東京大学
情報理工学系研究科 創造情報学専攻
修士論文

処理の差異と順序が記述可能な並列コレクション向け Java 言語拡張の提案

Java language extension for specifying
partial areas & orders in Parallel Collection

宗 桜子
Sakurako Soh

指導教員　千葉 滋 教授

2014 年 1 月

概要

本研究では，並列コレクションを利用した反復計算に対して，コレクションの計算毎に異なる処理やその計算順序をユーザが柔軟に定義する為の Java 言語拡張，Mosaic Collection for Java (MoCoJ) を提案する．並列コレクションを用いた map や reduction 処理では，コレクション中の要素に対する操作は並列実行され，更に要素の走査順序は隠蔽される．そのため，例えば物理シミューレションで用いられるステンシル計算のような領域全体に対する物理計算の反復処理を容易に記述できる可能性がある．しかし一方で，ステンシル計算を実装する為にはユーザはコレクションの部分領域毎に異なる処理を定義しなければならない上に，最適化の際に領域間の実行順序を制御する必要も生じる．

本研究では部分処理と部分順序を定義するための機構である部分メソッドディスパッチを Java 言語に導入する．部分メソッドディスパッチの利用により，ユーザは各部分領域固有の処理をメソッド上書きとして記述でき，更に領域間の実行順序を上書き条件として明示的に与えられる．また同名メソッドを上書きして走査順序を定義するため，MoCoJ では既存の並列コレクションを再利用することができる．

Abstract

In this paper, we propose a Java language extension, Mosaic Collection for Java (MoCoJ) for iterative calculations using Parallel Collections. With our language, users can define different calculations between particular elements and their orders in collections flexibly. In reductions or mappings using Parallel Collection, operations for elements in the collection are executed in parallel and futhermore their traversal orders are hidden. Therefore, there is a possibility that iterative processing for physical calculations on the entire area such as those used in physical simulations can be described easily. However, on the other hand, to implement a stencil calculation users have to define different process according to partial areas and control execution orders among partial areas when codes need optimizations.

We introduce partial method dispatch for specifying partial calculations and partial orders into Java language in this study. The use of partial method dispatch enables users to override default calculation by unique precess of each partial area, and also to give execution orders among areas as conditions for overriding explicitly. In addition, users can reuse original Parallel Collection using MoCoJ because of specifying traversal orders by overriding methods which has the same method name.

目次

第 1 章

はじめに

一般に HPC プログラミングにおいて，並列分散処理を用いたステンシル計算では通信最適化によって大幅な性能向上が見込まれる．しかし並列分散化したステンシル計算のプログラム中には通信コードが散在するだけでなく，問題の部分領域毎に行う処理に差異が生じたり，最適化手法の適用により問題領域の走査順序などのパラメータがハードコーディングされるなどして，プログラムは煩雑化する．そのためユーザにとって更なる最適化の為のコードの試行錯誤は困難になり，アプリケーションの最適化が十分に行えない恐れがある．

反復計算を記述するための並列コレクションライブラリを使うと，ステンシル計算を簡潔に記述できる可能性がある．次バージョンの Java でも搭載される予定の並列コレクションは，リスト処理を簡単な記述で実現できる機構である．コレクションに適用するメソッドはラムダメソッドやメソッド参照によって与えられ，その各要素に対する実行順序はユーザから隠蔽される．また，要素へのメソッド呼び出しは自動的に並列実行されるため，ユーザは実行順序を考慮することなく効率的なタスクスケジューリングの達成を期待できる．

しかし，単純に並列コレクションを利用するだけではステンシル計算を十分に記述することは難しい．まず，並列コレクションでは全ての要素に対して同一の処理を施すことが前提とされているため，領域毎の処理の違いを表しにくい．加えて，走査順序の隠蔽によってブラックボックス化されているため扱いやすい一方で，領域毎での処理の依存関係や通信最適化を考慮したコードの試行錯誤には向いていない．これらの理由により，ユーザはアプリケーションの記述や最適化を容易に行うことができない．

