東京大学大学院新領域創成科学研究科
人間環境学専攻

2022 年度

修士論文

ヘテロジニアスなハードウェア環境における マルチフロンタル法

2023 年 2 月 7 日提出

指導教員　奥田　洋司　教授

学籍番号　47216674

河野　奏人

目次

図目次

第 1 章

序論

1.1 モデル化とシミュレーション

自然現象・物理現象を人間が解明するに当たっては，対象となる現象や物体の変化を，微分方程式を利用して記述しモデルを構築すること，すなわち**モデル化 (modeling)** を行う．[1] それらの微分方程式には，解析解を持つものが存在するものの，手計算でそれを得るのは容易でないことが多い．また，計算機により数値的に計算することで近似的な数値解を得るしかない場合も存在する．

計算機は有限個の離散的な値を扱う資源により構成されるため，時間・空間が連続であるものを正確に扱うことはできない．よって，微分方程式を解くに当たってはさらに**離散化 (discretization)** を行う必要がある．このとき，モデル化した複数の方程式は互いに依存するため，多元連立方程式を考えることになる．以上より，ある現象を記述したモデルは，離散化した連立方程式として，行列を用いた線形方程式の形にまとめられる．

この線形方程式を解くことで，変数となる要素の分布や動態を知ることができる．現象のシミュレーションとは，このように数学的な方程式の求解に基づくものである．計算機を利用し方程式を解くことで，基礎科学や工学，医学，社会科学などの様々なシミュレーションを行う分野を，**計算科学 (computational science)** と呼ぶ．計算科学には挙げたような無数の学問領域が含まれ，主眼は各領域での問題解決にあり，計算機における演算についてのみを探究している訳でない．その点で，計算機自体の研究を主とする**計算機科学 (computer science)** とは異なっている．

1.2 計算機の背景

高性能計算 (High Performance Computing; HPC) に用いる計算機では，階層的な並列化が必要とされる．[2] 処理装置と記憶装置の配置を中心としたアーキテクチャにより計算機を分類すると，共有メモリ型と分散メモリ型に分けられるが，前者における並列化例として OpenMP [3] の利用，後者における例として MPI [4] の利用が挙げられる．さらに，両者を合わせたハイブリッド並列も実際には用いられる．

計算の高速化という観点では，科学計算において，**GPU (Graphics Processing Unit)** の利用が進展きたことが特筆される．GPU は元々，ゲームなどにおける描画用の装置であったが，現在では用途が多様化している．[5] 実際，東京大学の Wisteria (2021-) [6] や同じく柏 II キャンパス内にある産業技術総合研究所の ABCI 2.0 (2021-) [7] といったスーパーコンピュータでも NVIDIA 社の GPU を搭載している．本研究で特に計算の題材とする，有限

要素法による構造解析へ利用した先行研究も存在する．[8]

1.3 ソルバの背景

ハードウェアの進歩により計算機が搭載するメモリも増加してきた．[9] すると，後述するように，マルチフロンタル法を含む線形方程式の直接法ソルバでは，反復法よりもメモリ消費が増大するものの，現在では対応可能になった問題が多くなったと結論付けられる．

ソルバそのものの進展としては，直接法の並列化に関する研究開発は反復法ほど進んでいない点を指摘できる．これは，反復法は係数行列の行ごとの処理が容易である一方で，直接法は依存関係を考慮しなくてはならないという特性による．実際，特に疎行列の直接法については，IBM の Watson Sparse Matrix Package (WSMP) [10]，Intel の Math Kernel Library Parallel Direct Sparse Solver Interface (MKL PARDISO) [11] や，MUltifrontal Massively Parallel sparse direct Solver (MUMPS) [12] といった，代表として挙げられる限られた既存ライブラリに依存している．

1.4 本研究の目的

計算科学で必要となる線形方程式の解法には，第 3 章で具体例を挙げて紹介するように，直接法と反復法の 2 種類が存在する．反復法はその演算の構成より並列化が行いやすく，多数のプロセッサを利用して効率良く求解することができる．また，構造解析に用いられる有限要素法をはじめとして，方程式に表れる係数行列が疎である場合に，メモリの消費量を少なく抑えることが可能である．しかしながら，いかなる線形方程式にも適用できる訳ではなく，反復法ではいつまでも解が得られないような行列が存在する．直接法は，これらのメリット・デメリットの裏返しを特徴として有し，並列化の難しさやメモリ消費の問題があるものの，より広い範囲で計算科学の問題に対応できる手法である．中でもマルチフロンタル法を利用した直接法は，第 4 章で詳しく述べる通り，並列化やメモリ効率について優れた実行を行う計算手法である．

したがって，本研究の目的は次のようにまとめられる．まず，直接法の適用可能性を重視し独自にマルチフロンタル法の実装を行うことで，一部の既存ライブラリに依存する現状を改善し，工学をはじめとする計算科学に寄与することを企図する．さらに，マルチフロンタル法の強みである並列化について実装を行い，数値実験によってその有効性を検証する．

1.5 本論文の構成

第 2 章では，構造解析の分野において解くべき線形方程式を構成する手法である有限要素法について概観する．第 3 章では，線形方程式の解法について，直接法による求解を詳述するとともに，反復法との比較を提示する．第 4 章では，線形方程式の係数行列が疎である場合に注目し，マルチフロンタル法を用いた直接法で求解する過程を説明する．第 5 章では，プログラムの並列化に用いられる規格である MPI について紹介する．第 6 章では，第 4 章で導入した直接法の実装について述べることに加え，第 5 章を踏まえたマルチフロンタル法の並列実装についても言及する．第 7 章では，実装したプログラムを利用した数値実験について記載する．

第 2 章

有限要素法

大学初年度の力学までは剛体を扱う一方で，構造物などの変形や熱伝導を考えると，実際には物体中において連続的に値が分布する．これを解析するには，物体を連続体 (continuum) として扱う必要がある．しかしながら，計算機はデジタル，すなわち離散的な値を扱うものであり，連続値をそのまま入出力できない．そこで用いられる方法の 1 つが**有限要素法 (finite element method; FEM)** であり，物体を有限個の要素に分割することで，各要素ごとに値の近似を行う． [13][14]

本章では 1 次元の物体について，1 次関数・高次関数それぞれを用いた有限要素法を概観する．また構造解析でよく用いられる要素について，実際に計算機に入力する場合に用いられる記法について紹介する．なお，以下では判別のしやすさの観点から，図形・次数を示す数字は全てアラビア数字で表記する．

2.1　1 次元の有限要素法

1 次元の対象として，ある長さ L をもつ棒状の物体を考え，これを複数の要素 (element) に分割する．このとき，要素の区切りとなる点及び端点を節点 (node) と呼ぶ．有限要素法においては，この要素ごとに区分的に定義された関数 [14] を用いることで，求める解の近似を行う．解析解を ϕ，ϕ の節点 m での値を ϕ_m とすると

$$\phi \simeq \sum_m \phi_m N_m \tag{2.1}$$

という近似を行える．ここで，N_m は節点 m で 1，他の節点では 0 となる関数である．今，変数として各節点での値を利用していることから，区分的に定義された関数としては，「ある要素上で 1」となる関数でなく，「ある節点上で 1」となる関数を利用する．

具体的に，1 次元の要素について，最も単純な 1 次関数で近似することを考える．変数変換を行うことで，着目しているある要素の座標を $-1 \leq \xi \leq 1$ に設定する．このとき，区分的に定義された関数として

$$\eta = -\frac{1}{2}(\xi - 1) \tag{2.2}$$

$$\eta = \frac{1}{2}(\xi + 1) \tag{2.3}$$

をとれる．それぞれ，$\xi = -1$ で 1，$\xi = 1$ で 0 となる関数と，$\xi = -1$ で 0，$\xi = 1$ で 1 となる関数である．

与えられた微分方程式を解くには，重み付き残差法を用いることで

$$\boldsymbol{K}\boldsymbol{\phi} = \boldsymbol{f} \tag{2.4}$$

の形の線形方程式を構築する．このとき，ある要素上で区分的に定義された関数は，定義より他の要素上で常に値が 0 であると見なせるので，全体の係数行列 $\boldsymbol{K}$ は，各要素 e 上で得られた $\boldsymbol{K}^e$ の単純な和となる．すなわち

$$\boldsymbol{K} = \sum_e \boldsymbol{K}^e \tag{2.5}$$

である．区分的に定義された関数は上の通り具体的に求めることができるので，$\boldsymbol{K}^e$ を求めることは容易である．$\boldsymbol{K}$ の構成に当たっては，各要素が端点とする節点ごとに行・列を合わせた上で $\boldsymbol{K}^e$ を足し合わせれば良い．これは**普通組み立て**と表現される． [14]

以上より，有限個の要素に区切ることは，連続的な分布を離散的な近似により表現できるという物理的な意味だけでなく，その上で区分的に定義される関数を用いることで，全体の線形方程式の係数 $\boldsymbol{K}$ が容易に求まるという利点が存在する．本論文で扱う線形方程式の求解では，全体の $\boldsymbol{K}$ を操作の対象とするが，これを組み立てず，要素ごとの $\boldsymbol{K}^e$ のみを扱うやり方も存在する．なお，ここまでは節点上での値という抽象的な表現を用いたが，構造解析においては変数として変位を設定する．変位について求解することにより，変位-ひずみ関係式からひずみを，さらに応力-ひずみ関係式から応力を求められる． [15]

さらに精度の良い，滑らかな解析解の近似を考えたい．すると 1 次関数では不十分なこともあるため，区分的に定義される関数を 2 次以上とする構成法を利用できる．ここでは，構造解析でも用いられる 2 次の形状関数のみ見ることにする．先ほどど同じ局所座標系の下では，端点以外に要素の中心 $\xi = 0$ にも節点を設けることで

$$\eta = \frac{1}{2}\xi(\xi - 1) \tag{2.6}$$

$$\eta = -(\xi + 1)(\xi - 1) \tag{2.7}$$

$$\eta = \frac{1}{2}\xi(\xi + 1) \tag{2.8}$$

の 3 つの関数を用意できる．それぞれ，$\xi = -1, 0, 1$ で 1 をとり，他の 2 節点で 0 となる，要素上のみで区分的に定義された関数である．1 次の形状関数の場合と同様に，$\boldsymbol{K}$ は各 $\boldsymbol{K}^e$ の和で求めることができるので，ここで行うことは，重み付き残差法により $\boldsymbol{K}^e$ を求める際に利用する関数を 2 次の形状関数に置き換えるのみである．

以上，1 次元要素における有限要素法の概略について，1 次・2 次の形状関数の構成を中心に紹介した．2 次元要素においては，1 次元における近似を拡張し，要素 (3 角形や 4 角形) のある 1 節点で 1，他の節点で 0 となる x, y の関数を設定すれば良い．3 次元についても，同様に変数を 1 つ増やすことで要素上での関数を定義できる．

2.2　3 次元における 4 面体 2 次要素

4 面体 2 次要素とは，1 つの要素が 4 面体であり，かつ 2 次関数による近似を行うものである．すなわち，1 要素は 3 角形による 4 面から構成され，また上記で見た通り各辺について端点・中点を考慮する必要があるため，1 つの要素につき必要な節点数は 10 となる．

ここでは, 実際に構造解析ソフトウェアにて用いられる 4 面体 2 次要素の表記法について紹介する．FrontISTR [16] では，4 面体 2 次要素を「342」と表記し，メッシュファイル中において図 2.1 のように定義を行う．

