ 行列ベクトル積について説明する．

2.1 $\mathcal{H}$ 行列の概要

$\mathcal{H}$ 行列は，「近似小行列」または「フル小行列」として表現された部分行列（葉行列）の集合として表現される．ここで近似小行列は，2 つの低ランク行列の積で表現された葉行列である．すなわち，$l \times m$ のサイズの近似小行列は，$l \times k$ と $k \times m$ の 2 つのランク k の行列で表現される．ただし，k は l や m より十分小さい．フル小行列は，全要素の値により圧縮することなく表現された葉行列である．これらの葉行列は互いに重なり合うことも隙間を作ることもなく行列全体を埋め尽くす．葉行列を近似小行列とフル小行列のどちらで表現するかは，その行列が表現する物理要素間の関係などと要求される近似精度により定まるが，実際に要素を生成することなく決定することができる．フル小行列が占める領域の行列全体に対する割合が十分小さい場合，サイズ $N \times N$ の $\mathcal{H}$ 行列のデータサイズは $\mathcal{O}(NK\log N)$ となる．ただし K は k の許容される上限であり，要求される近似精度に応じて変化するが，通常 N より十分小さな値に設定される．

N 以下の自然数の組 (i, j) の集合 $[1, N] \times [1, N]$ の分割

[1] 京都大学情報学研究科
Graduate School of Informatics, Kyoto University
[2] 京都大学学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University
[3] 北海道大学情報基盤センター
Information Initiative Center, Hokkaido University
[4] JST CREST
a) munakata@sys.i.kyoto-u.ac.jp
b) tasuku@media.kyoto-u.ac.jp

（partition）の中で，その全ての元 $p = S_p \times T_p$ について S_p と T_p がともに連続した自然数の集合であるような集合族 P を，$[1, N] \times [1, N]$ の「区分け」と呼ぶ．正方行列 $A \in \mathbb{R}^{N \times N}$ の要素 $[A]_{i,j}$ $(i \in S_p,\ j \in T_p)$ からなる小行列を $A|^p$ と表記し，小行列の集合 $\{A|^p \mid p \in P\}$ を P による A の区分けと呼ぶ．また，行列 A の $\mathcal{H}$ 行列表現を $\tilde{A}$ と表記し，$A|^p$ に対応する $\tilde{A}$ の葉行列を $\tilde{A}|^p$ と表記する．$\tilde{A}|^p$ が近似小行列であるとき，$\tilde{A}|^p$ は以下のように 2 個の行列の積として表現される．

$$\begin{aligned} &\tilde{A}|^p = V_p W_p, \\ &V_p \in \mathbb{R}^{S_p \times r_p},\ W_p \in \mathbb{R}^{r_p \times T_p}, \\ &r_p \leq \min(\#S_p, \#T_p) \end{aligned} \tag{2.1}$$

ここで $\#S$ は集合 S の元数を意味する．また $r_p \in \mathbb{N}$ は $\tilde{A}|^p$ のランクを表し，通常 $\#S_p$ や $\#T_p$ より十分小さな値をとる．一方 $\tilde{A}|^p$ がフル小行列であれば，$\tilde{A}|^p = A|^p$ であり，元の小行列と一致する．したがって，もし全ての葉行列がフル小行列であれば，$\mathcal{H}$ 行列 $\tilde{A}$ は行列 A に一致する．

2.2 $\mathcal{H}$ 行列生成のアルゴリズム

$\mathcal{H}$ 行列生成は主に以下にあげる二つのステップに分けられる．

1. 行列の区分け P の決定
2. ACA+法を用いた葉行列の生成（フィル処理）

本論文では 1. の詳細に立ち入らないが，概要は以下のようになる．一般的に密行列は明示的な構造を持たないが，境界要素法などに現れる密な係数行列には，近似小行列で表現可能な陰的な構造が存在していることが知られている．$\mathcal{H}$ 行列法では，高いデータ圧縮性を達成するために，より大きく，より多くの近似小行列を密行列中に見出すことが重要になる．そこで行列 A が表現すべき N 個の要素間の関係に内在する階層性を利用して，要求精度を満たしつつ高い圧縮性を達成するように区分け P を構築する．たとえば天文学的なシミュレーションで見られるように，空間中に分布した N 個の物理的要素間の関係が両者の距離が遠いほど弱まるような例を考える．このような場合，内部では互いに近接している二つの要素集合 E_1 と E_2 との間の距離が十分に大きければ，E_1 の要素と E_2 の要素との関係を全ての要素対について完全に記述するのではなく，少数の代表要素をそれぞれから選択し，一方の代表要素と他方の全要素の関係に基づく表現，すなわち近似小行列を用いた表現により高精度の近似が可能となる．そこで，空間を分割して要素集合を求め，集合の空間的サイズや元数と集合間の距離に基づき近似精度を見積もり，精度が十分高ければ集合対に対応する P の元を確定し，不十分であれば再帰的に空間分割処理を行うことで，P を構築することができる．

2. のフィル処理では，1. で決定した区分け P の各元 p に対応する葉行列 $\tilde{A}|^p$ の要素計算を行う．近似小行列のフィル処理についてはいくつかの方法が提案されているが [1]，本研究では ACA+法と呼ばれる方法を採用している．この方法は，ベースとなる方法である ACA (Adaptive Cross Approximation) 法 [6][7] に改良を加えたものであり，どちらの方法も $A|^p$ の中からピボットと呼ばれる列ベクトル $[A|^p]_{*,j}$ と行ベクトル $[A|^p]_{i,*}$ を順次選択し，近似小行列 $\tilde{A}|^p$ を表現する V_p の列ベクトルと W_p の行ベクトルに，以下のベクトルを加える操作を繰り返す．

$$[V_p]_{*,k} = \frac{[A|^p]_{*,j} - \sum_{l=1}^{k-1} [W_p]_{i,l} [V_p]_{*,l})}{\delta_V(i,j)} \tag{2.2}$$

$$[W_p]_{k,*} = \frac{[A|^p]_{i,*} - \sum_{l=1}^{k-1} [V_p]_{l,j} [W_p]_{l,*}}{\delta_W(i,j)} \tag{2.3}$$

なお ACA 法と ACA+法では，ピボット列 j とピボット行 i の選択法が異なり，また ACA 法では $\delta_V(i,j) = 1, \delta_W(i,k) = [A|^p]_{i,j}$ であるのに対し，ACA+法で $(\delta_V(i,j), \delta_W(i,j)) \in \{(1, [A|^p]_{i,j}), ([A|^p]_{i,j}.1)\}$ がピボット選択の結果に応じて定められる．

ここで重要なポイントは，V_p と W_p の構築には $A|^p$ の全体を必要とせず，ピボットとして選択される少数の列ベクトルと行ベクトルのみを必要とすることである．したがって $A|^p$ をあらかじめ用意する必要はなく，$[A|^p]_{i,j}$ を計算する関数を必要に応じて呼び出すだけでよい．また $\tilde{A}|^p$ のランク r_p は，$\|\cdot\|_F$ を行列の Flobenius norm とし，要求される近似誤差の上限を ε としたとき，

$$\frac{\|A - \tilde{A}\|_F}{\|A\|_F} \leq \varepsilon \tag{2.4}$$

が満たされるように定める必要があるが，この条件は，全ての $p \in P$ について以下が成り立つことが近似的に十分条件となっている．

$$\frac{\|[V_p]_{*,r_p}\|_2 \|[W_p]_{r_p,*}\|_2}{\sqrt{\sum_{k=1}^{r_p} (\|[V_p]_{*,k}\|_2 \|[W_p]_{k,*}\|_2\})^2} \leq \varepsilon \tag{2.5}$$

そのためこの条件を用いて近似誤差判定を行うことで，A や $A|^p$ の全体を参照することなく r_p を決定してフィル処理を完了することができる．

一方 $\tilde{A}|^p$ がフル小行列である場合には，$[A|^p]_{i,j}$ を計算する関数を $i \in S_p$, $j \in T_p$ なる全ての i と j について呼び出し，$\tilde{A}|^p = A|^p$ とすればよい．また一般に $[A]_{i,j}$ の計算は互いに独立であるため，$\tilde{A}|^p$ が近似小行列かフル小行列かに関わらず，フィル処理は小行列を単位として完全に並列化することができる．

2.3 $\mathcal{H}$ 行列ベクトル積のアルゴリズム

本節では，ベクトル $x \in \mathbb{R}^N$ と $\mathcal{H}$ 行列 $\tilde{A}$ が与えられた

とき，$\tilde{A}$ と x の積 $y = \tilde{A}x$ を求めるアルゴリズムについて述べる．$\mathcal{H}$ 行列ベクトル積を求める方法はいくつか考えられるが，本研究では葉行列ごとに部分行列ベクトル積の演算を行い，各演算で得られた部分結果を最後に加算して y を求める手法を用いる．まず，近似小行列 $\tilde{A}|^p = V_p W_p$ に対する部分行列ベクトル積について考える．はじめに，$W_p \in \mathbb{R}^{r_p \times T_p}$ とそれに対応する x の部分ベクトル $x|_{T_p}$ との乗算を行うと，要素数 r_p のベクトル

$$\hat{x}|^p_{r_p} = W_p \cdot x|_{T_p} \tag{2.6}$$

が得られる．次に，$V_p \in \mathbb{R}^{S_p \times T_p}$ と $\hat{x}|^p_{r_p}$ との乗算を行うことで，求める部分ベクトル積の結果である要素数 S_p のベクトル

$$\hat{y}|^p_{S_p} = V_p \cdot \hat{x}|^p_{r_p} \tag{2.7}$$

が得られる．一方フル小行列 $\tilde{A}|^p$ に対する行列ベクトル積については，通常の密行列ベクトル積の演算手順により，

$$\hat{y}|^p_{S_p} = \tilde{A}|^p \cdot x|_{T_p} \tag{2.8}$$

が得られる．ここで

$$[\zeta(v|_S, T)]_i = \begin{cases} [v|_S]_{i-\min S+\min T} & i \in S \\ 0 & i \notin S \end{cases} \tag{2.9}$$