本研究では Java 言語に部分メソッドディスパッチと precedes 修飾子を導入することで，並列コレクションを利用したステンシル計算の問題を解決する．部分メソッドディスパッチは，メソッドシグネチャに記述した部分領域を条件とした述語ディスパッチ（predicate dispatch）によって，部分領域毎にメソッドをオーバーライドできる機構である．部分領域での全てのメソッド呼び出しをオーバーライドすることはもとより，領域中で一度だけ呼び出したいメソッドを定義することで，データ交換などの前処理をステンシル計算の本体から分離して管理できる．部分領域に応じて定義したメソッド間の実行順序は precedes 修飾子を用いて指定する．その際には，実行時のコンテキストによって位置が変化する部分領域間の順序制約も記述可能である．これにより，ユーザはコレクション中の要素の走査順序を，領域毎の依存関係を考慮

しながら制御できる．

以下，2 章では並列分散化したステンシル計算のプログラムにおける最適化の難しさを具体的に示す．3 章で関連研究を挙げ，4 章では並列コレクションによるステンシル計算の実装の利点，欠点について述べる．5 章では 4 章で挙げた問題点を解決するための機構として部分メソッドディスパッチと precedes 修飾子を導入した言語，MoCoJ を提案し，続く 6 章では提案言語のコンパイラの実装手法について述べる．7 章で提案言語で記述可能な最適化手法による効果を調べるため実験を行い，最後に 8 章で本論文をまとめる．

第 2 章

並列分散化したステンシル計算プログラムの最適化

2.1 ステンシル計算の並列分散化

HPC（High Performance Computing）分野におけるシミュレーションプログラムの多くでは，空間は格子状に分割した配列として表現され，その各要素（セル）上で隣接要素の値を使って物理演算を行うステンシル計算が頻繁に現れる．セルは個々にその地点でのインクの濃度や風の強さなどの物理量を表す値を持ち，配列中の全セルで時刻ステップ毎の物理量を反復的に計算し，状態を更新する．各セルでの計算は独立しているため，マルチコアやスーパーコンピュータを利用した並列コンピューティングによる恩恵を受けやすい．

一方，高速化のために並列分散させたステンシル計算では，単純な反復計算だけでなく隣接ノードとのデータ通信も必要になる．例えば，二次元水面に垂らしたインクが拡散する様子をシミュレーションする場合のステンシル計算のコード例をリスト 2.1 に示す．問題領域中の殆どのセル上では，その上下左右のセルの値との平均値で値を更新するという単純な処理を行う．しかし，並列分散化したプログラムが扱う問題領域は分割され，複数ノードに割り当てられる．各ノードは，近傍セルが存在するセル群（中心領域）では時刻ステップ毎に同一のステンシル計算を行うが，隣接ノードとの境界点ではセルの更新に必要な近傍セル（袖領域）が存在しないため，計算前に隣接ノードとのデータ交換を行わなければならない．

リスト 2.1. セル上での計算例

```
1 /* calculate on cell[i][j] */
2 double next = ( cell[i+1][j] + cell[i-1][j] +
3               cell[i][j+1] + cell[i][j-1] ) / 4;
```

2.2　通信最適化の煩雑さ

並列分散したステンシル計算では，典型的に通信最適化によって大幅な性能向上が見込まれる．最適化の過程では，効率的な通信のタイミングやキャッシュのヒット率を考慮した走査順序を決定するためにプログラム中のメソッド呼び出しの順番を複数回に渡って変更し，試行錯誤する必要がある．例えば通信最適化手法の一つとしてよく知られている通信オーバーラップは，隣接ノードとのデータ交換の際，非同期通信を開始した後，同期処理を行う前に交換しているデータに依存しない計算を行う．これにより通信の待ち時間を有効利用することになり，全体の実行時間を短縮できる．

さらに，通信回数を削減する最適化として，計算対象のデータを移動させる計算（シフト計算）や，空間，時間方向のループブロッキングが挙げられる [1]．ここで我々が用いるシフト計算という手法は grid compression 手法 [2] とも呼ばれ，配列中の要素の位置を時間ステップ毎に一つずつずらしながら同配列中に新しい値を上書きして，空間の状態を更新していくものである．従来の実装 (図 2.1) ではステップ毎に全ての隣接ノードと袖領域のデータを送信・受信する必要があるが，シフト計算を用いた実装では，例えば図 2.2 に示した例では右から左の方向にデータの流れが生じるため，片方向の隣接ノードへデータを与え，もう片方の隣接ノードから次ステップの計算に必要なデータを取得できるようになる．更に古い値，計算で得られた値は同じ一つの配列上で扱う事ができるので，ダブルバッファリングを行う必要が無い．このため，シフト計算を用いると通信だけでなくメモリ使用量も抑える事ができる．