ここで，`!NODE` に続く行は各節点の番号及び x, y, z それぞれの座標を示している．また，`!ELEMENT` に続く行は各要素の番号及び，その要素を構成する 10 節点の節点番号を示している．

```
!NODE
[node1], [node1x], [node1y], [node1z]
[node2], [node2x], [node2y], [node2z]
...
!ELEMENT, TYPE=342
[elem1], [node1.1], [node1.2], [node1.3], ..., [node1.10]
[elem2], [node2.1], [node2.2], [node2.3], ..., [node2.10]
...
```

図 2.1　FrontISTR における 4 面体 2 次要素の記法

```
** Nodes
*Node, NSET=[node set]
[node1], [node1x], [node1y], [node1z]
[node2], [node2x], [node2y], [node2z]
...
** Volume elements
*Element, TYPE=C3D10, ELSET=[elem set]
[elem1], [node1.1], [node1.2], [node1.3], ..., [node1.10]
[elem2], [node2.1], [node2.2], [node2.3], ..., [node2.10]
...
```

図 2.2　Abaqus における 4 面体 2 次要素の記法

Abaqus [17] では「C3D10」と表記し，メッシュファイル (inp ファイル) 中で図 2.2 のように定義を行う．[18] 節点・要素の定義における記法は，図 2.1 の場合と統一している．

第 3 章

線形方程式の数値解法

第 2 章で見た有限要素法をはじめとして，自然界の現象をシミュレーションするに当たり，現象を支配する法則を何らかの形で連立 1 次方程式に落とし込む手法が用いられる．大学初年次に導入されるように，連立 1 次方程式は行列・ベクトルの積の形で書き表すことができるが，高次元であれば人の手で解くことは困難である．そこで計算機を用いて解となるベクトルを求める方法が必要となる．[*1]

3.1 概要

解くべき連立 1 次方程式として

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{3.1}$$

を対象とする．ここで係数行列 $\boldsymbol{A}$ は n 次正方行列 ($\boldsymbol{A} \in \mathbb{R}^{n\times n}$)，右辺ベクトル $\boldsymbol{b}$ は n 次元ベクトルで，いずれも離散化したモデルの条件から与えられるため既知であるとする．これを解いて，解となる n 次元ベクトルである $\boldsymbol{x}$ を求める．

数学的には，$\boldsymbol{A}$ が正則であれば

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{3.2}$$

と $\boldsymbol{A}$ の逆行列を利用すれば求解できる．しかし，高次元になってくると，計算機であっても逆行列を計算するのは現実的でない．そこで

(1) 数学的な行列の操作を組み合わせることで，逆行列を利用しなくても解が求まる形とする方法

(2) $\boldsymbol{x}$ の近似解を代入し，真の解との誤差を利用して解を更新することにより真の解に収束させていく方法

の 2 つが主に利用される．

[*1] 行列とベクトルは積を計算できるものでなく，$n \times n$ 行列と $n \times 1$ 行列の積とするのが正確という意見も存在する．しかしながら，$n \times 1$ 行列を指して n 次元縦ベクトルと称しても誤解はない上，計算科学においては「行列ベクトル積」という用語が一般的に用いられるため，本論文では行列とベクトルから積を計算できるものとして扱う．

3.2 直接法

(1) のように行列の操作を利用する方法は，近似解を利用せず真の解を直接求めることから，**直接法 (direct method)** や**直接解法**と呼ばれる．直接法にも種類があるが，ここでは $\boldsymbol{A}$ を行列の積の形に分解する手法を導入する．[*2]具体的に用いられるのが，**LU 分解 (LU decomposition / LU factorization)** と呼ばれるやり方である．[*3] これは係数行列 $\boldsymbol{A}$ を下三角行列 (lower triangular matrix) $\boldsymbol{L}$ と上三角行列 (upper triangular matrix) $\boldsymbol{U}$ の積に分解するもので，L, U はそれぞれの三角行列の名称から頭文字をとったものである．式で表すと (3.3) となる．

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U} \tag{3.3}$$

なお，上 (下) 三角行列という呼び方は対角成分まで非零であるものと対角成分が 0 であるもののいずれを指すこともあるが，LU 分解においては前者が該当する．逆に後者について特に言及する場合，本論文では一貫して狭義上 (下) 三角行列 (strictly upper/lower triangular matrix) という名称を用いることとする．

3.2.1 LU 分解

LU 分解は，計算手法の違いにより以下の 3 つに分類される．ここでは，それぞれの手法についてアルゴリズムを紹介する．

3.2.1.1 外積形式 (outer-product form)

$\boldsymbol{L}$ の対角成分が全て 1 であるとして分解を行う手法である．すなわち

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{bmatrix} \tag{3.4}$$

という形の分解を行う．

まず第 1 行に着目すると，$a_{1,j} = 1 \cdot u_{1,j}$ であることから

$$u_{1,j} = a_{1,j} \tag{3.5}$$

と値が求まる．次に第 1 列に着目すると，$a_{i,1} = l_{i,1}u_{1,1}$ であることから

$$l_{i,1} = \frac{a_{i,1}}{u_{1,1}} \tag{3.6}$$

と値が求まる．

第 2 行は，$j \geq 2$ について $a_{2,j} = l_{2,1}u_{1,j} + u_{2,j}$ より

$$u_{2,j} = a_{2,j} - l_{2,1}u_{1,j} \tag{3.7}$$

[*2] 他の手法としては，行基本変形を施すことにより第 2 行以降の係数を削減する**ガウスの消去法**が代表的である．
[*3] 反復法の前処理に用いられる不完全 LU 分解 (imcomplete LU factorization; ILU) と区別して，頭に「完全」をつけることもある．

Algorithm 1 Outer-product Form LU Factorization

```
for k = 0 to n − 1 do
    akkinv = 1.0 / a_{k,k}
    for i = k + 1 to n − 1 do
        a_{i,k} = a_{i,k} × akkinv
        aik = a_{i,k}
        for j = k + 1 to n − 1 do
            a_{i,j} = a_{i,j} − aik × a_{k,j}
        end for
    end for
end for
```

図 3.1　外積形式の LU 分解アルゴリズム

を得る．また第 2 列は，$i > 2$ について $a_{i,2} = l_{i,1}u_{1,2} + l_{i,2}u_{2,2}$ より

$$l_{i,2} = \frac{a_{i,2} - l_{i,1}u_{1,2}}{u_{2,2}} \tag{3.8}$$

を得る．式 (3.7), (3.8) においては右辺のいずれの成分も既知であるので求められる．以下同様に，分解するインデックスとして k に着目したとき，まず第 k 行の成分が全て求まり，次いで第 k 列の成分が求まる．第 k 行・第 k 列についての式はそれぞれ

$$u_{k,j} = a_{k,j} - \sum_{s=1}^{k-1} l_{k,s}u_{s,j} \quad (j \geq k) \tag{3.9}$$

$$l_{i,k} = \frac{a_{i,k} - \sum_{s=1}^{k-1} l_{i,s}u_{s,k}}{u_{k,k}} \quad (i > k) \tag{3.10}$$

である．

数学的には以上のように書けるが，実際には各成分の決定の度に式 (3.9), (3.10) に表れる総和を計算すると計算量が膨大になってしまう．また，解のベクトルを求めるに当たっては，分解後の $\boldsymbol{L}, \boldsymbol{U}$ のみ保存されていれば良く，元の $\boldsymbol{A}$ の値は分解後には必要ないため，計算機上で $\boldsymbol{A}$ の値に $\boldsymbol{L}, \boldsymbol{U}$ の値を上書きするアルゴリズムが採用される．このとき，$\boldsymbol{L}$ の対角成分は 1 であると仮定されているため，対角成分より下の部分を $\boldsymbol{L}$ で，対角成分とそれより上の部分を $\boldsymbol{U}$ で上書きする．すなわち

$$a_{i,j} \leftarrow \begin{cases} u_{i,j} & i \leq j \\ l_{i,j} & i > j \end{cases} \tag{3.11}$$

と上書きする．以上を踏まえた上での外積形式のアルゴリズムは図 3.1 に示す通りとなる．

なお，図 3.1 のアルゴリズムも含め，本論文では C 言語における 2 次元配列の格納を想定し，外側のループを行について，内側のループを列について回す形で記載する．Fortran 言語のように行優先で格納されている場合には，2 重ループの内外を入れ替えた方がデータへのアクセス効率が良くなる．

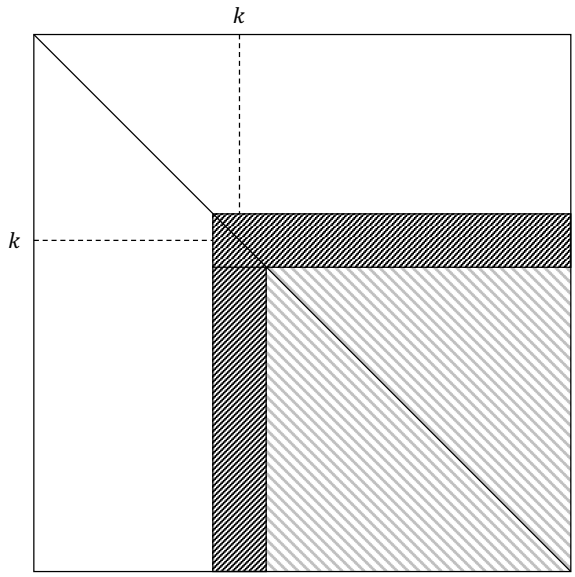

図 3.2　外積形式の LU 分解における更新イメージ．細い右上がりの黒線で塗られた領域が参照領域，太い右下がりの灰色線で塗られた領域が更新領域である．

アルゴリズムから分かる通り，分解するインデックス k に着目したとき，前のループで既に第 k 行の成分が求まっており，まず $u_{k,k}$ の逆数を用いて第 k 列の成分を確定させている．次いで，第 k 行・第 k 列の成分を参照し，インデックスが $k+1$ 以降の全成分について更新を行っている (図 3.2) ．第 k 行・第 k 列の成分からなるベクトルの外積 (直積) が更新に利用されるため，外積形式という名称である．[*4] このように，着目しているインデックス k の右側を見て更新操作を行うことから，外積形式を **right-looking algorithm** という．

3.2.1.2　内積形式 (inner-product form)

外積形式と同じく $\boldsymbol{L}$ の対角成分が全て 1 であるとして分解を行うが，参照・更新領域が異なる．外積形式においては，$\boldsymbol{U}$ の第 k 行と $\boldsymbol{L}$ の第 k 列を交互に定めていったが，内積形式では，第 k 列について $\boldsymbol{U}, \boldsymbol{L}$ ともに値を定めていく．まず第 1 列の $\boldsymbol{U}$ の成分は $u_{1,1}$ であり，これは $u_{1,1} = a_{1,1}$ と求まる．また $\boldsymbol{L}$ の成分 $l_{i,1}$ は外積形式の場合と同様に式 (3.6) によって求められる．

第 2 列の $\boldsymbol{U}$ の成分を行のインデックスが小さい方から見ていくと，まず $u_{1,2} = a_{1,2}$ より $u_{1,2}$ が求まる．次に $u_{2,2}$ であるが，式 (3.7) を参照すれば，$u_{2,2} = a_{2,2} - l_{2,1}u_{1,2}$ と計算できることが分かる．このとき，決定したばかりの $u_{1,2}$ を参照している．また $\boldsymbol{L}$ の成分は式 (3.8) から求められるが，$\boldsymbol{L}$ の第 1 列の値だけでなく，同じ第 2 列の $u_{1,2}, u_{2,2}$ を参照している．

以下同様にして，第 k 列について値を定める場合には，$u_{1,k}$ から $u_{i-1,k}$ を参照して $u_{i,k}$ $(i > 1)$ を更新する．さら