を定義すると，全ての葉行列についての部分行列ベクトル積 $\hat{y}|^p_{S_p}$ を求めた後，$\zeta(\hat{y}|^p_{S_p}, [1, N])$ を全て足し合わせることにより積ベクトル

$$y = \sum_{p \in P} \zeta(\hat{y}|^p_{S_p}, [1, N]) \tag{2.10}$$

が得られる．

3. $\mathcal{H}$ 行列生成の並列実装

本章では，$\mathcal{H}$ 行列生成の並列実装について述べる．2.2 節で述べた通り，$\mathcal{H}$ 行列の生成は，行列の葉行列への区分け処理と，各葉行列のフィル処理から構成される．本研究では，$\mathcal{H}$ 行列行列生成の計算量の大部分を占めるフィル処理の並列化を考える．各葉行列のフィル処理は独立に実行できるため，各 OpenMP スレッドに処理すべき葉行列集合をタスク集合として割り当てることによってフィル処理全体のタスク並列化が可能であり，従来実装 [4][5] も提案実装もともにそのようなタスク並列化を行っている．従来実装では，各葉行列に対して計算量見積値を設定することで各葉行列のフィル処理にかかる時間を事前予測し，それに基づいて負荷が均衡するように各 MPI プロセスおよび各 OpenMP スレッドにタスクを静的に割り当てている．しかしこの手法では，行列の性質によっては MPI プロセスや OpenMP スレッドの間で負荷が均衡しない．そこで本研究では，各葉行列に対する計算量見積値を活用しつつ，動的タスク割当，タスク実行順序の変更，および大きなタスクの分割の各手法を導入することで，負荷均衡の改善を試みた．

3.1 従来手法による $\mathcal{H}$ 行列生成の並列実装

従来手法のタスク割当は以下の手順で行われる．

1. 各葉行列のフィル処理に対する計算量を見積もる．
2. 各プロセスに行の範囲を均等になるように割り当てる．
3. 0 番プロセスから順に葉行列の一番左上の要素が 1. で割り当てた行の範囲に含まれるような行列をそのプロセスに割り当てる．
4. 各 MPI プロセス内で各スレッドに均等に葉行列を割り当てる．

以下の各節では，上記の各手順について説明する．

3.1.1 計算量の見積

$\mathcal{H}$ 行列生成にかかる計算量は，葉行列要素数の総和 $N_P = \sum_{p \in P} N_p$ に比例する．ただし，N_p は $p \in P$ に対応する葉行列 $\tilde{A}|^p$ の行列要素数である．$\tilde{A}|^p$ が近似小行列であるとき，N_p は以下の式で与えられる．

$$N_p = r_p \cdot (\#S_p + \#T_p) \tag{3.1}$$

他方，$\tilde{A}|^p$ がフル小行列であるとき，N_p は以下の式で与えられる．

$$N_p = \#S_p \cdot \#T_p \tag{3.2}$$

式 (3.1) における N_p は，近似小行列を表現する低ランク行列のランク r_p がフィル処理完了まで判明しないため，事前に求めることはできない．そこで，r_p の見積値 r_{est} を導入することで，$\tilde{A}|^p$ が近似小行列である場合にそのフィル処理にかかる計算量 N_p^{est} の値を以下のように見積もる．

$$N_p^{\text{est}} = r_{\text{est}} \cdot (\#S_p + \#T_p) \tag{3.3}$$

一方，式 (3.2) の N_p はフィル処理の前に算出可能であるため，$\tilde{A}|^p$ がフル小行列である場合にそのフィル処理にかかる計算量見積値は $N_p^{\text{est}} = N_p = \#S_p \cdot \#T_p$ とする．

これらを用いると，全葉行列の計算量見積値の総和は $N_P^{\text{est}} = \sum_{p \in P} N_p^{\text{est}}$ となる．低ランク行列のランク数の見積値 r_{est} は全ての近似小行列に対して同じ値が設定される．したがって，実際には近似小行列ごとに異なる値をとる r_p と r_{est} との間には無視できないずれが生じる．また最適な見積値 r_{est} を見出すのは困難であるため，r_{est} は経験則により定められる．

3.1.2 MPI プロセスに対する静的タスク割当

行の範囲に基づく各 MPI プロセスへのタスク割当について説明する．いま，n_{proc} を MPI プロセス数，$P_\mu \subset P$ を μ 番（$0 \le \mu < n_{\text{proc}}$）MPI プロセスに割り当てられる

```
1: Procedure
2:   N^est_{P_μ} ← Σ_{p∈P_μ} N^est_p ; N̄^est_{P_μ} ← N^est_{P_μ}/n_thr
3:   S ← 0 ; ∀ν ∈ {0,...,n_thr − 1} : P_{μ,ν} ← ∅ ; ν ← 0
4:   Do k = 1, #Q_μ
5:     p ← the k-th element in Q_μ
6:     S ← S + N^est_p
7:     If S > (ν + 1)N̄^est_{P_μ} then
8:       ν ← ν + 1
9:     End If
10:    P_{μ,ν} ← P_{μ,ν} ∪ {p}
11:  End do
12: End-Procedure
```

図 2 μ 番プロセスに属する OpenMP スレッドへタスク集合割当を得るアルゴリズム

タスク集合とする．まず，各プロセスに対して行の範囲を均等になるように暫定的に割り当てる．

次に $\mu = 0$ から順に，葉行列を行単位に割り当てて行く．すなわち，ある行 i について 1 行目が i であるような葉行列，つまり $\min(S_p) = i$ となるような葉行列 $\tilde{A}|^p$ は，すべて特定のプロセスに割り当てられる．またこの行単位の割当は，上記の暫定的な行の割当を基準に行うが，割り当てられる葉行列の総計算量がプロセスあたりの計算量の平均値，つまり $N_P^{\mathrm{est}}/n_{\mathrm{proc}}$ からの乖離が許容範囲を超える場合には，暫定的に定めたプロセス間の行の境界を上下することで許容範囲に収まるようにする．

3.1.3 OpenMP スレッドに対する静的タスク割当

3.1.2 節の手法で各 MPI に割り当てられたタスクをさらにプロセス内の OpenMP スレッドに割り当てる手順について説明する．n_{thr} を各 MPI プロセス内の OpenMP スレッド数，$P_{\mu,\nu} \subset P_\mu$ を μ 番 MPI プロセスに属する ν 番 OpenMP スレッドに割り当てられるタスク集合とする．このとき，葉行列（タスク）集合 P の全要素を，位置に基づいてソートする．すなわち，小行列 $A|^p$（$p \in P$）の最も左上の要素が A の i_p 行 j_p 列に位置するとしたとき，$Ni_p + j_p$ の昇順にソートする．このソート結果であるタスク列をグローバルタスクキュー Q とする．さらに，3.1.2 節で述べた各 MPI プロセスへのタスク割り当てによって μ 番プロセスではローカルタスクキュー Q_μ を得る．従来実装では，各 OpenMP スレッドに割り当てる葉行列の計算量見積値の合計がなるべく均等になるようなタスク割当を葉行列数 $\#P$ の線形時間で実現するため，**図 2** のアルゴリズムによってタスク割当を行う．このアルゴリズムは，まず各 OpenMP スレッドに割り当てられるタスクの計算量見積値の平均 $\overline{N_{P_\mu}^{\mathrm{est}}} = N_{P_\mu}^{\mathrm{est}}/n_{\mathrm{thr}}$ を求める．次に，$\nu = 0, 1, \ldots, n_{\mathrm{thr}} - 1$ のぞれぞれについて順に，Q_μ からの取得済タスクの計算量見積値の合計 S が閾値 $(\nu+1)\overline{N_{P_\mu}^{\mathrm{est}}}$ を超える直前まで取得したタスクを ν 番目の OpenMP スレッドに割り当てるという作業を行う．

3.1.4 フィル処理

$P_\mu \subset P$ が割り当てられた μ 番 MPI プロセスは他 MPI プロセスと通信をすることなく，$p \in P_\mu$ に対応する葉行列のフィル処理を実行できる．同様にタスク集合 $P_{\mu,\nu} \subset P_\mu$ が割り当てられた ν 番 OpenMP スレッドも独立にフィル処理が実行できる．したがって，各 OpenMP スレッドが割り当てられた各葉行列に関するフィル処理を 2.2 節で示した手順で実行することで全体のフィル処理が完了する．

3.1.5 従来手法の問題点

従来手法の問題点として，主に以下の 2 点の理由により，十分な負荷均衡が得られないということが挙げられる．

(1) 実際に生成される低ランク行列のランク r_p は近似行列ごとに異なり，計算量見積値 N_p^{est} と実際の計算量 N_p の間に無視できないずれがある．さらに r_p の事前予測も困難である．従来実装では，r_{est} は全葉行列の r_p の平均値 $(\sum_{p\in P} r_p)/\#P$ に近い値を取るように経験的に定められている．しかし，問題の行列の性質によっては，実際の低ランク行列のランク値が近似小行列間で大きな偏差を持つこともある．このような偏差があると，r_{est} よりも大きなランクを持つ近似小行列のフィル処理の計算量は過剰に小さく見積もられ，その逆も生じる．そのため，各プロセス・スレッドに割り当てられるタスクの実際の負荷も見積値から大きくずれてしまう．

(2) 計算量 N_p がスレッドあたりの平均計算量 $(\sum_{p\in P} N_p)/(n_{\mathrm{proc}} \cdot n_{\mathrm{thr}})$ を上回る大きなタスクが存在する．そのため，並列化による性能向上が巨大な葉行列のフィル処理にかかる計算時間に律速されてしまう．