図 2.1. 従来手法による実装

また，時間や空間ブロッキングをシフト計算と併用し計算を局所的に進めることで，片方向のデータ送受信の回数をさらに削減できる．時間ブロッキングは，値の更新に必要なデータが揃った箇所から優先的に計算を行い，一回のイテレーション中に前もって定めた時間ステップ分先まで状態を更新する手法である．一度に進む時間ステップ数に応じて配列サイズを大きく取れば，シフト計算により設定したステップ分の計算イテレーションを通信無しに行える．空

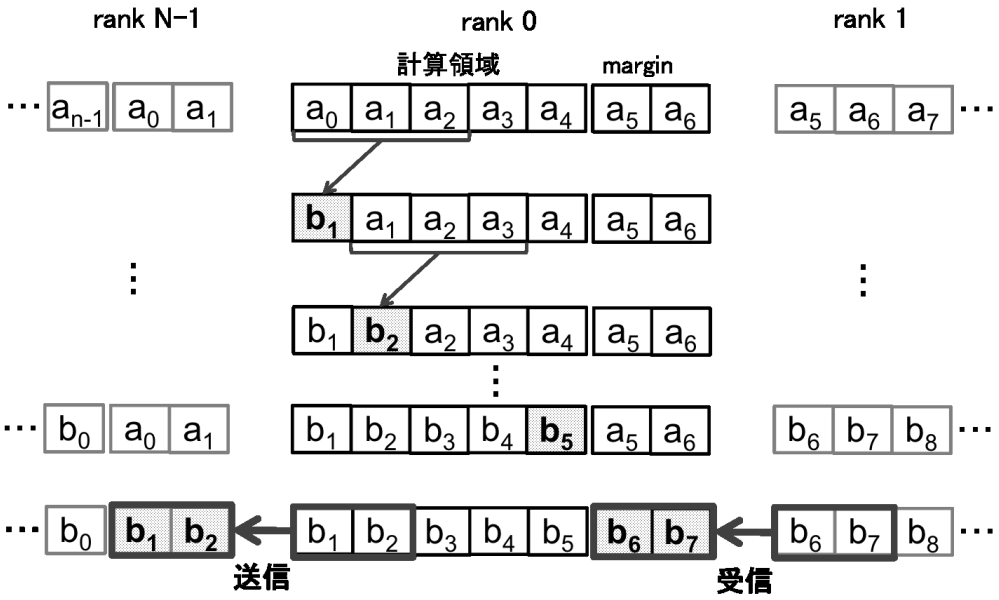

図 2.2. シフト計算による実装

間ブロッキングは，キャッシュサイズに応じてブロックサイズを定め，ブロック内の要素上で連続して計算する手法である．一般にシフト計算，時間ブロッキングと組み合わせて用いられ，キャッシュサイズ分連続した要素上で値を更新することで，メモリアクセスを局所化しキャッシュヒット率の向上にも寄与する．図 2.3 は，一次元の配列上でシフト計算，時間・空間ブロッキングを用いた場合の配列の状態遷移を表している．ここでは時間ブロックサイズ，すなわち通信無しで進むステップ数と空間ブロックサイズを共に 4 とし，それに伴って 8 要素分大きく配列を取っている．各セルの英字はそのセルが持つ値がどの時刻のものなのかを表す．グレーでハイライトしたセルは，次の行に状態が遷移する際に参照されるセル群である．まず 1 行目から 2 行目にかけてでは，空間ブロックサイズが 4 であることから，配列の先頭にある 6 つのセル (a_0〜a_5) を参照して，新たな 4 つの値 b_1〜b_4 を順に計算・更新を行っている．次に，引き続き値 a_4〜a_9 を用いて計算を行う事もできるが，先に今更新された値 b_1〜b_4 によって，更に 1 ステップ先の値 c_1，c_2 を計算しておく．このように，時間・空間ブロッキングでは必要なデータの揃った場所から優先的に計算を繰り返し行い，担当領域のセルの値が全てタイムステップ分更新されたところで，通信によりマージン部分の新たな値を取得する．