[*4] クロス積 (cross product) $\boldsymbol{a} \times \boldsymbol{b}$ のことではない．

Algorithm 2 Inner-product Form LU Factorization

1: **for** $k = 0$ to $n-1$ **do**
2: **for** $j = 0$ to $k-1$ **do**
3: ajk $= a_{j,k}$
4: **for** $i = j+1$ to $n-1$ **do**
5: $a_{i,k} = a_{i,k} \times$ ajk
6: **end for**
7: **end for**
8: $a_{k,k} = 1.0/a_{k,k}$
9: **for** $i = k+1$ to $n-1$ **do**
10: $a_{i,j} = a_{i,k} \times a_{k,k}$
11: **end for**
12: **end for**

図 3.3　内積形式の LU 分解アルゴリズム

に，$u_{1,k}$ から $u_{k,k}$ までと $l_{i,\bullet}$ $(i > k)$ の全てを参照することで $l_{i,k}$ を更新する．よって第 k 列についての式は

$$u_{1,k} = a_{1,k} \tag{3.12}$$

$$u_{i,k} = a_{i,k} - \sum_{s=1}^{i-1} l_{i,s} u_{s,k} \quad (1 < i \leq k) \tag{3.13}$$

$$l_{i,k} = \frac{a_{i,k} - \sum_{s=1}^{k-1} l_{i,s} u_{s,k}}{u_{k,k}} \quad (i > k) \tag{3.14}$$

である．以上を踏まえた上での内積形式のアルゴリズムは図 3.3 に示す通りとなる．

着目しているインデックス k の左側を見て[*5]更新操作を行うことから，**left-looking algorithm** という．

3.2.1.3　クラウト法

$\boldsymbol{U}$ の対角成分が全て 1 であるとして分解を行う．外積形式とは逆に，$\boldsymbol{L}$ の第 k 列を定め，次いで $\boldsymbol{U}$ の第 k 行を決定していくのが大まかな流れである．ただしこのとき，第 k 行より上の $\boldsymbol{U}$ および第 k 列より左の $\boldsymbol{L}$ の成分を参照していく必要がある．アルゴリズムは図 3.5 に示す通りとなる． [19]

3.2.2　コレスキー分解

係数行列 $\boldsymbol{A}$ が対称な場合，下三角行列 $\boldsymbol{L}$ と上三角行列 $\boldsymbol{U}$ に分けるのではなく，下三角行列 $\boldsymbol{L}$ とその転置 $\boldsymbol{L}^\top$ に分解する方法を採用できる．

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^\top \tag{3.15}$$

[*5] 外積形式で見ている右側領域は更新対象であるのに対し，内積形式で見ている左側領域は参照対象であるという違いは存在する．

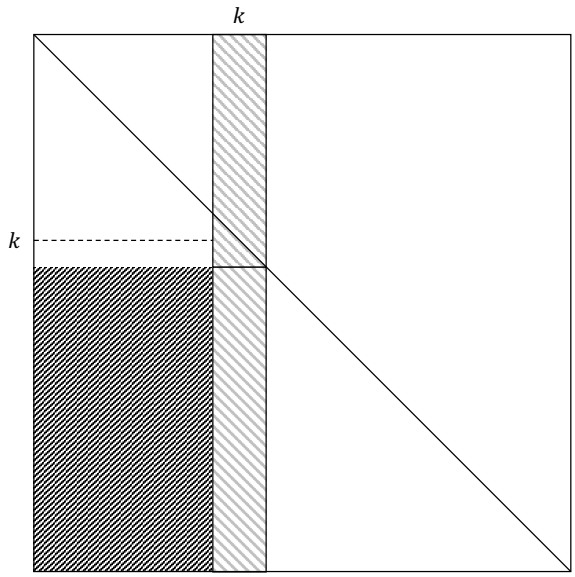

図 3.4　内積形式の LU 分解における更新イメージ．細い右上がりの黒線で塗られた領域が参照領域，太い右下がりの灰色線で塗られた領域が更新領域である．ただし $u_{\bullet,k}$ については更新・参照いずれの役割も果たす．

これを**コレスキー分解 (Cholesky factorization)** と呼ぶ．行列要素を書き下すと

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} = \begin{bmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{bmatrix} \begin{bmatrix} l_{1,1} & l_{2,1} & \cdots & l_{n,1} \\ 0 & l_{2,2} & \cdots & l_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{n,n} \end{bmatrix} \tag{3.16}$$

となる．$\boldsymbol{L}^\top$ の第 1 行に着目すると，$l_{1,1} = \sqrt{a_{1,1}}$，　$l_{k,1} = a_{1,k} \ / \ l_{1,1}$ となる．次に第 2 行に着目すると $l_{2,2} = \sqrt{a_{2,2} - l_{2,1}^2}$，　$l_{k,2} = (a_{2,k} - l_{k,1} l_{2,1}) \ / \ l_{2,2}$ と定められる．以下，同様に繰り返していけば良い．

3.2.3　修正コレスキー分解

コレスキー分解においては，対角成分の平方根による除算が発生していた．よって，対角成分が 0 に近い場合，この演算は失敗する可能性がある．これを避けるために利用されるのが**修正コレスキー分解**であり，以下のように定式化される．

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^\top \tag{3.17}$$

ここで $\boldsymbol{L}$ は狭義下三角行列であり，対角成分は 0 である．$\boldsymbol{D}$ は対角行列 (diagonal matrix) であり，非対角成分は全て 0 である．式 (3.17) の見た目よりより LDLT 分解とも表記される．

3.2.4　前進後退代入

計算手法の違いはあるが，ここまでの過程で式 (3.3) の分解の形が得られた．この分解を利用して解のベクトル $\boldsymbol{x}$ を求める．ここで，新たな n 次元ベクトル $\boldsymbol{y}$ を導入し

$$\boldsymbol{y} = \boldsymbol{U}\boldsymbol{x} \tag{3.18}$$

Algorithm 3 Crout method LU Factorization

```
a_{0,0} = 1.0 / a_{0,0}
for j = 1 to n − 1 do
    a_{0,j} = a_{0,j} × a_{0,0}
end for
for k = 0 to n − 1 do
    for j = 0 to k − 1 do
        ajk = a_{j,k}
        for i = k to n − 1 do
            a_{i,k} = a_{i,k} − a_{i,j} × ajk
        end for
    end for
    a_{k,k} = 1.0/a_{k,k}
    for i = 0 to k − 1 do
        aki = a_{k,i}
        for j = k + 1 to n − 1 do
            a_{k,j} = a_{k,j} − aki × a_{i,j}
        end for
    end for
    for j = k + 1 to n − 1 do
        a_{k,j} = a_{k,j} × a_{k,k}
    end for
end for
```

図 3.5　クラウト法の LU 分解アルゴリズム

と定める．これを元の式 (3.1) に代入すると

$$\boldsymbol{Ax} = \boldsymbol{LUx} = \boldsymbol{Ly} = \boldsymbol{b} \tag{3.19}$$

となる．

第 1 段階では

$$\boldsymbol{Ly} = \boldsymbol{b} \tag{3.20}$$

から $\boldsymbol{y}$ を計算する．このときインデックスの小さい行から大きい行に向かって値を定め代入していくため，**前進代入 (forward substitution)** と呼ぶ．[*6]

[*6] 似た単語に**前進消去 (forward elimination)** があり，次に述べる後退代入とセットで用いられる．ただし，「前進消去後退代入」とは，

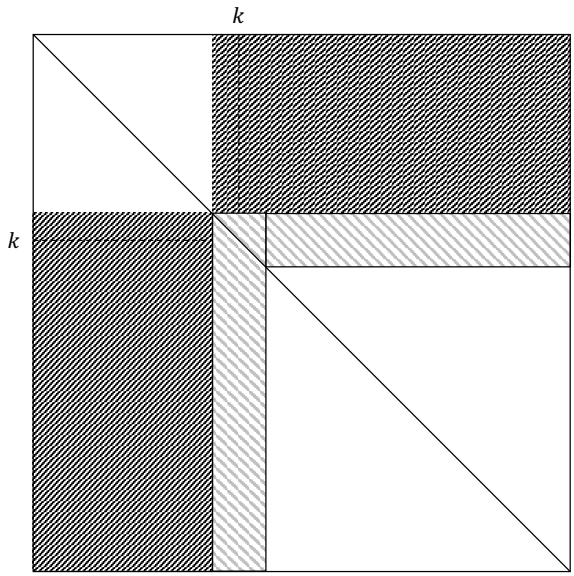

図 3.6　クラウト法の LU 分解における更新イメージ．細い右上がりの黒線で塗られた領域が参照領域，太い右下がりの灰色線で塗られた領域が更新領域である．

Algorithm 4 Forward Substitution for $l_{i,i} = 1$

1: ▷ Do nothing for b_0
2: **for** $i = 1$ to $n - 1$ **do**
3:　　**for** $j = 0$ to $i - 1$ **do**
4:　　$b_i = b_i - b_j \times a_{i,j}$
5:　　**end for**
6: **end for**

図 3.7　$l_{i,i} = 1$ の場合の前進代入

このとき，最終的に必要なものは解のベクトル $\boldsymbol{x}$ であり，左辺ベクトル $\boldsymbol{b}$ の値は保存されていなくて良い．よって，LU 分解の過程と同様に，$\boldsymbol{y}, \boldsymbol{x}$ の値を $\boldsymbol{b}$ の値の上に上書きしていく．これにより，必要なメモリ量を削減することができる．図 3.7・3.8 に，$\boldsymbol{Ly} = \boldsymbol{b}$ から $\boldsymbol{y}$ を求めるアルゴリズムを掲載する．ただし，$l_{i,i}$ の値が 1 であるか否かで除算が必要かが異なるため，場合分けして記載する．また，アルゴリズム中で $a_{i,j}$ と記載されるものは，$a_{i,j}$ のメモリ上にあるデータの意味であり，実際には $l_{i,j}$ の値で上書きされていることを前提とする ($i > j$ であるため)．

第 2 段階では

$$\boldsymbol{Ux} = \boldsymbol{y} \tag{3.21}$$

から $\boldsymbol{x}$ を計算する．このときインデックスの大きい行から小さい行に向かって値を定め代入していくため，**後退代入 (backward substitution)** と呼ぶ．[*7]前進代入と同様に，$l_{1,1} = 1$ により場合分けして記載すると図 3.9・3.10 の通りとなる．

ガウスの消去法において，インデックスの大きい行に向かって係数を消去していき，逆向きに解の値を求めていく操作を指す．ここではインデックスの大きい行に向かって値を定めているのであり，消去を行っている訳ではないので，前進代入という用語が適切だろう．

[*7] 「後進代入」という表記も見られる．

Algorithm 5 Forward Substitution for $u_{i,i} = 1$

1: $b_0 = b_0 \ / \ a_{0,0}$
2: **for** $i = 1$ to $n - 1$ **do**
3: **for** $j = 0$ to $i - 1$ **do**
4: $b_i = b_i - b_j \times a_{i,j}$
5: **end for**
6: $b_i = b_i \ / \ a_{i,i}$
7: **end for**

図 3.8　$u_{i,i} = 1$ の場合の前進代入

Algorithm 6 Backward Substitution for $l_{i,i} = 1$

1: $b_{n-1} = b_{n-1} \ / \ a_{n-1,n-1}$
2: **for** $i = n - 2$ to 0 **do**
3: **for** $j = i + 1$ to $n - 1$ **do**
4: $b_i = b_i - b_j \times a_{i,j}$
5: **end for**
6: $b_i = b_i \ / \ u_{i,i}$
7: **end for**