3.2 提案手法による $\mathcal{H}$ 行列生成の並列実装

本研究では，3.1.5 節で述べた問題点を解決し負荷均衡を改善するため，以下の手法を順に導入することで従来実装を改良した．

(1) 各 MPI プロセス内の OpenMP スレッドへのタスク割当処理への動的負荷分散の導入

(2) 計算量を考慮したタスク割当順序の変更

(3) 大きなタスクの複数スレッドによる並列処理

(4) MPI プロセスへのタスク割当処理への動的負荷分散の導入

以下の各節では，これらの各手法について説明する．

3.2.1 OpenMP のスレッドへのタスクの動的割当

OpenMP スレッドに対する動的タスク割当について説明する．MPI プロセス μ の各スレッドは，以下のような手続き OpenMP の動的スケジューリング機能を用いて実行することにより，動的なタスクの取得および実行を行う．

1. プロセス μ のタスクキュー Q_μ からタスクを c 個まとめて取得する．

2. 取得した c 個のタスク（葉行列）に対して順にフィル処理を実行する．

3. Q_μ が空でなければ 1. に戻る．

3.2.2 タスクの負荷を考慮した担当順序の変更

3.1.2 節や 3.1.3 節で述べたように，従来実装のタスク割当では，小行列の位置に基づいてタスクをソートし，その順序でタスクを割り当てている．しかし，タスクキューの最後の方に計算量が非常に大きなタスクがあった場合，動的タスク割当を行ったとしても，フィル処理の終盤でタスクキューにタスクが残っていないにも関わらず，あるOpenMP スレッドだけがその大きなタスクを実行しているという状況に陥り，負荷均衡が悪化する．そこで，タスク集合 P_μ を位置ではなく計算量見積値 N_p^{est} に基づいて降順ソートし，その結果をローカルタスクキュー Q_μ とすることで，この状況の回避を試みた．

3.2.3 大きなタスクの並列実行

3.1.5 節で述べたように，計算量が非常に大きいタスクが存在していると，それがボトルネックとなり，並列実行の性能が制限されてしまう．そこで，負荷均衡の阻害要因となりうる大きな計算見積値を持つタスクには MPI プロセス内の全スレッドを割り当てて並列に処理することで負荷均衡の改善を試みた．

α を大きなタスクの閾値を決めるパラメータとし，$N_p^{\mathrm{est}} > \alpha \cdot \overline{N_{P_\mu}^{\mathrm{est}}}$ なる関係を満たす計算量見積値 N_p^{est} をもつタスク p は「大きい」と判定する．大きいと判定されたタスクは，3.2.1 節で述べたタスク並列処理に先立って MPI プロセス内の全スレッドを使って並列に処理する．ただし，本研究で行った性能評価では，大きいと判定されたフル小行列は存在しなかったため，近似小行列のフィル処理についてのみ並列処理の実装を行い，フル小行列のフィル処理の並列実装は行っていない．

2.2 節で示した近似小行列 $\tilde{A}|^p$ のフィル処理は，要求近似精度を満たすまで V_p と W_p にベクトルを追加するループと，その中で式 (2.2) と (2.3) にしたがってベクトルの各要素を計算する 2 個のループの，二重ループ構造となっている．この外側ループには明らかにループ運搬依存があるため並列化はできないが，内側の要素計算ループは要素ごとに独立した計算を行うため完全に並列化することができる．本研究では，この並列処理を OpenMP の並列ループを用いて実装した．

大きなタスクの並列処理を採用した各プロセスのフィル処理全体の手順は以下のとおりである．

1. P_μ の全タスクを計算量見積値 N_p^{est} に基づいて降順ソートし，その結果をローカルタスクキューとする．

```
1: Procedure
2:   N^est_{P_μ} ← Σ_{p∈P_μ} N^est_p
3:   N̄^est_{P_μ} ← N^est_{P_μ} / n_thr
4:   P^div_μ ← ∅
5:   For each p ∈ P_μ do
6:     if p is an approximate submatrix and N^est_p > α N̄^est_{P_μ} then
7:       P^div_μ ← P^div_μ ∪ {p}
8:       Q_μ ← Q_μ − {p}
9:     End if
10:  End do
11:  For each p ∈ P^div_μ do
12:    fill Ã|^p using all threads in μ-th process
13:  End do
14:  // OpenMP dynamic scheduling
15:  !$OMP do schedule(dynamic, c)
16:  Do k = 1, #Q_μ
17:    p ← the k-th element in Q_μ
18:    fill Ã|^p
19:  End do
20:  !$OMP end do
21: End-Procedure
```

図 3 大きなタスクの並列処理を導入したフィル処理の疑似コード

2. ローカルタスクキューの先頭部分のうち，計算量見積値 N_p^{est} が $\alpha \cdot \overline{N_{P_\mu}^{\mathrm{est}}}$ より大きいタスクに関しては，μ 番 MPI プロセス内の全 OpenMP スレッドで並列処理する．

3. 2. の完了後，P_μ 内の残りのタスクを 3.2.1 節で述べたタスク並列処理により実行する．

この手順の疑似コードを**図 3** に示す．

なお，ここで導入したタスク内の並列処理のオーバーヘッドは，通常タスク間の並列処理のオーバーヘッドよりも大きい．したがってタスク内並列処理の導入は，負荷均衡を改善できる半面，処理のスループットを低下させ，また計算量見積値と実際の計算コストの乖離の要因にもなる．このオーバーヘッドはタスクのサイズ N_p が小さすぎるタスクに対しては相対的に大きくなってしまうため，「大きい」タスクの閾値を決めるパラメータ α を小さくしすぎるとフィル処理全体のスループットも低下する．そのため α の値は，タスク並列処理だけでは性能ボトルネックとなるようなタスクが「大きい」と判定されるような最大の値に設定されることが理想である．

3.2.4 MPI プロセスへのタスクの動的割当

ここまで 3.2 節では，OpemMP スレッドへのタスク割当に関してのみ動的割当を採用した実装について述べてきた．しかし，従来手法で採用されている MPI プロセスへの静的タスク割当では，プロセス間の負荷均衡のずれも決して無視できない．加えて，3.2.3 節で導入したタスク内並列処理によって，MPI プロセスに割当てられた全タスクの計算量見積値の合計とその MPI プロセスのフィル処理の実際の全計算コストとの間にさらなる乖離が生ずるが，この影響を事前に正確に予測するのは困難である．これらの

問題を解決し，さらなる負荷均衡の改善を得るためには，MPI プロセスレベルでも動的なタスク割当が必須である．しかし，全 MPI プロセスの各スレッドが 1 つのグローバルタスクキューからタスクを取得するような実装では，負荷均衡の代償として通信コストの増大を招き，結果として性能がかえって悪化してしまう．そこで，本研究では，各 MPI プロセスがグローバルタスクキューからまとまった数のタスクを取得し，それらを 3.2.1 節の動的タスク割当を採用したタスク並列処理によって実行するということを繰返す階層型マスタワーカ方式を採用した．また，この方式においても，3.2.2 節で述べた計算量見積値に基づくタスクの取得優先順位の設定や，3.2.3 節で述べた大きなタスクのスレッド並列処理も採用した．

具体的な処理手順を以下に示す．ただし，α は 3.2.3 節と同じく「大きな」タスクの閾値を決定するパラメータである．また β は MPI プロセスが一度に取得するタスクの量を決定するパラメータである．

1. タスク集合 P の全タスクを計算量見積値 N_p^{est} に基づいて降順ソートし，その結果を全 MPI プロセスが共有するグローバルタスクキュー Q とする．

2. Q の先頭部分のうち，計算量見積値 N_p^{est} が $\alpha(\sum_{p\in P} N_P^{\mathrm{est}})/(n_{\mathrm{thr}} n_{\mathrm{proc}})$ より大きいタスクを，各 MPI プロセスに対してサイクリックに割り当てる．

この後，各 MPI プロセスで以下の処理を行う．

3. 2. で自プロセスに割り当てられたそれぞれのタスクを，プロセス内の全 OpenMP スレッドを使って並列に処理する．

4. 計算量見積値の合計が β 以上となる最小の数のタスクをグローバルタスクキュー Q からまとめて取得し，得られたタスクをローカルタスクキュー Q_μ に追加する．μ 番プロセス内の各 OpenMP スレッドは，Q_μ からタスクを c 個ずつ取得し実行するということを Q_μ が空になるまで繰り返す．

5. 4. を Q が空になるまで繰り返す．

上記の手順のうち，3.–5. の各 MPI プロセス内の処理を**図 4** に示す．

本研究では，グローバルタスクキューの管理および各プロセスのキューからのタスク取得処理に関して，(1) プロセス 0 の 1 スレッドをキュー管理のために割り当て，各プロセスはそのスレッドとの MPI_Send，MPI_Recv を用いた通信によりタスクの要求や受け取りを行う実装，および (2) キューの実体をプロセス 0 に配置し，各プロセスは MPI_Win_lock，MPI_Fetch_and_op，MPI_Win_unlock によりキューの排他的ロックの獲得・解放や情報取得・更新を行う実装の 2 種類の実装を行った．

```
1: Procedure
2:   S ← 0
3:   While Q ≠ ∅
4:     For each p ∈ P_μ^div do
5:       fill Ã|^p using all threads in μ-th process
6:     End do
7:     Do k = 1, #Q
8:       lock Q
9:       p ← the k-th element in Q
10:      S ← S + N_p^est
11:      if S > β or k = #Q then
12:        pop k elements from Q and enqueue them into Q_μ
13:        break
14:      End if
15:    End do
16:    unlock Q
17:    // OpenMP dynamic scheduling
18:    !$OMP do schedule(dynamic, c)
19:    Do k = 1, #Q_μ
20:      p ← the k-th element in Q_μ
21:      fill Ã|^p
22:    End do
23:    !$OMP end do
24:    Q_μ ← ∅
25:  End While
26: End-Procedure
```

図 4 階層型マスタワーカ方式における各 MPI プロセスの処理

4. $\mathcal{H}$ 行列ベクトル積の並列実装

本章では，$\mathcal{H}$ 行列ベクトル積の並列実装について述べる．2.3 節で述べた通り $\mathcal{H}$ 行列ベクトル積は，各葉行列とそれに対応するベクトルとの乗算（式 (2.6)–(2.8) に相当），および葉行列ベクトル積の結果の縮約演算（式 (2.10) に相当）によって行われる．このうち葉行列ベクトル積の計算は，フィル処理と同様に葉行列ごとに独立に実行できるため，各 OpenMP スレッドに処理すべき葉行列集合をタスク集合として割り当てることによるタスク並列化が可能であり，従来実装，提案実装ともにそのような並列化を行っている．

しかし，$\mathcal{H}$ 行列ベクトル積ではフィル処理とは異なる以下の点について実装時に注意が必要である．

(1) 葉行列ベクトル積の計算を行うためには，担当するスレッドのプロセス内に前のフィル処理で生成された葉行列要素のデータが存在しなければならない．そのため，フィル処理と葉行列ベクトル積でプロセスの割当を変更する場合には，その要素データを転送するための追加コストがかかる．

(2) 近似小行列を表現する低ランク行列のランク数が確定しているため，ほぼ正確な葉行列ごとの計算量の見積もりが可能である．

(3) 葉行列ベクトル積のタスク割当は，その後の縮約演算の並列化可能性に影響する．

(1) の問題のため，$\mathcal{H}$ 行列ベクトル積においてプロセスレベルのタスク割当を動的に変更することは不適切だと考えられる．従来実装のプロセスレベルのタスク割当では，フィル処理時の割当をそのまま採用することとしており，本研究の実装でも同じ戦略を採用した．ただし提案実装のフィル処理では，プロセス間で負荷がより均等になるようにタスク割当を行ったため，生成された葉行列の要素数の合計も均等となり，$\mathcal{H}$ 行列ベクトル積のプロセス間の負荷均衡も改善が期待できる．

各プロセス内でのスレッドへのタスク割当は，フィル処理の割当から変更してもほとんどコストはかからない．そのため従来実装では，確定した低ランク行列のランク数を用いて算出し直した計算量見積値に基づいて負荷が均等になるようスレッドへのタスク割当を更新している．上記 (2) の通りこの際の見積値はほぼ正確だが，3.1.5 節で述べた「大きな」タスクが並列性能のボトルネックとなる問題はここでも残る．そこで本研究の実装では，3.2.2 節や 3.2.3 節で導入した見積値に基づくタスク割当順序の変更や，大きなタスクの並列処理をここでも導入した．動的タスク割当は，フィル処理時と比べると必要性は薄いが，実行時の外乱等による実行時間のずれを緩和できると期待されるため，本研究の実装では採用した．

上記 (3) の問題について考える．縮約演算の並列度を上げるためには，各スレッドが担当する葉行列の $\mathcal{H}$ 行列上の行の範囲の和集合が，スレッド間でなるべく重ならないようにする必要がある．従来実装ではこの重複がなるべく少なくなることを目指したタスク割当を行っており，縮約演算もそれを前提として実装されている．提案実装では，負荷の均衡化を第一目標としたタスク割当の結果，各スレッドが担当する葉行列の位置は $\mathcal{H}$ 行列中に分散し，重複範囲が大きくなってしまう．そのため，従来実装をそのまま利用すると，性能が著しく劣化する．そこで本研究では，この重複範囲が大きい場合の性能劣化がなるべく緩和されるよう，縮約演算の実装を変更した．

4.1 従来手法による $\mathcal{H}$ 行列ベクトル積の並列実装

4.1.1 計算量見積値の導入

葉行列 $\tilde{A}|^p$ に関する部分ベクトル積の計算量は，フィル処理と同様 $\tilde{A}|^p$ の要素数 N_p に比例する．ただし，近似小行列を表現する低ランク行列のランク数はフィル処理で確定しているため，3.1.1 節のフィル処理の計算量見積もりで用いた予測値 r_{est} は不要であり，全葉行列の要素数を正確に算出できる．

4.1.2 MPI プロセスに対するタスク割当

本章の冒頭で述べた通り，従来実装でのプロセスレベルのタスク割当では，タスク割当変更にともなう葉行列の要素データ移動のコストを避けるため，フィル処理時の割当をそのまま採用している．すなわち，3.1.2 節で設定した各プロセスへのタスク割当 $P_\mu \subset P$ $(0 \leq \mu < n_{\mathrm{proc}})$ を葉行列ベクトル積のタスク割当でもそのまま用いる．

4.1.3 OpenMP スレッドに対する静的タスク割当

従来手法における OpenMP スレッドへのタスク割当は，3.1.3 節で説明したフィル処理における静的タスク割当と同じアルゴリズムで行う．ただし，各タスクの計算量見積値には，4.1.1 節の方法で算出し直した値を用いる．

4.1.4 葉行列ベクトル積の縮約演算

従来実装における $\mathcal{H}$ 行列ベクトル積の部分結果ベクトルの縮約演算は，まず各 MPI プロセス内で全 OpenMP スレッドの部分結果を足し合わせ，さらにそれらの結果を全プロセスで足し合わせることによって行われる．

まず OpenMP スレッド間での縮約演算について述べる．いま，μ 番 MPI プロセスに属する ν 番 OpenMP スレッドが部分結果として持つベクトルの $\mathcal{H}$ 行列上の行の範囲は以下で定義される関数 $R(\mu,\nu)$ で定められる．

$$R(\mu,\nu) = \{i \mid \min_{p\in P_{\mu,\nu}}(\min S_p) \leq i \leq \max_{p\in P_{\mu,\nu}}(\max S_p)\} \tag{4.1}$$

ここで，各スレッドが自分に対して割り当てられた葉行列ベクトル積を実行することで得る不完全な解ベクトルを $\hat{y}|^{\mu,\nu}_{R(\mu,\nu)}$ とする．$\hat{y}|^{\mu,\nu}_{R(\mu,\nu)}$ は以下の計算で求められる．

$$\begin{aligned}\sum_{p\in P_{\mu\nu}} \zeta(\tilde{A}|^p \cdot x|^p_{T_p}, R(\mu,\nu)) &= \sum_{p\in P_{\mu,\nu}} \zeta(\hat{y}|^p_{S_p}, R(\mu,\nu)) \\ &= \hat{y}|^{\mu,\nu}_{R(\mu,\nu)}\end{aligned} \tag{4.2}$$

さらに各スレッドが算出したベクトルを加算することで，μ 番プロセス内の全スレッドに対する縮約結果 $\hat{y}|^{\mu}_{R(\mu)}$ を得る．ただし，$R(\mu) = \{i \mid \min_{\nu=0}^{n_{\mathrm{thr}}-1}(\min R(\mu,\nu)) \leq i \leq \max_{\nu=0}^{n_{\mathrm{thr}}-1}(\max R(\mu,\nu))\}$ である．

$$\hat{y}|^{\mu}_{R(\mu)} = \sum_{\nu=0}^{n_{\mathrm{thr}}-1} \zeta\left(\hat{y}|^{\mu,\nu}_{R(\mu,\nu)}, R(\mu)\right) \tag{4.3}$$

従来実装では，式 (4.2) の加算結果は，各スレッドが葉行列ベクトル積のタスク処理時に，その計算結果を $\hat{y}|^{\mu,\nu}_{R(\mu,\nu)}$ 用の配列に順次足し込むようにすることで自然に得られる．また，式 (4.3) の加算は，ν 番スレッドがベクトル $\hat{y}|^{\mu,\nu}_{R(\mu,\nu)}$ を $\hat{y}|^{\mu}_{R(\mu)}$ 用の配列に足しこむということを，全スレッドが並列に実行することで行われる．ただしこのとき，$R(\mu,\nu)$ が 2 つ以上のスレッドで重複する範囲のベクトル要素に関しては排他制御が必要である．従来実装では，$R(\mu,\nu)$ の重複の有無に関わらず，OpenMP の atomic ディレクティブを用いてこの排他制御を行っている．

次に，MPI プロセス間でのベクトル加算を行うことで，最終的に求める $\mathcal{H}$ 行列ベクトル積の値 y を得る．

$$y = \sum_{\mu=0}^{n_{\mathrm{proc}}-1} \zeta\left(\hat{y}|^{\mu}_{R(\mu)}, [1,N]\right) \tag{4.4}$$

従来実装では，式 (4.4) の加算は，各プロセスが自分よ

りランク番号が 1 大きい番号を持つプロセス（$(\mu-1)$ 番プロセスは 0 番プロセス）にベクトルを送信し，自分よりランク番号が 1 小さい番号を持つプロセス（0 番プロセスは $(\mu-1)$ 番プロセス）から受け取ったベクトルを足しこむ，という処理を $(\mu-1)$ ステップ繰り返すことで行われる．この結果，全プロセスがベクトル y の値を持つことになる．各ステップの MPI 通信コストは $\max_{\mu=0}^{n_{\mathrm{proc}}-1} \#R(\mu)$ で与えられる．3.1.2 節のプロセスへのタスク割当は，この値がなるべく N/n_{proc} に近くなるようにすることも目的の一つとしている．

4.2 提案手法による $\mathcal{H}$ 行列ベクトル積の並列実装

4.2.1 MPI プロセスに対するタスク割当

本章の冒頭で述べた通り，プロセスレベルのタスク割当では，提案実装でも動的割当を採用せず，従来実装と同じくフィル処理時の割当をそのまま採用することにした．すなわち，3.2.4 節のマスターワーカ方式の動的負荷分散の結果として得られたタスク割当を，葉行列ベクトル積でもそのまま採用する．

4.2.2 OpenMP スレッドに対する動的タスク割当

4.2.1 節の方針で各プロセスに割り当てられたタスクの処理については，3.2.1 節および 3.2.3 節と同様に，スレッドへの動的割当，および大きなタスクの並列処理の手法を適用する．