図 3.9　$l_{i,i} = 1$ の場合の後退代入

3.3　反復法

(2) のように近似解の更新を行なっていく方法は，更新アルゴリズムを反復させることから，**反復法 (iterative method)** や**反復解法**と呼ばれる．また直接解法に対置して**間接解法**とも呼ばれる．反復法にも定常反復法と非定常反復法が存在し，前者にはヤコビ法・ガウス=ザイデル法・SOR 法を，後者には共役勾配 (conjugate gradient; CG) 法やその発展系を挙げられる．本論文においてはアルゴリズムを紹介するに留め，手法の詳細には立ち入らない．例として，図 3.11 に CG 法のアルゴリズム [19] を示す．

3.4　直接法と反復法

直接法と反復法にはそれぞれ利点・欠点が存在し，場合により使い分けることが求められる．まず反復法の利点として，特に疎行列を扱う場合，直接法のように行列を分解してその分のデータを格納する必要がないため，メモリを節約できる点が挙げられる．また前処理の利用によって直接法よりも速く解を求めることが可能になる．一方の直接法の利点としては，反復法では収束しない形の線形方程式も扱えることを第一に挙げられる．物性値の大きく異なる

Algorithm 7 Backward Substitution for $u_{i,i} = 1$

▷ Do nothing for b_{n-1}

for $i = n-2$ to 0 **do**

  for $j = i+1$ to $n-1$ **do**

    $b_i = b_i - b_j \times a_{i,j}$

  end for

end for

図 3.10　$u_{i,i} = 1$ の場合の後退代入

Algorithm 8 Conjugate Gradient Method

$\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$

$\boldsymbol{p}_0 = \boldsymbol{r}_0$

for $k = 0$ to maximum iteration **do**

  $\alpha_k = \frac{\boldsymbol{r}_k^\top \boldsymbol{p}_k}{\boldsymbol{p}_k^\top \boldsymbol{A}\boldsymbol{p}_k}$

  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$

  $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{A}\boldsymbol{p}_k$

  if $\|\boldsymbol{r}\| \leq \varepsilon$ **then**

    break

  end if

  $\beta_k = \frac{\boldsymbol{r}_{k+1}^\top \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^\top \boldsymbol{r}_k}$

  $\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$

end for

図 3.11　共役勾配法のアルゴリズム

複合材料からなる物体の構造解析や，係数行列の対角成分に 0 の入る接触解析，複数部材からなるアセンブリ構造を扱う場合，反復法では残差が 0 に近付かず，収束解を得られないことがある．また，反復回数の定まらない反復法と異なり，計算回数をあらかじめ求めることが可能である．加えて，行列を分解するという操作は，右辺ベクトルである $\boldsymbol{b}$ が変化しても結果が変わらないため，異なる試行により複数の $\boldsymbol{b}$ に対して求解する必要がある場合にも，結果を再利用できるという利点がある．逆に反復法では，都度残差ベクトル $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$ を計算する必要がある．

上記のように，より一般の係数行列を扱える直接法は，これからの計算科学の分野においても必要とされる手法である．さらに，計算機の発達により，メモリ容量も大きくなってきたため，必ずしも直接法を計算するには不足であるということはなくなってきた．ただし，それでも反復法より消費が大きいことは確かであるため，メモリを節約しつつも直接法で計算することが求められる．次章では，特に疎行列の場合に，分解によるメモリ消費を抑えた直接法の手順を見ていく．

第 4 章

疎行列線形方程式の直接解法

前章で導入した直接法は係数行列 $\boldsymbol{A}$ の疎密に関わらず用いられるものである．しかしながら，有限要素法によって得られる係数行列のように疎行列を扱う場合，LU 分解は多数の**フィルイン (fill-in)** を生じる．フィルインとは，元の係数行列 $\boldsymbol{A}$ では成分の値が 0 であったところに，分解後の行列では非零の値が入ることを指す．6.1 節で述べるように，疎行列を記録するには非零成分のみを取り出すことが行われるため，フィルインの増加は使用メモリ量の増加を意味する．本章では，$\boldsymbol{A}$ が疎かつ正定値対称行列の場合について，フィルインの抑制によりメモリ消費を減らしつつ，かつ並列化と組み合わせることで効率良く求解を行える方法を紹介する．

4.1 解法の全体像

求解は主に 5 ステップからなる．初めのオーダリングでは，分解によるフィルインがなるべく生じないように行列要素の並び替えを行う．シンボリック分解では，フィルインの個数を確定するとともに，次の分解操作における処理の依存関係を求める．マルチフロンタル法は，係数行列 $\boldsymbol{A}$ が疎な場合にコレスキー分解の並列化を実現する手法である．前進後退代入は LU 分解の場合と同一である．反復的な解の改良は，直接法により得られた解における数値誤差を緩和するため，反復法のように残差を利用して解を更新する操作である．

4.2 オーダリング

行列の要素に適当な順序を付けて入れ替えることから**オーダリング (ordering)** と呼ばれる．[*1]元の順序からの並び替えであることから，**リオーダリング (reordering)** と表記することもある． [20] 方法には，Cuthiil-McKee 法，Nested Dissection 法などの種類が存在するが，ここでは Nested Dissection 法を取り上げる．

Nested Dissection 法とは，行列の非零要素をグラフ化した際に，そのグラフを再帰的に分割・並び替えしていくことで依存関係のない部分を多くする手法である．有限要素法の場合，メッシュをそのまま利用することができる．頂点 (vertex) の集合を V，辺 (edge) の集合を E とすると，グラフは $G = (V, E)$ と表される．このとき，各頂点は行列要素のインデックスに対応し，辺が存在することは，両端の頂点に対応するインデックスにおける要素が非零であることを示す．式で表すと

$$V = \{i | i \text{ is } \boldsymbol{A}'\text{s index}\}, \quad E = \{(i, j) | a_{i,j} \neq 0\} \tag{4.1}$$

[*1] 日本語で書けば「順序付け」

となる．

グラフ G を，領域 A, B の 2 つに分割する． [21] このとき A と B はなるべく大きくかつ同じ大きさになるようにとり，A に含まれる頂点と B に含まれる頂点を両端とする辺が 1 つもないようにする．A, B 以外の頂点集合を，セパレータ S とする．ここで行列のインデックスを

(1) A に含まれる頂点に対応するインデックス
(2) B に含まれる頂点に対応するインデックス
(3) S に含まれる頂点に対応するインデックス

の順に並び替える．すると，A, B の設定より，両者をまたがる辺は存在しないことから．非対角領域に零行列が現れる．一方で，S に対応するインデックスの部分，すなわちインデックスが大きい成分については，非零成分と 0 が混在し，零行列が見られない領域が帯状に形成される．このようにして得られる行列を**縁付きブロック対角行列 (banded block diagonal matrix)** と呼ぶ． [21]

A, B それぞれについて上記の領域分割操作を再帰的に繰り返すことで，A, B に対応する対角領域の密行列からも，さらに再帰的に縁付きブロック対角行列を得られる．このとき，非対角部分が零行列となっている場合には独立に分解操作が行えるため，並列化に優れる．

4.3 シンボリック分解

シンボリック分解 (symbolic factorization) とは，分解と名がつくものの LU 分解のように行列積の形に分解する訳でなく，フィルインの数を決定する作業である．また，フィルインの場所が特定されることに伴って，成分ごとの依存関係も決定できる．[*2] このとき，依存関係を木構造で表したものを**消去木 (elimination tree)** と呼び，木の親ノードが自身の子ノード全てに依存している形をとる．つまり，葉 (leaf) はそれぞれ依存しているものが存在せず，根 (root) は他の全てのノードに依存していることになるので，葉から根に向かうという計算順序が決定される．

分解操作を行うマルチフロンタル法は，外積形式のコレスキー分解の発展系である．そのため，依存のない部分で独立に分解できるとは言え，その中ではインデックスの小さい方から順に分解を行う．よって消去木の作成においては，インデックスの小さい方 (行・列) から順に走査していき，ある着目しているインデックス i に依存するインデックス j のうち最小のものを i の親ノードとする． [22] 具体的には

$$\text{parent of } i = \min_{j>i}\{j | a_{i,j} = 1 \text{ or } (a_{i,j} = 0 \text{ and } \exists c \in \{i\text{'s child}\}, a_{c,j} = 1)\} \tag{4.2}$$

となる．また，i に依存するが i の親である j には依存しないインデックス k があったとき，$i \to j$ の順での分解は既に定まっているので，分解順序としては j の後に k が処理される．よって k は j にも依存する形となり，フィルインが生じる．上式の通り，次に j の親を定める場合には，このフィルインも含めて j に依存するインデックスで最小のものを探索する．

[*2] 依存を生じさせることが必要になったのでフィルインが発生する，とも言える．

4.4 マルチフロンタル法

有限要素法により得られる線形方程式をはじめとして，左辺の $\boldsymbol{A}$ には大規模かつ疎な行列を持つことがある．すると，疎性を生かしたメモリ利用を行い，かつ並列計算を行うことで規模の大きさにも対応するには反復法が望ましい手法であるが，先に述べた通り収束しない可能性も存在するため，確実な求解には直接法が向いている．したがって，疎性を生かし，かつできるだけ並列計算を可能にした直接法アルゴリズムが重要である．それを実現するのが**マルチフロンタル法 (multifrontal method)** であり，以下で見るように行列の依存のない部分単位で分解を行う．

歴史的には，Iron [23] により 1970 年に有限要素法の係数行列に対するフロンタル法が提案された．その後，1976 年には Hood [24] により非対称行列を扱う解法へと拡張が行われた．また，Duff & Reid [25] は 1983 年に不定値対称行列を対象とした研究を発表している．なお．今日では正定値対称行列のみをターゲットとした場合もマルチフロンタル法という名称が用いられているが，厳密には Iron, Hood は “frontal solver” という用語を使用していた．

LU 分解では，分解の対象として着目しているインデックス k をループで回すことで順に全てのインデックスに対して同じ演算を行っていた．一方で，マルチフロンタル法においては独立な部分を先に同時に計算するよう順序を入れ替えるため，ループにより全てのインデックスについて見ていくことはしない．また，オーダリングにより依存のない領域を確保したことで，着目するインデックス k により演算に必要な成分・回数が異なってくる．したがって，アルゴリズム [26] としては，図 4.1 のように k についての分解のみを示す関数を提示する．

着目しているインデックス k についての分解は，図 4.1 に示されるアルゴリズムの関数に従って行われる．ここで，$\boldsymbol{F}^k$ をインデックス k についての**フロンタル行列 (frontal matrix)**，最終的に得られる $\boldsymbol{U}^k$ を**アップデート行列 (update matrix)** と呼ぶ．$\boldsymbol{F}^k$ ははじめの段階では第 k 行 (列) の非零成分のみからなる行列であり，これに消去木において k の子たる i のアップデート行列 $\boldsymbol{U}^i$ を足し込む．図 4.1 から分かる通り，$\boldsymbol{U}^i$ は FACTOR(i) によって得られる．この加算演算は，$\boldsymbol{F}^k$ と $\boldsymbol{U}^i$ のインデックスを揃えた上で値を足す必要があり，そのまま和をとる訳にいかないため，Extend-Add 演算と呼ばれる (図 4.1 中の $\oplus$)．これを全ての子 i について行う．