ただし，3.2.2 節に相当するタスク列のソートは，計算量見積値ではなく，3.1.3 節で説明した，葉行列の位置関係に基づいて行う．これは，同じ行に位置する複数の葉行列に関する葉行列ベクトル積演算はなるべく連続して行ったほうが，乗数ベクトル x へのアクセスに関する参照局所性が高くなると考えられるためである．

このようにソートされたタスク列のタスクのうち，計算量見積値が閾値を超えたものを「大きな」タスクと判定して先にスレッド並列処理で実行し，残りのタスクを OpenMP の動的スケジューリングを用いたタスク並列処理で実行する．ただし，計算量見積値は 4.1.1 節で述べた従来手法と同様に，フィル処理で確定した低ランク行列のランク値を用いて算出する．

大きなタスクに対応する葉行列に関する葉行列ベクトル積のスレッド並列処理，すなわち近似小行列 $\tilde{A}|^p = V_p \cdot W_p$ と，それに対応する x の部分ベクトル $x|_{T_p}$ の積

$$\tilde{A}|^p \cdot x|_{T_p} = V_p \cdot W_p \cdot x|_{T_p} \qquad (4.5)$$

の計算のスレッド並列化は，W_p と V_p をそれぞれ列方向，行方向で分割したものを各スレッドの担当領域とすることで行った．

4.2.3 葉行列ベクトル積の縮約演算

葉行列ベクトル積の縮約演算は，4.1.4 節で説明した従来実装をそのまま利用しても正しい結果を得ることができる．しかし，以下の理由により性能は低下してしまう．

(1) 4.2.2 節の動的タスク割当の結果，各 OpenMP スレッドが部分結果として持つベクトルの $\mathcal{H}$ 行列上の行の範囲 $R(\mu,\nu)$ のスレッド間の重複範囲は非常に大きい．そのため，スレッド間の縮約演算における排他制御のために用いる atomic ディレクティブのオーバーヘッドが大きくなってしまう．

(2) 4.2.1 節のプロセスへのタスク割当の結果，$R(\mu)$ のサイズは全プロセスで N に近い値になってしまう．そのため，プロセス間の縮約演算のための通信コストも大きくなる．

この問題を解決するため，以下の通りスレッド間およびプロセス間の縮約演算の実装を変更した．

まず，スレッド間の縮約演算については，$\hat{y}|^{\mu,\nu}_{R(\mu,\nu)}$ に関する縮約演算をそのまま ν 番スレッドが担当するのではなく，$R(\mu)$ を均等に分割して各スレッドの担当範囲とする並列化を行った．この並列化は排他制御を必要としない．

次に MPI プロセス間の縮約演算は，4.1.4 節で述べた $(\mu-1)$ ステップの通信ではなく，MPI_Allreduce を用いて実装した．MPI プロセスが木を構成するように接続され，その木の枝に沿って縮約演算が行われるような最適な MPI_Allreduce の実装が行われていたとすると，この実装の通信コストは $\mathcal{O}(N \log n_{\mathrm{proc}})$ になると期待される．

5. 性能評価

5.1 評価方法

3 章および 4 章で示した各実装を評価するため，京都大学学術情報メディアセンターのスーパーコンピュータ Laurel を用いて性能評価を行った．評価環境の詳細を**表 1** に示す．例題として，表面電荷法による解析で用いられる係数行列を $\mathcal{H}$ 行列として生成する処理，およびその行列に対する $\mathcal{H}$ 行列ベクトルの積の計算を行い，生成処理のうちフィル処理にかかる時間および $\mathcal{H}$ 行列ベクトル積の計算時間をそれぞれ評価した．評価に用いた $\mathcal{H}$ 行列の詳細を**表 2** に示す．

なお，以下の性能評価の実行時間には，独立に 5 回の時間計測を行った結果の中央値を採用している．

表 1 評価環境

	Appro Green Blade 8000
Processor	Xeon E5-2670 (2.6GHz, 8 cores × 2/node)
Memory	DDR3-1600 64GB/node
Network	InfiniBand FDR×2, Fat tree
OS	Red Hat Enterprise Linux Server 6.2 (Santiago)
Compiler	Intel Fortran Compiler 15.0 (-O3 最適化)
MPI	Intel MPI Library 5.0 (8 threads/process)

5.2 $\mathcal{H}$ 行列生成の性能評価

5.2.1 測定結果

$\mathcal{H}$ 行列生成に関する性能測定値を**表 3** に示す．また，この表のうち従来実装である S/S と最も良い性能が得られた D(1)/D の，性能向上率と平均性能向上率を**図 5** に図示する．

表 3 の記号やデータの意味は以下の通りである．実装はどの戦略に従って $\mathcal{H}$ 行列生成処理を行ったかを後述する記号によって示し，n_{proc} は MPI プロセス数，n_{thr} は MPI プロセスごとの OpenMP スレッド数を表している．実行時間は $\mathcal{H}$ 行列生成のフィル処理に要する時間，すなわち全プロセス・スレッドがフィル処理を完了するまでに要する時間である．一方，平均実行時間は各 OpenMP スレッドが実際にフィル処理をしている時間，すなわち他のスレッドやプロセスの完了を待つ時間を除いた実行時間の平均値であり，負荷が完全に均衡したと仮定した場合の実行時間を意味している．性能向上率および平均性能向上率は，各並列実装の実行時間および平均実行時間を逐次実行の時間で割った値である．平均ロック時間は MPI プロセスがタスク要求をした時に，タスク割当が行われるまでの平均待ち時間を表している．

次に各実装の記号 I_p/I_t は，I_p が MPI レベルでの負荷分散の方法を，I_t が OpenMP レベルでの負荷分散の方法を，それぞれ表している．I_p は S，D(1)，D(2) のいずれかであり，S は静的タスク割当を，また D(1) と D(2) は 3.2.4 節の (1) と (2) で述べた動的タスク割当を，それぞれ意味する．また I_t は S あるいは D であり，それぞれ静的タスク割当と動的タスク割当を意味し，後者は 3.2.1–3.2.3 節で述べた全ての手法を組み合わせたものである．

なお，静的タスク割当のパラメータは，3.1.1 節で述べた近似小行列のランク数の見積値 r_{est} を 7 に設定し，3.1.2 節で述べた負荷の平均値からの乖離の許容値は今回の評価では無制限とした．動的タスク割当のパラメータについては，3.2.1 節で述べた OpenMP スレッドによるタスク取得

表 2 評価に用いた $\mathcal{H}$ 行列の詳細

パラメータ	値
許容誤差 ε	2.0×10^{-5}
行列の次元	1188000
フル行列数	786880
近似小行列数	1086804
フル小行列の次元の分布 (min/ave/max)	3/7/11
近似小行列の次元の分布 (min/ave/max)	11/72/216000
フル小行列の要素数の分布 (min/ave/max)	9/73/550
近似小行列の要素数の分布 (min/ave/max)	48/1716/33199200
近似小行列のランクの分布 (min/ave/max)	2/7/53
密行列表現のメモリ量	10767700.195 MB
$\mathcal{H}$ 行列表現のメモリ量	14688.955 MB
$\mathcal{H}$ 行列表現のメモリ量/密行列表現のメモリ容量	0.136 %

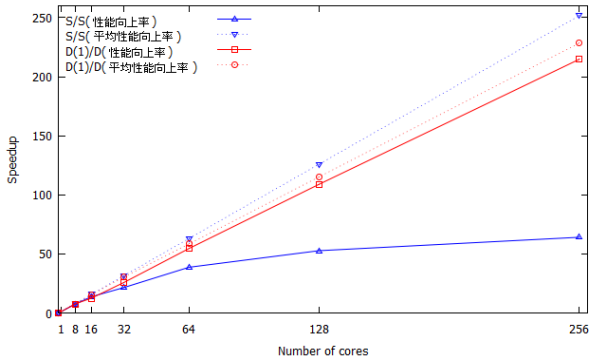

図 5 $\mathcal{H}$ 行列生成のフィル処理の S/S と D/D(1) の両実装の比較

単位 c を 1 に，3.2.3 節で述べた大きいタスクを判定する閾値 α を 0.1 に，それぞれ設定した．また 3.2.4 節で述べた MPI プロセスに一度に割り当てるタスク集合の総計算量である β は，集合内の全タスクの逐次実行時間が 1 秒程度となるように，予備実験に基づいて 959,806 と設定した．

5.2.2 S/S の結果

従来実装である S/S では，プロセス数が大きくなるに従い，性能向上率が急激に悪化している．表 3 で S/S の実行時間と平均実行時間の差がプロセス数の増加にともなって拡大していることから，スレッド間の負荷均衡の悪化が，十分な性能向上が得られない原因であると考えられる．各スレッドの負荷をより詳しく調べるため，S/S の 32 プロセス・8 スレッド実行時に各スレッドがフィル処理で生成した葉行列要素の総数を測定した．測定結果を**図 6** に示す．この図から，多くのスレッドが 7.5×10^6 個程度の要素しか生成していない一方で約 3.3×10^7 個という突出した数の要素を生成しているスレッドが存在し，負荷の不均衡が大きいことが確かめられる．

負荷不均衡の原因は 3.1.5 節で議論した通り，計算量見積値と実際の計算量のずれと，大きなタスクの存在であると考えられる．特にプロセス数を 16 から 32 に増やしたときに性能向上がほとんど得られていないのは，後者の要因が大きいと考えられる．**図 7** は，フィル処理の逐次実行時に，各タスクの処理に要する時間を測定した結果であるが，この図から 23.3 秒という長い処理時間を要するタスクが存在することがわかる．このことは，タスク分割を行わない限り，逐次実行時の実行時間 1479.3 秒を 23.3 秒で割った値である 63.5 倍程度以上の性能は決して得られないことを意味する．S/S の 32 プロセス実行では，このタスク並列化のみで得られる限界に近い性能向上率が得られているといえるが，これ以上の性能を得るためにはタスク分割は必須である．