図 4.1 でのアルゴリズムでは, コメントで示しているように, 元々 $(s+1)\times(s+1)$ 行列であった $\boldsymbol{F}^k$ が, Extend-Add 演算により $(t+1)\times(t+1)$ 行列になったと仮定している．インデックス k に対応する部分は分解後の $\boldsymbol{L}$ の成分に利用され，残りから $\boldsymbol{U}^k$ が作成されるため，$\boldsymbol{U}^k$ は $t\times t$ 行列である． [21]

4.5 前進後退代入

本章では $\boldsymbol{A}$ として正定値対称行列を考えていたため，コレスキー分解が可能である．よって，LU 分解において $\boldsymbol{U}$ としていたところを $\boldsymbol{L}^\top$ で置き換える．その他は同様である．つまり

$$\boldsymbol{y} = \boldsymbol{L}^\top \boldsymbol{x} \tag{4.3}$$

とおき

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{L}\boldsymbol{L}^\top \boldsymbol{x} = \boldsymbol{L}\boldsymbol{y} = \boldsymbol{b} \tag{4.4}$$

という関係から，初めに $\boldsymbol{y}$ を求め，次いで $\boldsymbol{x}$ を求める．

Algorithm 9 Multifrontal Method

function FACTOR(k)

$$\boldsymbol{F}^k = \begin{bmatrix} a_{k,k} & a_{k,q_1} & a_{k,q_2} & \cdots & a_{k,q_s} \\ a_{k,q_1} & 0 & 0 & \cdots & 0 \\ a_{k,q_2} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{k,q_s} & 0 & 0 & \cdots & 0 \end{bmatrix}$$

for i is k's child **do**

  FACTOR(i)

  $\boldsymbol{F}^k = \boldsymbol{F}^k \oplus \boldsymbol{U}^i$ ▷ $\oplus$ denotes Extend-Add operation

end for

for $i = 0$ to t **do** ▷ Assume $\boldsymbol{F}^k \in \mathbb{R}^{(t+1)\times(t+1)}$, $\boldsymbol{U}^k \in \mathbb{R}^{t\times t}$

  $l_{q_i,k} = \boldsymbol{F}^k_{i,0} \,/\, \sqrt{\boldsymbol{F}^k_{0,0}}$

end for

for $j = 1$ to t **do**

  for $i = j$ to t **do**

    $\boldsymbol{U}^k_{i,j} = \boldsymbol{F}^k_{i,j} - l_{q_i,k} \times l_{q_j,k}$

  end for

end for

end function

図 4.1　マルチフロンタル法によるインデックス k の分解

4.6　反復的な解の改良

反復法においては，各ステップにおける近似解ベクトルを元の式に代入し，そこで得られた残差ベクトルのノルムが十分小さくなるまで反復を繰り返していた．一方，直接法においては，残差を小さくすることを考えず，数学的な操作のみに依拠して求解を行うため，計算機でこれを実現するとその途上で数値誤差が生じてしまう．これを補正するため，以上の過程で得られた解ベクトルを元の式に代入し，反復法のように得られた残差ベクトルを小さくする操作を行うことが望ましい．これを**反復的な解の改良 (iterative refinement)** と呼ぶ．

アルゴリズムとしては図 4.2 のように示せる．このとき，$\boldsymbol{z}$ の計算には前進後退代入を利用する (実装の部分で述べる)．

直接法の求解過程での数値誤差を低減する物であるため，ここでは求解に利用した浮動小数点数よりも高い精度を利用することが求められる． [19] 具体的には，単精度 (single precision; float, fp32) で求解を行ったなら倍精度 (double precision; double, fp64)，倍精度で求解を行ったなら倍々精度 (double-double precision) や 4 倍精度

Algorithm 10 Iterative Refinement

1: **for** proper time **do**

2: $\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$

3: $\boldsymbol{z} = (\boldsymbol{L}\boldsymbol{L}^{\top})^{-1}\boldsymbol{x}$

4: $\boldsymbol{x}_{\mathrm{new}} = \boldsymbol{x} + \boldsymbol{z}$

5: **end for**

図 4.2　反復的な解の改良

(quadruple precison; fp128) を用いる．現在の計算機においては 64 bit が主流となっていることを踏まえると，高精度な演算が求められる計算においては，浮動小数点数として基本的に倍精度を用いるのが良い．またハードウェア的に 4 倍精度が実装されている計算機が少ないことを踏まえると，反復的な解の改良においては倍々精度との組み合わせが望ましい．

第 5 章

並列計算

本章では，分散メモリ型計算機において並列計算の実現のために利用される規格である Message Passing Interface (MPI) について概観し，マルチフロンタル法の並列化に当たって必要な要素を紹介する．具体的な関数の実装については次章で言及する．

5.1 計算機の分類

計算機の分類における代表的なものとして，Flynn により 1966 年に提案された 4 類型が存在する． [27][28][9] これは，命令流とデータ流がそれぞれ単一か複数かによって区分するものであった．具体的には

- 単一命令流単一データ流 (single instruction stream, single data stream; SISD)
- 単一命令流複数データ流 (single instruction stream, multiple data stream; SIMD)
- 複数命令流単一データ流 (multiple instruction stream, single data stream; MISD)
- 複数命令流複数データ流 (multiple instruction stream, multiple data stream; MIMD)

の 4 つである．なお，計算機における「命令」とは「加算」「乗算」などの演算や「データの書き込み」といった単一の動作のことを指す．最も単純なのが SISD であり，名称の通り，ただ 1 つのデータに対しただ 1 つの命令を実行する．具体的には，a という値に対し，別の b という値を足し込む $a + b$ という演算を考えられる．SIMD はこれを複数データに発展させたもので，配列等により用意された複数の異なるデータに対し，同じ命令を実行する．例えば，4 つの配列データ (4 次元ベクトルと見なせば良い) $a[0], a[1], a[2], a[3]$ に対し，別の 4 データ $b[0], b[1], b[2], b[3]$ を足し込む操作を挙げられる． [9]

MISD は，複数のプロセッサを用意することで単一のデータに対し別の命令をそれぞれ実行する仕様であるが，一般的な実装は存在しない．最後に MIMD であるが，SIMD と同様に複数のデータを扱うものの，全データに同じ命令を実行する訳ではなく，条件分岐により別々の命令を提供することができる． [9]

MIMD におけるアプリケーションの実装として，**SPMD (single program, multiple data stream)** という考え方がある． [9] これは，複数のプロセッサにおいて同じプログラムを走らせるものの，プロセッサによって処理の内容を変化させ，また各プロセッサには異なるデータ (一連のデータの異なる部分など) を扱わせるというものである．処理の内容が異なるという点で複数命令流，別のデータを扱うという点で複数データ流である．

計算機の分類の仕方としては，1 つのメモリが複数のプロセッサによって共有されているか，プロセッサごとにメモリが存在するか，という区分けも可能である．前者を**共有メモリ型**，後者を**分散メモリ型**という．上で挙げた SPMD は分散メモリ型での実行に適しており，各プロセッサがそれぞれメモリを有することで，MIMD 演算が可能となる．一方の共有メモリ型は SIMD 演算向きであり，代表的な利用法としては，OpenMP [3] により配列演算に対する独立なループを並列化することを挙げられる． [9]

5.2 Message Passing Interface

SPMD を実現する規格として，**MPI (Message Passing Interface)** が存在する．MPI を利用したプログラムでは，プロセッサごとにランクと呼ばれる番号付けがなされ，そのランクによって分岐させることで処理を変更することが可能である．また，その単一プログラムに対して異なるデータを与えることで SPMD であると言える．分散メモリ型においては，プログラム中において同じ名前で定義された変数でも，異なるプロセッサ同士では異なる値を持つ可能性があり，また，あるプロセッサからは異なるプロセッサのもつデータに直接アクセスすることができない．しかし，これでは別々のプロセッサで計算した値を合算したり，逆にあるプロセッサのみが持っているデータを共有したり，といった操作を行えない．そのため，MPI にはデータのやり取りに関する様々な関数が定義されている．代表例としては，1 対 1 の送信を行う MPI_SEND，1 対 1 の受信を行う MPI_RECV，1 対多の送受信を行う MPI_BCAST を挙げられる．

MPI の仕様は，MPI Forum [4] によって定められており，現在の最新版は 2021 年 9 月に策定された MPI-4.0 である．一方で，実装には様々な種類があり，オープンソースの代表的なものには Open MPI [29], MPICH [30], MVAPICH [31] を挙げられる．その他，計算機ベンダによる独自実装も存在しており，Intel MPI や Fujitsu MPI などが使われている．東京大学のスーパーコンピュータでは，Oakbridge-CX [32] が Intel MPI を，Wisteria [6] が Intel MPI, Fujitsu MPI をそれぞれ利用可能である．

5.3 MPI によるプロセス管理インタフェース

上で説明した通り，MPI は SPMD のモデルをプログラムするのに利用可能な規格である．この場合，プログラム (アプリケーション) の実行開始時に，用いる全てのプロセッサで同じプログラムを起動させる．例えば，最も単純な例として 4 プロセッサで「a.out」というプログラムを起動させることをコマンドで表すと，`mpirun -np 4 ./a.out` となる．

一方で，MPI-2.0 より導入された機能として，プロセス生成・管理機能が存在する． [4] これは，初めから全てのプロセッサでプログラムを動かすのでなく，ある Master と呼ばれるプロセス (複数のこともある) [*1]が，実行途上に Worker と呼ばれるプロセスを生成したり消去したりするプログラミングモデルである．Master/Worker モデルにおいては，動的にプロセス数が決定されるため，スーパーコンピュータ等のクラスタ計算機において多数のプロセッサが接続されている場合，必ずしもプロセッサ間が最適に接続されない可能性がある．一方で，異なる内容のプログラムを実行する Worker を生成することも可能なため，異なるハードウェアでの実行を組み合わせたり，初めから各プ

[*1] inclusive language に反する表現であるが，一般的に用いられてきた単語であるためここでも使用している．

ロセッサでの処理が均等でないと分かる場合に実行数や内容を変えたりすることが容易である (これは SPMD の考え方とは反するものである) . 以下では，前章で説明したマルチフロンタル法の並列化を Master/Worker モデルで実現することを目指し，MPI による仕様についてより詳しく見る.

5.4 MPI_COMM_SPAWN によるプロセス管理

MPI においてプロセス生成を行う関数として, MPI_COMM_SPAWN が定義されている．ここでは Open MPI [29] による実装を確認し，その機能を概説する．C 言語による文法として掲げられているものをそのまま引用 [33] すると

```
#include <mpi.h>
int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
    MPI_Info info, int root, MPI_Comm comm,
    MPI_Comm *intercomm, int array_of_errcodes[])
```

となっている.[*2]第 1 引数の`*command` は名称の通りコマンドであるが，具体的には Worker として生成するプロセスにおいて実行させるアプリケーションの名称を指す．第 2 引数の`*argv` はそのアプリケーションに渡す引数であり，MPI での実行を指示するバイナリ (実行ファイル) が受け取る引数ではない．よって第 1 引数のアプリケーションの名称自体は含まれない．第 3 引数は，Worker として生成するプロセス数である．つまり，ここまでの 3 つの引数により，コマンドライン上で `mpirun -np [maxprocs] [*command] [*argv]` と打つのと同等の働きをなす.

第 5 引数は Master となるプロセスのランクであり，本論文では触れていないが，データの授受に際してランク (送信元・送信先) を指定するのと同様である．第 6 引数と第 7 引数の `MPI_Comm` とは，コミュニケータと呼ばれるもので，MPI プログラムを実行中のプロセスの集合を示す．今，そもそも Master 用に起動しているプログラムが存在し (1 つでも複数でも良い) ，この Master となり得るプロセスで 1 つの集団ができている．第 6 引数には，この集団，もしくは Master となるある単一のプロセスのみの集団を指定することで，実際に Master として働くものがいずれかを定める．一方，第 7 引数にある`*intercomm` とはインターコミュニケータのことであり，Master となり得る (Master と同じプログラムが走る) プロセスの集団と共に，Worker として動いているプロセスの集団も格納される．Worker の起動後は，このインターコミュニケータを用いることで，Master と各 Worker での通信が可能となる.

[*2] C 言語向けの関数では，関数名のうち MPI_に続く部分の 1 文字目のみ大文字となっている．Fortran 言語向けでは，全て大文字となっている.

第 6 章

マルチフロンタル法を用いた直接法の実装

6.1 疎行列の格納方式

行と列の 2 次元で構成される行列は，計算機上においても 2 次元配列として格納することで，数学的に書き下した場合とインデックスの対応がとれるため，アルゴリズムの実装において配列要素へのアクセスが理解しやすい．しかしながら，成分に 0 の多い疎行列を利用する場合には，積の結果が 0 となるため不要な演算が多く，2 次元配列を利用して全ての成分を格納するのは非効率なメモリ消費と言える．したがって，非零成分のみを格納する形式が考案されてきた． [20]

直感的に分かりやすいのは，図 6.1 のように全ての非零成分について行・列・値を記載した **COO (Coordinate) 形式**と呼ばれる表記法である．対称行列の場合には，対角成分以外で対称となる成分を省略し，上 (下) 三角行列の部分のみ記載することもある．

COO 形式は非零成分の位置が見た目から理解しやすいが，同一行の成分について見ていっている間，非零成分の数だけ行のインデックスを格納するのがメモリの無駄になっている．よって，非零成分の行のインデックスを格納する代わりに，「ある行の非零成分の開始点を示すポインタ[*1]」を格納しても良い．この手法を採用したのが，図 6.2 に示す **CSR (compressed sparse row) 形式**である． [20] **CRS (compressed row storage) 形式**とも呼ばれる．

疎行列ベクトル積 (sparse matrix-vector multiplication; SpMV) を考える場合，CSR 形式が計算上有利である．

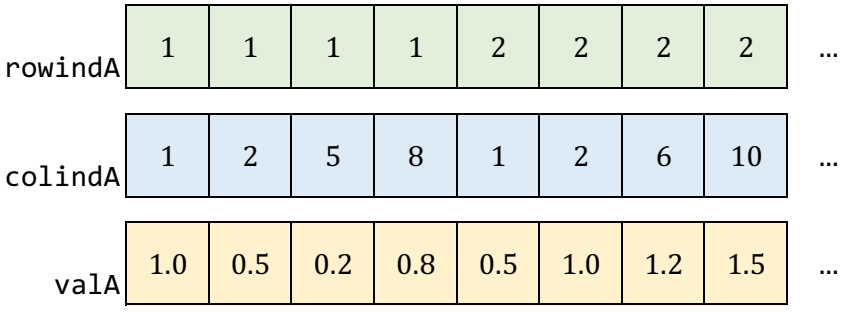

図 6.1　COO 形式による疎行列の格納

[*1] メモリ位置を指すポインタではなく，列インデックス・値の配列における各行の開始点の要素番号のことである．

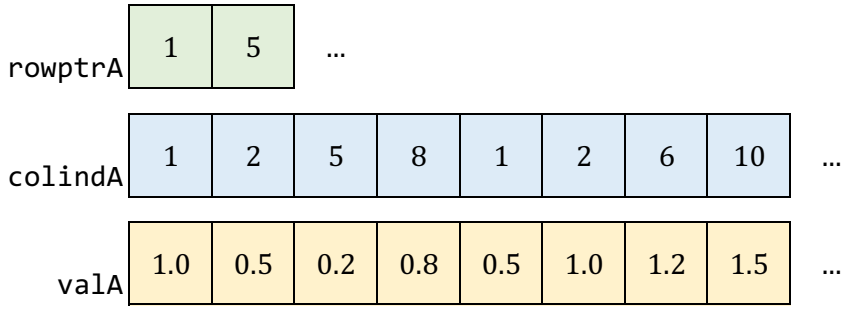

図 6.2　CSR 形式による疎行列の格納

具体的に，所与の $\boldsymbol{K}, \boldsymbol{\phi}$ から，$\boldsymbol{K}\boldsymbol{\phi} = \boldsymbol{f}$ という演算によりベクトル $\boldsymbol{f}$ を求めることを考える．

$$f_i = \sum_{j=1}^{n} k_{i,j} \times \phi_j \tag{6.1}$$

より，ベクトルの第 i 成分を求めるには，行列の第 i 行の非零成分が全て必要になる．このとき，COO 形式で格納された疎行列を利用するならば，各非零成分が第 i 行のものであるかの判定を都度行い，真であれば $m_{i,j} \times v_i$ を計算することになる．一方 CSR 形式であれば，第 i 行の成分であるかは図 6.2 でのポインタを収めた配列 `rowptr` によって予め分かるため，各 i に対し $\texttt{rowptr}[i+1] - \texttt{rowptr}[i]$ 回ループを回すことを記述可能である．

6.2　オーダリングの実装

ここでは既存ライブラリである METIS [34] を利用した方法を紹介する．METIS による Nested Dissection 法の関数を利用することで置換ベクトルと逆演算のベクトルが返ってくる．元の行列を $\boldsymbol{A}$，置換後の行列を $\boldsymbol{A}'$ とすると，置換行列 $\boldsymbol{P}$ による行列同時置換とは

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{6.2}$$

$$\boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^{-1}\boldsymbol{P}\boldsymbol{x} = \boldsymbol{P}\boldsymbol{b} \tag{6.3}$$

$$\boldsymbol{A}'\boldsymbol{x}' = \boldsymbol{b}' \tag{6.4}$$

という操作により式 (6.4) を得ることである．以後の計算に当たっては $\boldsymbol{x}'$ を求め，最後に $\boldsymbol{x} = \boldsymbol{P}^{-1}\boldsymbol{x}'$ とすることで真の解 $\boldsymbol{x}$ を得る．

METIS から得られるベクトルは

$$\texttt{perm[i]} : \boldsymbol{A}_{\texttt{perm[i]},j} = \boldsymbol{A}'_{i,j} \tag{6.5}$$

$$\texttt{iperm[i]} : \boldsymbol{A}'_{\texttt{iperm[i]},j} = \boldsymbol{A}_{i,j} \tag{6.6}$$

となるものである．これを利用して $\boldsymbol{P}, \boldsymbol{P}^{-1}$ を構成する．

$$\boldsymbol{P} = \begin{cases} 1 & (i,j) = (i, \texttt{iperm}[i]) \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

$$\boldsymbol{P}^{-1} = \begin{cases} 1 & (i,j) = (\texttt{iperm}[j], j) \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

とすると，$\boldsymbol{P}^{-1}$ を右からかけることにより列基本変形 (列の置換)，$\boldsymbol{P}$ を左からかけることにより行基本変形 (行の置換) がそれぞれ実現される．これにより $\boldsymbol{A}$ に対する行列同時置換が実現され，置換後にも $\boldsymbol{A}$ の対称性は維持される．ただし，実際の動作としては行列積を計算する訳でなく，置換後の順番となるように CSR 形式で格納された情報を並び替える．

6.3 シンボリック分解の実装

オーダリングにより行列同時置換を行った後の，列インデックスを格納した配列 `colindA` について見ていく．ある行について注目したとき，自身のインデックスよりも大きいインデックスは，いずれも自身に依存しているものである (非零成分しか格納していないため)．よって，この中で最小のインデックスが消去木での親ノードとなる．親ノードについては，行列の次元数 (有限要素法のメッシュであれば節点数) n の長さをもつ配列を別に用意しておき，そこに記録すれば良い．また，$n \times n$ の 2 次元配列を用意し，依存関係を記録する．元々依存関係のあった部分に加えて，子ノードと親ノードの行を比較することで，フィルインの箇所が明らかになる．自身以上のインデックスにおいて依存関係のフラッグが立った個数が，子ノードのアップデート行列 $\boldsymbol{U}^i$ を足し込む前の初期のフロンタル行列 $\boldsymbol{F}^k$ のサイズとなる．

6.4 マルチフロンタル法の実装

アルゴリズムの通り，分解の指示は消去木の親ノードから子ノードへと再帰的に行われる．逆に，実際の演算は葉から始まる．よって，消去木の根に当たる，最大のインデックスを初めの分解対象として与え，以下再帰的に全てのインデックスに対して分解操作を行う．シンボリック分解より親子関係は得られているため，アルゴリズムに従って構築した関数を，実際にインデックスを変えながら実行させれば良い．

6.4.1 MPI プロセス生成による並列化

消去木において親子関係にないサブツリー・ノード同士は，並列に計算することが出来る．したがって，消去木の根から順番に辿り，一番初めに分岐したところから下のサブツリー同士を 2 並列として計算することが可能である．同様に，再帰的に分解の指示を行っていき，分岐に当たる (子を複数持つ) ノードの度に並列数を増やすことが可能である．ここでは，最も単純な 2 並列を考えることにすると，最も根に近い分岐の部分から並列化が実施出来るので，このノードに達したときに，2 プロセスを生成すれば良い．生成した各 Worker には，分岐の子ノードに対応するインデックスをそれぞれ与えることで，異なるサブツリーの分解操作を行うことが出来る．サブツリー部分の分解終了後にはプロセスの消去を行い，並列化出来ない部分は Master 1 プロセスで計算する．以上が，動的プロセス生成を利用したマルチフロンタル法の並列化である．

Algorithm 11 Forward Substitution for CSR format

```
1: b[0] = b[0] / valL[0]
2: for i = 1 to n − 1 do
3:     w = rowptrL[i + 1] − rowptrL[i]
4:     for j = 0 to w − 2 do
5:         b[i] = b[i] − b[colindL[j]] × valL[rowptrL[i + j]]
6:     end for
7:     b[i] = b[i] / valL[rowptrL[i + 1] − 1]
8: end for
```

図 6.3　CSR 形式を用いた前進代入

Algorithm 12 Backward Substitution for CSR format

```
1: b[n − 1] = b[n − 1] / valU[N − 1]
2: for i = n − 2 to 0 do
3:     w = rowptrU[i + 1] − rowptrU[i]
4:     for j = 1 to w − 1 do
5:         b[i] = b[i] − b[colindU[j]] × valU[rowptrU[i + j]]
6:     end for
7:     b[i] = b[i] / valU[rowptrU[i]]
8: end for
```

図 6.4　CSR 形式を用いた後退代入

6.5　前進後退代入の実装

前進後退代入は，一般の正方行列に対する LU 分解でもコレスキー分解でも分解後に利用する手法である．よってこの項目においては，上三角行列を $\boldsymbol{U}$ と表記する．コレスキー分解においては $\boldsymbol{L}^\top$ に当たると見なせば良い．

アルゴリズムとしては第 3 章に示した通りであるが，CSR 形式の疎行列を用いた場合について記述する．ここで，図 6.2 と同じように，rowptrL, rowptrU はそれぞれ $\boldsymbol{L}, \boldsymbol{U}$ の行頭ポインタ，colindL, colindU は列のインデックス，valL, valU は非零成分の値を指す．また colindU, valU の配列長を N とする (コレスキー分解の場合は colindL, valL も同じ長さとなる)．