5.2.3 S/D の結果

OpenMP レベルでの動的タスク割当とタスク分割を導入した S/D では S/S に比べると大幅に性能が改善されている．特に S/S と異なり，プロセス数を 8 から 16 に増加

表 3 $\mathcal{H}$ 行列生成のフィル処理の性能測定結果

実装	n_{proc}	n_{thr}	実行時間 [s]	平均実行時間 [s]	性能向上率 対逐次実行	平均性能向上率 対逐次実行	平均ロック時間 [s]
逐次実行	1	1	1479.3	—	—	—	—
S/S	1	8	190.0	188.5	7.4	7.8	—
	2	8	107.6	93.9	13.7	15.8	—
	4	8	68.8	47.0	21.5	31.5	—
	8	8	38.2	23.5	38.7	63.0	—
	16	8	28.1	11.7	52.7	126.0	—
	32	8	23.1	5.9	64.1	251.7	—
S/D	1	8	194.9	194.6	7.6	7.6	—
	2	8	101.3	97.1	14.6	15.2	—
	4	8	52.9	48.5	27.9	30.5	—
	8	8	28.6	25.3	51.8	58.5	—
	16	8	16.3	12.8	90.8	115.4	—
	32	8	11.2	6.4	131.9	229.6	—
D(1)/D	2	8	118.5	96.9	12.5	15.3	0.261×10^{-2}
	4	8	57.4	48.5	25.8	30.5	0.133×10^{-2}
	8	8	27.1	25.3	54.5	58.5	0.660×10^{-3}
	16	8	13.6	12.8	108.9	115.2	0.337×10^{-3}
	32	8	6.9	6.5	214.8	228.4	0.368×10^{-3}
D(2)/D	2	8	135.3	102.5	10.9	14.4	19.164
	4	8	59.2	50.3	25.0	29.4	1.339
	8	8	28.9	26.7	51.2	55.5	0.608
	16	8	15.1	13.7	97.9	108.1	0.850
	32	8	8.2	7.0	180.6	211.0	0.848

させたときにも大幅な性能向上が得られているが，これはタスク分割の効果が大きいと考えられる．しかし，たとえば 32 プロセス（計 256 スレッド）実行時の性能向上率は 131.9 倍であり，依然として十分な性能向上率が得られているとはいえない．平均性能向上率は 229.6 倍であることから，負荷の不均衡が性能律速の原因になっていると考えられる．

S/D の平均実行時間は S/S より少し悪化しているが，これは動的タスク割当のオーバーヘッドのほか，3.2.3 節で議論したタスクの分割処理よるスループットの低下が原因と考えられる．スループット低下の影響を調査するため，分割対象となった各タスクの 8 スレッドによる並列処理時間を測定し，図 7 の逐次処理時間と比較した．表 3 の測定で用いた $\alpha = 0.1$ の設定では，分割対象のうち最も小さなタスクの逐次処理時間は 0.9 秒であった．また，分割対象となったタスクの個数は全 $1,873,684$ 個中 58 個であり，分割対象となった全タスクの逐次処理時間の合計 203.7 秒に対し，同じタスクの 8 スレッドによる並列処理時間の合計は 36.9 秒であった．すなわち，タスクの並列処理による平均性能向上率は 5.5 倍（203.7 秒/36.9 秒）であり，理想的な値である 8 倍に対して 70%程度の効率となっている．この値は極端に低いものではなく，また分割対象のタスクの計算量が全体に占める割合が 1/7 程度であることからも，分割による負荷均衡度向上の効果がスループットの低下を大幅に上回るものであることが確認できた．

5.2.4 D(1)/D および D(2)/D の結果

MPI レベルでも動的タスク割当を導入した D(1)/D および D(2)/D では，S/D よりさらに性能向上率が改善された．特に D(1)/D では，32 プロセス（計 256 スレッド）実行時に 214.8 倍という理想に近い性能向上率が得られている．この結果は，主に負荷均衡の改善によって得られたと考えられる．実際，D(1)/D の実行で図 6 と同様に各スレッドが生成した葉行列要素数を示す**図 8** を見ると，全スレッドの負荷がほぼ理想的に均衡していることがわかる．また，平均実行時間も S/D からほとんど悪化していない．これは，プロセスレベルの動的タスク割当のオーバーヘッドをきわめて小さいことを示している．図 5 からも従来実装である S/S との性能差は明らかであり，特に 32 プロセス実行時では D(1)/D は S/S に比べて 3.4 倍の性能が得られた．

D(2)/D は 2 プロセス実行時を除いて，D(1)/D よりやや低い性能しか得られなかった．両者の性能差は，MPI プロセスによるグローバルタスクキューからのタスク取得にかかる時間（表 3 のロック時間）の違いによって生じたと考えられる．D(1)/D ではフィル処理を行うスレッドは 1 つ減らされているが，グローバルタスクキューからのタスク取得はより短時間で行えると考えられる．一方 D(2)/D では，全スレッドがフィル処理に参加できる半面，MPI_Win_lock による排他的ロックの取得に時間がかかってしまう．今回の測定では後者の悪影響のほうが大きかったと考えられる．

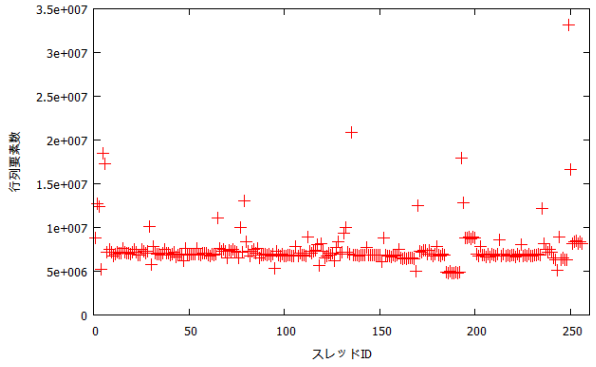

図 6　S/S の実行時（$n_{\mathrm{proc}} = 32, n_{\mathrm{thr}} = 8$）の各スレッドの生成行列要素数

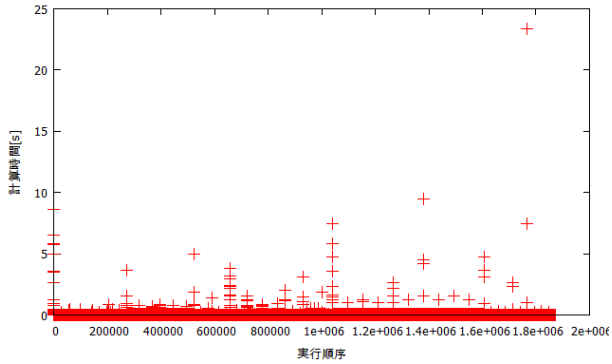

図 7　フィル処理の各タスクの逐次実行時間（横軸は S/S の 1 プロセス実行時のタスク実行順序）

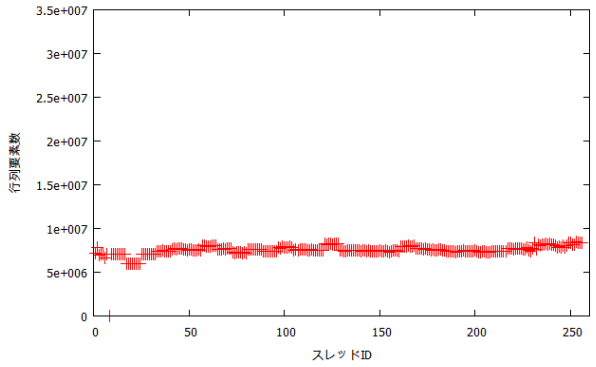

図 8　D(1)/D の実行時（$n_{\mathrm{proc}} = 32, n_{\mathrm{thr}} = 8$）の各スレッドの生成行列要素数

5.2.5　プロセスへのタスク割当結果

図 9 と図 10 に，S/S と D(1)/D の 32 プロセス実行時のプロセスへのタスク割当結果をそれぞれ示す．これらの図から，静的割当では同じ行にある葉行列はなるべく同じプロセスに割り当てられている一方，動的割当では各プロセスが担当する葉行列が $\mathcal{H}$ 行列全体に散らばっていることがわかる．