6.6 反復的な解の改良の実装

演算に用いた精度よりも高精度な型を用いる．ここでは求解までを倍精度で行うとし，QD [35] というライブラリを利用して倍々精度での解の改良を行う．求解の結果得た解のベクトルや，係数行列 $\boldsymbol{A}$，右辺ベクトル $\boldsymbol{b}$ は倍精度で記録されている．これらは，計算時に値を 1 つ取り出した際，ライブラリに含まれる `c_dd_copy_d` を用いることで，値をコピーし倍々精度で代入出来る．また，例えば「倍精度 – 倍々精度」から倍々精度で解を得たい場合には，`c_dd_sub_d_dd` (C 言語において，倍精度 [d] から倍々精度 [dd] の数値を引き [sub]，倍々精度 [dd] で格納する) という精度を混在させた形で演算出来るメソッドが用意されている．

改良のアルゴリズム中には

$$\boldsymbol{z} = (\boldsymbol{L}\boldsymbol{L}^\top)^{-1}\boldsymbol{x} \tag{6.9}$$

というベクトルを求める部分があるが，実際の演算においては

$$(\boldsymbol{L}\boldsymbol{L}^\top)\boldsymbol{z} = \boldsymbol{x} \tag{6.10}$$

から $\boldsymbol{z}$ を求める方程式として扱う． [19] これは，コレスキー分解を行った後の式 $\boldsymbol{L}\boldsymbol{L}^\top\boldsymbol{b} = \boldsymbol{x}$ と全く同じ形であり，前進後退代入によって $\boldsymbol{z}$ が得られることが分かる．

第 7 章

数値実験

7.1 計算条件

7.1.1 入力ファイル

実装したプログラムの検証に用いる入力ファイルとして，FrontISTR [16] の計算時に出力した係数行列・右辺ベクトルのファイルを用いた．また，実行結果の検算用に，同時に出力された反復法での解ベクトルのファイルも利用した．このとき係数行列のファイルは，6.1 節で説明した CSR 形式で表記されていることを前提とする．

以下で利用した題材は，例題として提供された，12,621 節点からなるモデルである．このとき，元の係数行列 $\boldsymbol{A}$ に含まれる非零成分は 971,199 であった．行列の全成分に占める割合は約 0.61 % であり，疎行列であると言える．また，ここで得られた $\boldsymbol{A}$ は正定値対称行列である．

7.1.2 計算環境

まず手元での検証環境として，Apple M1 チップを搭載した MacBook Air 2020 年モデルを利用した (以下 “Mac”)．クラスタ計算機の環境としては，研究室が所有する Intel Xeon Platinum 9242 (2.3 GHz, 48 Core) からなるものを利用した (以下 “Lab”)．また，東京大学情報基盤センターのスーパーコンピュータである，Oakbridge-CX [32] 及び Wisteria-Odyssey [6] においてもプログラムを実行した．

7.2 逐次計算プログラムの結果

はじめに並列計算を考慮しないプログラムを作成した．つまり，マルチフロンタル法の分解演算を 1 プロセッサのみで再帰的に行う．このとき，消去木上での分解指示の流れは，根から葉へと流れ，ある葉から親ノードへと順に分解されると，今度は別の枝を葉へと下る．すなわち，深さ優先探索の動きになっている．

Mac, Lab, Wisteria をそれぞれ利用して実行し，反復法の解との差を計算した．このとき誤差の絶対値における最大値は 10^{-16} オーダーであったことから，実装したプログラムにより正しい結果が得られたと結論付けられる．

また，Mac と Lab においてそれぞれ 5 回ずつ実行した結果の算術平均を表 7.1 にまとめる．

比較として，既存ライブラリである MUMPS [12] を用いたプログラムも作成した．MUMPS とは，1.3 節で紹介した，疎行列線形方程式に対して直接法による求解を行うライブラリの 1 つであり，マルチフロンタル法が用いられ

表 7.1　実行時間の結果 [単位: 秒]

	Mac	Lab
Ordering	0.566	0.130
Symbolic Factorization	0.133	0.311
Multifrontal	33.701	25.570
Forward and Backward Substitution	0.198	0.131
Iterative Refinement	0.229	0.108
Total until Substitution	34.598	26.142
Total	34.827	26.250

ている．また第 5 節で紹介した MPI による並列化が組み込まれているため，複数プロセッサによる実行が可能である．MUMPS を用いたプログラムを Mac で実行したところ，5 回の実行時間の算術平均は 0.358 秒であった．この結果は，MUMPS のデフォルト設定を利用し反復的な解の改良を行っていない場合の，全体の実行時間 (行列・ベクトルの読み込みは含まない) である．すなわち，表 7.1 での Total until Substitution の項目と直接比較可能である．

表 7.1 を参照すると，Mac と Lab では当然計算機の性能差が存在するため，特に演算に時間のかかるマルチフロンタル法での分解に実行時間差が生まれている．しかしながら，いずれにしろ MUMPS での実行時間と比較すると分解に時間を要していることは明らかであり，今後の効率化が求められる部分である．1.4 節で述べた通り，本研究では，既存ライブラリに依存しない形でのマルチフロンタル法の独自実装を目的の 1 つとしており，上記の通り，実装により正確と言える結果を得られた．ただし，実装においては 4.4 節 図 4.1 のアルゴリズムを再現しているのみであり，消去木において親子関係にあるインデックスの探索や，アップデート行列 $\boldsymbol{U}^i$ の格納においてメモリ効率を意識した構成となっていない．そのような点で，MUMPS を利用した場合に比べ大きさな差が生じていると考察される．

7.3　並列計算プログラムの結果

プロセス生成・消去を行うプログラムを作成し，逐次プログラムと同等の結果を得られることを確認するため，消去木の分岐におけるプロセス生成を行う回数を 0 と設定した (動的プロセス生成を行わないようにした) ．Lab での検証において，得られた誤差は逐次プログラムの場合と同オーダーであったことから，MPI によるプログラム実行としてもアルゴリズムとして破綻していないと言える．

プロセス生成・消去の実験として，Mac での実行で 2 プロセス生成，及びさらにそこから 2 プロセスずつの生成 (4 並列) を指示したところ，正しく消去木上のノードを辿ることが確認された．このとき，後者については合計 $1+2+4=7$ プロセスが存在していることになる．しかしながら，競合の関係からか，最後まで実行し結果を得ることは出来なかった．クラスタ計算機と異なり分散メモリ型であるとは言い切れないため，再帰的な分解・プロセス生成により適切なメモリ管理が行えなくなったと考えられる．

またクラスタにおける実行では，Master 部分は問題ないものの，Worker の生成が不安定であった．具体的には，生成出来るものの通信面でエラーとなったり，そもそも開始時点で割り当てがなされなかったり，という内容である．前者に関しては，MPI の実行ファイルのオプションを用いて，Master の実行時に同一計算ノード内となるよう指定

したところ，確かに 2 つの Worker がその内で実行されることを確認したが，通信エラー自体は改善しなかった．したがって，動的プロセス生成による並列化は，アルゴリズムとしては問題なく消去木上の走査・分解を行えるが，異種環境のハードウェアに対応するには今後の改修が必要である．

第 8 章

結論

行列の疎性を生かし，かつ適切な並び替えにより三角行列の積への分解を並列に行える線形方程式の直接解法として，マルチフロンタル法の実装を行った．既存ライブラリによるオーダリング・高精度演算を組み合わせつつ，先行研究 (特に [26]) で示されたアルゴリズムによって，依存のあるインデックスに対して再帰的に分解操作を実行させることで，数値実験において，反復法での解と同等の値をもつ解を得られた．

第 1 章で述べた通り，疎行列線形方程式の直接法による求解は，既存のライブラリによるところが大きい．よって本研究の主たる貢献としては，それらのライブラリに依存せず求解可能な実装を行えたことを挙げられ，今後オープンソース化などを目指すに当たっての基礎になったと言える．一方で，数値実験の結果より明らかな通り，現段階ではライブラリに匹敵する計算効率を発揮できていない．特に主たるマルチフロンタル法の演算部分の計算時間が長いことから，将来的により効率化することが求められる．

数値実験で得られた解の検証を行うに当たっては，手法による解の差が小さいことを確認するため，反復法でも計算可能な題材を扱った．ここで正確と言える結果を得られたということは，本実装が反復法では収束しない問題にも適用可能であることを述べられる．ただし，3.4 節で挙げたような反復法で収束しない事例を利用した訳ではないことから，実際にそれらの場合に得られる線形方程式を入力とし，結果を検証する必要がある．

並列化において，消去木の構造に応じたプロセス実行という認識のしやすさ，また本研究のキーの 1 つである異種環境での利用の想定から，動的にプロセス管理を行う MPI_COMM_SPAWN をプログラム中に組み込んだことも，本研究における貢献と述べられる．しかしながら，逐次プログラムでの実行と同様の順序を再現できたものの，メモリや通信面での調整の不足により，正確な結果の取得には至らなかったことから，演算の正確性を担保できていない点を限界として挙げられる．

その他，マルチフロンタル法を利用した直接法のメモリ消費が少ないことがアルゴリズムからしか示されておらず，一般的な LU 分解を行った場合と比べどの程度効率化されたかの定量的指標を本研究では用いることができなかった．

以上をまとめると，今後の課題としてはまず，実装を行えた逐次プログラムを踏まえ，異なる環境において適応的に動作できる，動的プロセス管理を用いたアルゴリズムの完成を挙げられる．また，プロセスの一部を異種環境に振り分けるような，Worker の対象を多様化できる構成も実装が望まれる．その上で，アルゴリズムの有効性を提示するため，直接法のみで求解可能とされる問題を扱うこと，メモリについても定量的な指標を用いることが必要となる．

ソルバの対象という面では，歴史的に非対称 [24] や不定値対称 [25] の行列も扱われてきたことから，正定値対称でない $\boldsymbol{A}$ への対応を挙げられる．これらを扱えることで，実際の設計現場で用いられる多様な問題に対応することを

望める．加えて，現在はソルバ単体であるため，構造解析プログラムである FrontISTR [16] に演算機能を組み込むことで，実用する上での実現可能性を考察できる．本研究が今後掲げる到達点は，上記のように求解可能な問題の幅を広げ，アプリケーションとして実装することにより，実社会のものづくり現場に寄与することである．

参考文献

[1] 藤野清次, 阿部邦美, 杉原正顯, 中嶋徳正. 線形方程式の反復解法. 計算力学レクチャーコース. 丸善出版, 2013.

[2] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 427–436, 2009.

[3] The OpenMP Architecture Review Board. OpenMP. https://www.openmp.org/.

[4] MPI Forum. MPI Forum. https://www.mpi-forum.org/.

[5] Richard Vuduc and Jee Choi. *A Brief History and Introduction to GPGPU*, pp. 9–23. Springer US, Boston, MA, 2013.

[6] 東京大学情報基盤センタースーパーコンピューティング部門. Wisteria/BDEC-01 スーパーコンピュータシステム. https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/.

[7] 産業技術総合研究所. ABCI. https://abci.ai/ja/.

[8] Serban Georgescu, Peter Chow, and Hiroshi Okuda. Gpu acceleration for fem-based structural analysis. *Archives of Computational Methods in Engineering*, Vol. 20, No. 2, pp. 111–121, Jun 2013.

[9] デイビッド・A・パターソン, ジョン・L・ヘネシー, 成田光彰. コンピュータの構成と設計 第 5 版 上・下. 日経 BP, 2014.

[10] IBM. Watson Sparse Matrix Package (WSMP). https://researcher.watson.ibm.com/researcher/view_group.php?id=1426.

[11] Intel. oneMKL PARDISO - Parallel Direct Sparse Solver Interface, 2022. https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top/sparse-solver-routines/onemkl-pardiso-parallel-direct-sparse-solver-iface.html.