4 章冒頭や 4.2.3 節で議論した通り，この割当結果の違いは $\mathcal{H}$ 行列行列積の縮約演算の性能に影響する．

5.3　$\mathcal{H}$ 行列ベクトル積の性能評価

5.3.1　測定結果

$\mathcal{H}$ 行列ベクトル積に関する性能測定結果を表 4 に示す．

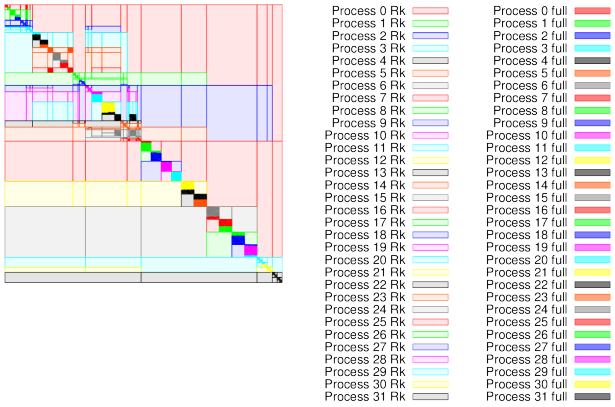

図 9　静的割当による各プロセスへのフィル処理のタスク割当結果（全 32 プロセス）

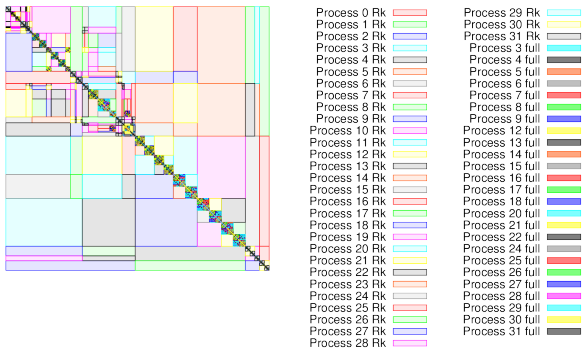

図 10　動的割当による各プロセスへのフィル処理のタスク割当結果（全 32 プロセス）

また，この表の結果のうち S[S]/S と S[D]/D の両実装の性能向上率を図 11 に図示する．表 4 の記号やデータの意味は以下の通りである．実装はどの戦略に従って $\mathcal{H}$ 行列行列ベクトル積を行ったかを後述する記号によって示し，n_{proc} は MPI プロセス数，n_{thr} は MPI プロセスごとの OpenMP スレッド数を表している．実行時間は $\mathcal{H}$ 行列ベクトル積を 100 回行うのに要した総計算時間を 100 で割った値である．性能向上率は，各並列実装の実行時間を逐次実行の実行時間で割った値である．葉行列ベクトル積実行時間，OpenMP スレッド間の縮約演算実行時間，および MPI プロセス間の縮約演算実行時間は，前述の総計算時間のうち各処理に要した時間を，それぞれ 100 で割った値である．ただし，葉行列ベクトル積と OpenMP スレッド間の縮約演算は各プロセスで独立に実行されているため，各処理に最も時間を要したプロセスに関する実行時間を示した [*1]．

次に各実装の記号の意味を説明する．S[S]/S は 4.1 節で示した従来実装，すなわち MPI プロセスへのタスク割当は S/S 実装によるフィル処理時の割当をそのまま採用し，OpenMP スレッドへのタスク割当は静的に行い，縮約演

[*1] そのため，実行時間は 3 つの内訳時間の和と必ずしも一致しない．

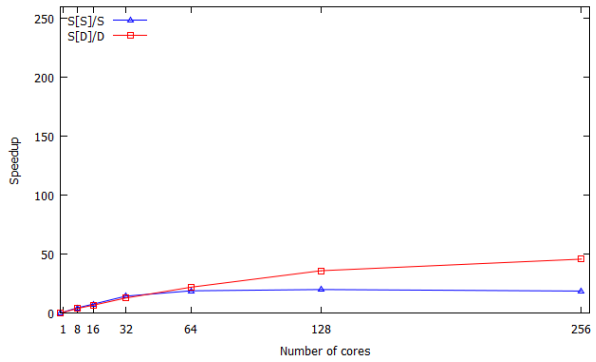

図 11　$\mathcal{H}$ 行列ベクトル積における S[S]/S と S[D]/D の性能向上率の比較

算には 4.1.4 節の手法を用いた実装を意味する．S[D]/D は 4.2 節で示した提案実装，すなわち MPI プロセスへのタスク割当は D(1)/D 実装によるフィル処理時の割当をそのまま採用し，OpenMP スレッドへのタスク割当では動的割当や大きなタスクの分割を行い，縮約演算には 4.2.3 節の手法を用いた実装を意味する．

なお，S[D]/D のスレッドレベルの動的タスク割当のタスク取得単位 c は 100 に設定した．

5.3.2　考察

表 4 と図 11 の結果から，8 プロセス以上による実行で S[D]/D は S[S]/S より良い性能が得られていることがわかる．特に 32 プロセス実行では，S[D]/D は S[S]/S に比べて 2.5 倍（= 108 ms/44 ms）の性能向上を達成できた．さらに 16 プロセス以上では，葉行列ベクトル積，縮約演算の両方で S[D]/D の性能が S[S]/S を上回っている．

大きいプロセス数での葉行列ベクトル積の両実装間の性能差の原因は，フィル処理と同じく，プロセス間およびスレッド間の負荷均衡の違いにあると考えらえる．まずプロセス間の負荷均衡を確かめるため，全プロセスに割り当てられた葉行列要素数の最大値および平均値を調査した．その結果を**表 5** に示す．この結果から，S[S]/S では平均に比べて極端に多い葉行列要素数を割り当てられたプロセスが存在する一方，S[D]/D では良好なプロセス間の負荷均衡が得られていることがわかる．これは，4 章冒頭で議論した通り，フィル処理でのプロセスレベルの動的タスク割当の結果を利用した恩恵によるものと考えられる．次に，スレッドレベルの負荷均衡を調べる．**図 12** と**図 13** は，それぞれ S[S]/S と S[D]/D による葉行列ベクトル積の実行時に，各スレッドに割り当てられた葉行列の要素数を示す．図 12 の傾向は S/S 実装によるフィル処理のタスク割当結果を示した図 6 とよく似ており，スレッドレベルでも負荷が均衡していないプロセスがあるほか，約 3.3×10^7 個という突出して多い要素を割り当てられたスレッドが存在することがわかる．このことから，葉行列ベクトル積でも，大きなタスクの存在が負荷均衡の悪化の原因になっていることが確かめられる．図 13 では各プロセス内のスレッドレベルの負荷はほぼ理想的に均衡している．これは，S[D]/D で導入した動的タスク割当と大きなタスクの分割の効果によるものと考えられる．

縮約演算でも S[S]/S があまり良好な性能が得られなかった原因の一つは，今回の評価で用いた $\mathcal{H}$ 行列の性質にあると考えられる．図 9 を見ると，$\mathcal{H}$ 行列を構成する葉行列の中に，列方向に極端に長いものがいくつか存在することがわかる．このような葉行列が存在すると，4.1.5 節で議論した $R(\mu,\nu)$ の重複範囲や $\max_{\mu=0}^{n_{\mathrm{proc}}-1} \#R(\mu)$ の値が大きくなってしまう．その結果，スレッド間の排他制御のコストやプロセス間の通信量が増大し，性能悪化を招いてしまったと考えられる．さらに，S[S]/S ではプロセス数にほぼ比例してプロセス間の縮約演算にかかる時間が増加しているが，これは部分結果の送信のための通信ステップ数が $(n_{\mathrm{proc}}-1)$ 回必要であることが直接影響したためと考えられる．S[D]/D では，スレッド間の縮約演算は排他制御を用いない並列化を行ったため，$R(\mu,\nu)$ の重複範囲に影響されない安定した性能が得られている．また，プロセス間の縮約演算では，各プロセスが必ず N 個のベクトル要素を送信する必要があるため，小さいプロセス数では S[S]/S より長い時間を要している．しかし，縮約通信に MPI_Allreduce 関数を利用したことによって，プロセス数の増加にともなう通信時間の増大は抑制され，その結果 16 プロセス以上では S[S]/S より短い時間で縮約演算を完了できている．

以上のように，提案実装である S[D]/D では従来実装である S[S]/S よりも大幅に優れた性能が得られているが，並列実行による性能向上率はさほど高くない．すなわち 32 プロセス（計 256 スレッド）実行の性能向上率は 45.7 倍であり，フィル処理の 214.8 倍に比べると 1/4 以下の値となっているが，これは以下の二つの理由によるものと考えられる．

第 1 の理由は，スレッド並列処理の効率がメモリバンド幅の制約により低く抑えられることである．たとえば 1 プロセス・8 スレッド実行の性能向上率は，フィル処理では 7.5 倍程度であるのに対し，行列ベクトル積では 4 倍程度の低い値となっている．これは，フィル処理では 1 個の行列要素を計算するのに多数の演算を必要とするのに対し，行列ベクトル積では行列要素あたり乗算と加算の 2 演算しか行われず，メモリアクセスの性能，特に再利用性が全くなくキャッシュの効果が期待できない行列要素に関するアクセス性能に，全体性能が支配されることによる．メモリアクセス性能は演算性能とは異なり，プロセッサ全体で一定の上限があるため，スレッド数を増やしても性能が向上しにくく，結果的に 8 スレッド実行での性能向上率が低く抑えられることとなる．

第 2 の理由は，プロセス数が増えるに連れて，プロセス間の縮約演算の時間が全体に占める割合が大きくなること

表 4　$\mathcal{H}$ 行列ベクトル積の性能測定結果

実装	n_{proc}	n_{thr}	実行時間 [ms]	性能向上率 対逐次実行	葉行列ベクトル積 実行時間 [ms]	スレッド間の縮約演算 実行時間 [ms]	プロセス間の縮約演算 実行時間 [ms]
逐次実行	1	1	2012	—	—	—	—
S[S]/S	1	8	465	4.3	459	6.1	—
	2	8	268	7.5	257	6.0	6.9
	4	8	141	14.3	127	6.0	8.4
	8	8	107	18.8	91	6.5	14.9
	16	8	101	19.9	64	6.3	31.9
	32	8	108	18.6	42	6.0	61.9
S[D]/D	1	8	536	3.8	534	2.5	—
	2	8	306	6.6	295	2.8	8.1
	4	8	158	12.7	144	2.8	11.3
	8	8	92	21.9	74	3.1	15.7
	16	8	56	35.9	40	3.8	13.5
	32	8	44	45.7	23	3.0	19.4

表 5　葉行列ベクトル積のプロセスレベルでのタスク割当結果（$n_{\mathrm{proc}} = 16$，$n_{\mathrm{thr}} = 8$）

実装	最大行列要素数	平均行列要素数	平均行列要素数/最大行列要素数
S[S]/S	99486183	60107407	0.604
S[D]/D	66753950	60107407	0.900

である．S[D]/D では，S[S]/S よりも縮約演算の時間を大幅に短縮し，またプロセス数の増加に伴う時間増も小さく抑えてはいるが，原理的にはプロセス数を増やすと必ず縮約演算時間は増加する．この結果，32 プロセスでの実行時間全体に占める縮約演算時間の割合は約 44%に達しており，並列計算性能を強く制約する要因となっている．

6. 関連研究

$\mathcal{H}$ 行列の生成・演算ライブラリには本論文で比較対象とした $\mathcal{H}$ACApK [4][5] のほか，Hlib [8] やその並列版実装である HLIBpro [9]，および AHMED [10] がある．HLIBpro の実装については，文献 [11] に $\mathcal{H}$ 行列生成，$\mathcal{H}$ 行列ベクトル積，$\mathcal{H}$ 行列積，および $\mathcal{H}$ 逆行列の各演算の共有メモリ環境向けの並列化手法，文献 [12] に $\mathcal{H}$ 行列に対する LU 分解の並列化手法が示されているが，$\mathcal{H}$ 行列生成や $\mathcal{H}$ 行列ベクトル積の分散環境における並列化手法を示した文献は見つからず，今後ソースコード等の調査を行う必要がある．

MPI プロセスレベルの動的負荷分散をマスターワーカ方式により実現する実装はいくつか存在する．たとえば文献 [13] では，巨大生体分子に対する第一原理電子状態計算を行うための FMO 法の計算のうち，高い計算コストを占める分子積分計算の MPI 並列化を行っているが，この際 MPI 間の負荷を均衡させるためマスタワーカ方式による動的負荷分散を採用している．この実装は，5.2 節の D(1)/D に近いが，タスクあたりの負荷が十分大きいため，一度に取得するタスク数は常に 1 とし，各タスクはプロセス内で

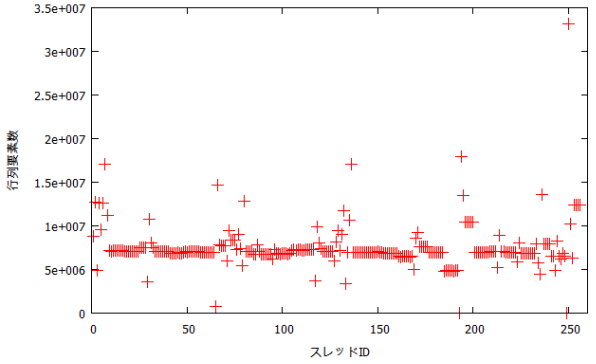

図 12　S[S]/S の実行時（$n_{\mathrm{proc}} = 32$，$n_{\mathrm{thr}} = 8$）の各スレッドの担当行列要素数

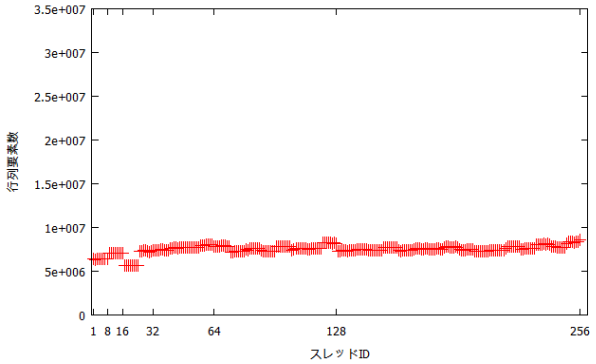

図 13　S[D]/D の実行時（$n_{\mathrm{proc}} = 32$，$n_{\mathrm{thr}} = 8$）の各スレッドの担当行列要素数

は静的負荷分散によりスレッド並列処理している．

7. 今後の課題

$\mathcal{H}$ 行列生成に関しては，本研究で行った最大 256 並列実行の評価では，ほぼ理想的な性能向上が得られた．しかし，今後スーパーコンピュータ等の並列計算機の規模がさらに大きくなり，そのような環境でさらに大きなサイズの $\mathcal{H}$ 行列を生成する際には，今回の提案実装では不十分にな

ることが想定される．

たとえば本研究では，$\mathcal{H}$ 行列生成処理のうち，フィル処理のみを並列化の対象としているが，並列度が上がっていくと，現時点では計算量の割合が小さいため無視できている区分けの決定処理の並列化も行わなければ，生成処理全体の性能向上が得られなくなると考えられる．

また，今日の並列計算機の性能向上は，各計算ノード内のコア数の増加より，ノード数の増加によって達成されることが多い．これは，MPI/OpenMP プログラム実行時のプロセス数が，プロセスあたりのスレッド数に対して増加していくことを意味する．このとき，本研究で提案した大きなタスクの分割をスレッドレベルで行うだけでは十分な分割にならず，プロセスレベルでもタスク分割を行うことも必要になると考えられる．

$\mathcal{H}$ 行列ベクトル積の性能改善も重要な課題である．5.3.2 節で述べたように，$\mathcal{H}$ 行列ベクトル積における性能の律速要因の一つはメモリアクセスにかかる時間である．このうち，乗数ベクトルや結果保存のためのメモリアクセスに関しては，タスクの処理順序等を工夫することで，キャッシュメモリの利用効率の改善による時間短縮が見込める．提案実装では負荷均衡の改善に主眼を置いた並列化を行っているが，今後は，負荷均衡とキャッシュの利用効率を両立できるようなタスク割当戦略を検討する必要がある．またもう一つの律速要因である縮約演算時間の短縮には，各プロセスが担当する積ベクトルの範囲やその重複をできるだけ小さくする必要があるが，この要求を満たしつつ負荷を均衡化することは極めて困難である．したがって一定の負荷不均衡を許容しつつ，プロセス数が増加が縮約演算時間の増加に繋がらないようにする，新たなタスク割当方式を検討する必要がある．

また本研究の性能評価では，表面電荷法に現れる 1 種類の入力データのみを用いた．しかし，本研究で提案した実装手法の効果は，$\mathcal{H}$ 行列の性質によって大きく変化すると考えられるため，今後は他の物理問題を含めた様々な入力データに本実装を適用し，評価を行う必要がある．

8. まとめ

本研究では，動的負荷分散を用いて $\mathcal{H}$ 行列生成および $\mathcal{H}$ 行列ベクトル積の並列化を行った．$\mathcal{H}$ 行列生成の従来実装では，各葉行列の生成コストが正確に見積もれないことや，コストが大きすぎるタスクの存在により，十分な並列性能が得られなかった．そこで提案実装では，動的タスク割当に加えて，計算量を考慮したタスク割当順序の変更，および大きなタスクの複数スレッドによる並列処理の各手法を導入することで，性能向上の阻害要因を取り除いた．$\mathcal{H}$ 行列ベクトル積では，通信コストの観点からプロセスレベルのタスクの動的割当は採用しなかったが，スレッドレベルの動的割当および大きなタスクの並列処理はここでも導入し，負荷均衡の改善を図った．また，タスク割当の変更に対応するため，縮約演算の実装も変更した．これらの結果，提案実装では $\mathcal{H}$ 行列生成，$\mathcal{H}$ 行列ベクトル積ともに良好な負荷均衡と従来実装を大幅に上回る性能を達成することができた．特に 256 コア実行では従来実装に対して，$\mathcal{H}$ 行列生成では 3.4 倍，$\mathcal{H}$ 行列ベクトル積では 2.5 倍の性能が得られた．

今後の課題としては 7 章で述べたように，より大規模な並列計算環境を指向した並列 $\mathcal{H}$ 行列生成の実装や，キャッシュの利用効率の向上や縮約演算時間の短縮を目的とした $\mathcal{H}$ 行列ベクトル積の実装改善などが，本研究をさらに発展させる上での重要なものとして挙げられる．

参考文献

[1] Börm, S., Grasedyck, L. and Hackbusch, W.: Hierarchical Matrices, Lecture note No. 21 of the Max Planck Institute for Mathematics in the Sciences (2003).
[2] Hackbusch, W. and Khoromskij, B. N.: A Sparse $\mathcal{H}$-Matrix Arithmetic. Part II: Application to Multidimensional Problems, *Computing*, Vol. 64, No. 1, pp. 21–47 (2000).
[3] Hackbusch, W.: A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-matrices, *Computing*, Vol. 62, No. 2, pp. 89–108 (1999).
[4] ppOpen-HPC Project: ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning. `http://ppopenhpc.cc.u-tokyo.ac.jp`.
[5] Ida, A., Iwashita, T., Mifune, T. and Takahashi, Y.: Parallel Hierarchical Matrices with Adaptive Cross Approximation on Symmetric Multiprocessing Clusters, *Journal of Information Processing*, Vol. 22, No. 4, pp. 642–650 (2014).
[6] Kurz, S., Rain, O. and Rjasanow, S.: The Adaptive Cross-Approximation Technique for the 3D Boundary-Element Method, *IEEE Transactions on Magnetics*, Vol. 38, No. 2, pp. 421–424 (2002).
[7] Bebendorf, M. and Rjasanow, S.: Adaptive Low-Rank Approximation of Collocation Matrices, *Computing*, Vol. 70, No. 1, pp. 1–24 (2003).
[8] Börm, S. and Grasedyck, L.: Hierarchical Matrices. `http://www.hlib.org`.
[9] Kriemann, R.: HLIBpro. `http://www.hlibpro.com`.
[10] Bebendorf, M.: Another Software Library on Hierarchical Matrices for Elliptic Differential Equations (AHMED). `http://bebendorf.ins.uni-bonn.de/AHMED.html`.
[11] Kriemann, R.: Parallel $\mathcal{H}$-Matrix Arithmetics on Shared Memory Systems, *Computing*, Vol. 74, No. 3, pp. 273–297 (2005).
[12] Grasedyck, L., Kriemann, R. and Le Borne, S.: Parallel Black Box $\mathcal{H}$-LU Preconditioning for Elliptic Boundary Value Problems, *Computing and Visualization in Science*, Vol. 11, No. 4-6, pp. 273–291 (2008).
[13] 稲富雄一，眞木　淳，本田宏明，高見利也，小林泰三，青柳　睦，南　一生：京コンピュータでの効率的な動作を目指した並列 FMO プログラム OpenFMO の高性能化，*Journal of Computer Chemistry, Japan*, Vol. 12, No. 2, pp. 145–155 (2013).