[12] Mumps Technologies. MUMPS : a parallel sparse direct solver. https://mumps-solver.org/index.php.

[13] O. C. Zienkiewicz and K. Morgan. *Finite Elements and Approximation*. John Wiley and Sons, 1983.

[14] O. C. ジェンキェヴィッチ, K. モーガン, 伊理正夫, 伊理由美. 有限要素と近似. ワイリー・ジャパン, 1984.

[15] 奥田洋司, 稲垣和久, 竹内光秀. 非線形並列有限要素法 FrontISTR の理論・実装・応用. 計算力学レクチャーコース. 丸善出版, 2022.

[16] FrontISTR Commons. FrontISTR. https://www.frontistr.com/index.php.

[17] Dassault Systèmes. SIMULIA. https://www.3ds.com/products-services/simulia/.

[18] Inc. ABAQUS. ABAQUS Version 6.6 Documentation, 2006. https://classes.engineering.wustl.edu/

2009/spring/mase5513/abaqus/docs/v6.6/books/gss/default.htm.

[19] 寒川光, 藤野清次, 長嶋利夫, 高橋大介. HPC プログラミング. IT テキスト. オーム社, 2009.

[20] 緒方隆盛. 疎行列直接法ソルバ入門. 計算工学 = Journal of the Japan Society for Computational Engineering and Science, Vol. 25, No. 2, pp. 22–22:6, 2020.

[21] 山本有作. 疎行列連立一次方程式の直接解法. 計算工学 = Journal of the Japan Society for Computational Engineering and Science, Vol. 11, No. 4, pp. 1458–1462, 2006.

[22] Joseph W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, Vol. 34, No. 1, pp. 82–109, 1992.

[23] Bruce M. Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, Vol. 2, No. 1, pp. 5–32, 1970.

[24] P. Hood. Frontal solution program for unsymmetric matrices. *International Journal for Numerical Methods in Engineering*, Vol. 10, No. 2, pp. 379–399, 1976.

[25] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software*, Vol. 9, No. 3, pp. 302–325, sep 1983.

[26] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 5, pp. 502–520, 1997.

[27] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, Vol. 54, No. 12, pp. 1901–1909, 1966.

[28] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, Vol. C-21, No. 9, pp. 948–960, 1972.

[29] The Open MPI Project. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/.

[30] Mathematics and Computer Science Division, Argonne National Laboratory. MPICH. https://www.mpich.org/.

[31] The Ohio State University. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot. https://mvapich.cse.ohio-state.edu/.

[32] 東京大学情報基盤センタースーパーコンピューティング部門. Oakbridge-CX スーパーコンピュータシステム. https://www.cc.u-tokyo.ac.jp/supercomputer/obcx/service/index.php.

[33] The Open MPI Project. MPI_Comm_spawn(3) man page (version 4.1.4). https://www.open-mpi.org/doc/current/man3/MPI_Comm_spawn.3.php.

[34] George Karypis. METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering. http://glaros.dtc.umn.edu/gkhome/metis/metis/download.

[35] David H. Bailey. High-Precision Software Directory. https://www.davidhbailey.com/dhbsoftware/.

A

謝辞

指導教員である奥田洋司先生からは，修士課程 2 年間にわたり，研究に関して常にお声がけいただいた他，本研究以外にもソフトウェア開発や共同研究をはじめとして，様々なプロジェクトに携わる機会をいただきました．この場を借りて深く御礼申し上げます．同じく研究室の松永拓也先生からも，研究室ゼミの場において，研究に関して様々な助言・ご指摘をいただいた他，昨年開催された SC22 への参加に際してもお世話になりました．改めて御礼申し上げます．

研究室の特任研究員でいらっしゃる林雅江様には，研究に関してアドバイスいただいた他，FrontISTR 関連の研究会等参加にあたっては毎度大変お世話になりました．いつも資料作成が遅くご迷惑おかけしたことをお詫びすると共に，様々なサポートに対して感謝申し上げます．客員連携研究員の山口太一様には，特に共同研究のプロジェクトにおけるシミュレーションモデルの作成においてお世話になりました．職務の合間を縫ってお手伝いいただき，誠にありがとうございました．また，客員共同研究員の秋葉博様，櫛田慶幸様からは，ゼミや研究会を通じて貴重なお話を伺うことが出来ました．研究室を通じたつながりに感謝いたします．

研究室秘書の渡辺夏実様には，修士課程進学時より事務手続きの面で大変お世話になりました．本論文の執筆時点ではまだ今年度分の作業が終わっておりませんが，研究実施におけるベースのサポートに感謝いたします．

研究室 OB の井原遊様には，本研究テーマに関して，初期にキーとなるアドバイスをいただきました．また，同じく OB の筑波大学・森田直樹先生には，研究発表の場において助言をいただきました．

研究室の先輩である和田一宏さん，同期の原口泰雅さんとは，修士 1 年前半での勉強会を通じ，知識の共有・交流をすることが出来ました．またその後も，様々な機会において，研究以外にも話をすることが出来ました．未だ COVID-19 の流行が収まらない中であり，対面で会う機会は中々ありませんでしたが，一番近くの学生として交流出来たことに感謝いたします．後輩に当たる河原井啓さんは，研究室に配属された年次としては同期であり，私の知識・技術が至らない面について度々サポートしていただきました．本当にありがとうございました．研究室の他の学生の皆様にも，ゼミでのコメントなどに関して感謝いたします．

また，個人としてご指導いただいた訳ではありませんが，情報基盤センターの先生方・職員の方々の甲斐あって，スパコン講習会や関連授業を，知識の獲得・深化に役立てることが出来ました．SC22 の際にはブースで交流もさせていただき，良い意見交換の機会となりました．

最後に，経済面・生活面でのサポートを提供してくれた両親に感謝いたします．

本研究で実装したプログラムの検証・数値実験に当たっては，東京大学情報基盤センターのスーパーコンピュータ

グループ利用 (プロジェクト名称: 「非線形問題における並列有限要素解析の高速化に関する研究」, 代表者: 奥田洋司 教授) の下，FUJITSU Supercomputer PRIMEHPC FX1000 and FUJITSU Server PRIMERGY GX2570 (Wisteria/BDEC-01) 及び Fujitsu PRIMERGY CX400M1/CX2550M5 (Oakbridge-CX) を計算資源として活用いたしました.

B

主要なソースコード

Listing B.1　Extend-Add 演算

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

typedef int64_t mf_int;
typedef double mf_real;

int Extend_Add(
    mf_int *nFrontal_k, mf_int *nUpdate_i,
    double **Frontal_k, double **Update_i,
    mf_int *Frontalind, mf_int *updateind_i
    )
{
    /* index mapping */
    mf_int *U2Find = (mf_int *)calloc(*nUpdate_i, sizeof(mf_int));
    mf_int tmpptr = 0;
    for (mf_int l = 0; l < *nFrontal_k; ++l) {
        mf_int findl = Frontalind[l];
        for (mf_int m = tmpptr; m < *nUpdate_i; ++m) {
            if (updateind_i[m] == findl) {
                U2Find[m] = l;
                tmpptr++;
                break;
            }
        }
        if (tmpptr == *nUpdate_i) break;
    }

    /* add matrix elements */
```

```
    for (mf_int l = 0; l < *nUpdate_i; ++l) {
        mf_int globalInd1 = U2Find[l];
        Frontal_k[globalInd1][globalInd1] += Update_i[l][l];
        if (l < *nUpdate_i-1) {
            for (mf_int m = l+1; m < *nUpdate_i; ++m) {
                Frontal_k[globalInd1][U2Find[m]] += Update_i[l][m];
                // Frontal_k[U2Find[m]][globalInd1] += Update_i[m][l];
            }
        }
    }

    free(U2Find);
    return 0;
}
```

Listing B.2　マルチフロンタル法による分解 FACTOR(k)

```
int Factor(
    mf_int *nvtxs, mf_int *k,
    mf_int *rowptrA, mf_int *colindA, mf_real *valA,
    int8_t **depend, mf_int *nFrontal, mf_int *parent,
    mf_int *updateind_k, mf_real **Update_k,
    mf_int *rowptrFLT, mf_int *colindFLT, mf_real *valFLT
    )
{
    mf_int start = rowptrA[*k];
    mf_int end = rowptrA[*k+1];
    mf_int width = end - start;
    mf_int nFrontal_k = nFrontal[*k];

    mf_int *Frontalind = (mf_int *)calloc(nFrontal_k, sizeof(mf_int));
    Frontalind[0] = *k;
    if (*k < *nvtxs-1) {
        mf_int tmpptr_ = 1;
        for (mf_int i = *k+1; i < *nvtxs; ++i) {
            if (depend[*k][i] == 1) {
                Frontalind[tmpptr_] = i;
                updateind_k[tmpptr_-1] = i;
                tmpptr_++;
            }
        }
    }

    double **Frontal_k;
```

```
    Frontal_k = (double **)malloc(sizeof(double*) * nFrontal_k);
    for (mf_int i = 0; i < nFrontal_k; ++i) {
        Frontal_k[i] = (double *)malloc(sizeof(double) * nFrontal_k);
        for (mf_int j = 0; j < nFrontal_k; ++j) {
            Frontal_k[i][j] = 0.0;
        }
    }

    /* initial Frontal matrix */
    mf_int tmpptr = 0;
    for (mf_int i = 0; i < width; ++i) {
        mf_int colind = colindA[start+i];
        for (mf_int j = tmpptr; j < nFrontal_k; ++j) {
            mf_int findj = Frontalind[j];
            if (findj == colind) {
                Frontal_k[0][j] = valA[start+i];
                // Frontal_k[j][0] = valA[start+i];
                tmpptr = j + 1;
                break;
            }
        }
        if (tmpptr == nFrontal_k) break;
    }

    for (mf_int i = 0; i < *k; ++i) {
        if (parent[i] == *k) {
            mf_int nUpdate_i = nFrontal[i] - 1;
            mf_int *updateind_i = (mf_int *)calloc(nUpdate_i, sizeof(mf_int));
            double **Update_i;
            Update_i = (double **)malloc(sizeof(double) * nUpdate_i);
            for (mf_int ii = 0; ii < nUpdate_i; ++ii) {
                Update_i[ii] = (double *)malloc(sizeof(double) * nUpdate_i);
                for (mf_int ij = 0; ij < nUpdate_i; ++ij) {
                    Update_i[ii][ij] = 0.0;
                }
            }
            Factor(
                nvtxs, &i, rowptrA, colindA, valA, depend,
                nFrontal, parent, updateind_i, Update_i,
                rowptrFLT, colindFLT, valFLT
                );
            Extend_Add(
                &nFrontal_k, &nUpdate_i,
```

```
                Frontal_k, Update_i,
                Frontalind, updateind_i
                );
            free(updateind_i);
            free(Update_i);
        }
    }

    mf_int startL = rowptrFLT[*k];
    double Frontal_k00 = Frontal_k[0][0];
    for (mf_int i = 0; i < nFrontal_k; ++i) {
        valFLT[startL+i] = Frontal_k[0][i] / sqrt(Frontal_k00);
        colindFLT[startL+i] = Frontalind[i];
    }
    mf_int nUpdate_k = nFrontal_k - 1;
    for (mf_int i = 0; i < nUpdate_k; ++i) {
        for (mf_int j = i; j < nUpdate_k; ++j) {
            Update_k[i][j] = Frontal_k[i+1][j+1] - valFLT[startL+i+1]*valFLT[startL+j+1];
        }
    }

    free(Frontalind);
    free(Frontal_k);

    return 0;
}
```