しかし，通信最適化によりコードが煩雑化すると，可読性や保守性が低下して更なる最適化の妨げになると共に，別アーキテクチャへの移植性が著しく低下する恐れが出てくる．プログラム中にはデータ通信，同期処理のコードが散在するだけでなく，対象セルが中心領域と袖領域のどちらに属するかを判定しながら領域毎に異なる処理を実装しなければならない．例えば，リスト 2.2 では，中心領域上で行う処理はセルの値更新のみであるが（7 行目），袖領域上では送信用データを用意した後に隣接ノードとのデータ交換を行ってから値を更新している

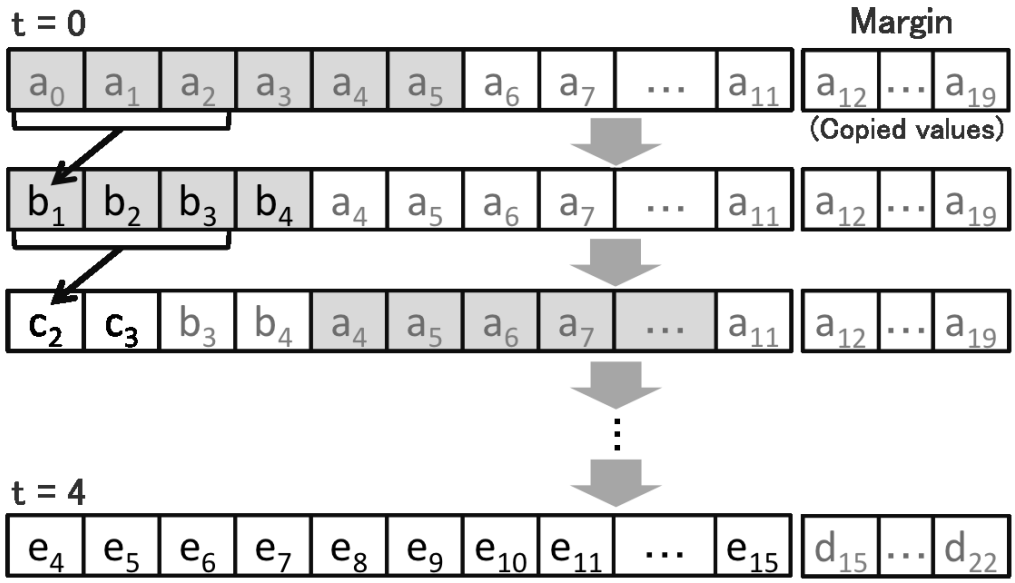

図 2.3. シフト計算と時間・空間ブロッキングを併用した実装

(2〜5 行目)．更に，時間ブロッキングや空間ブロッキングは，ループタイリング（loop tiling）やループ傾斜（loop skewing）のようなループ構造をまたがったコードの書き換えで実装される．そのような変更によって，配列の走査順序やブロックサイズはリスト 2.3 のように for 文のループ判定条件式中にハードコーディングされる．リスト 2.3 は，ループタイリングを利用した 3 次元ステンシル計算実装であり，直感的には時刻ステップを含めた 4 次元上に設定されたブロックサイズのセルを一括して更新する．そのために，9〜12 行目のメイン処理の for 文はブロックサイズ毎に分割され，1〜3 行目の for 文の処理ステップ bx，by，bz として現れる．ブロックサイズは 6〜8 行目において，最終的に到達する時刻ステップ t を使って導出している．このようなパラメータを変更しながらのチューニングはプログラマにとって大きな負担となる．また，for 文を用いた実装では要素の走査順序も線形に固定されるため，実行スケジュールを柔軟に指定することができない．

リスト 2.2. 領域毎の処理の違い

```
1 if (/* 袖領域 */) {
2   pack();
3   exchange();
4   unpack();
5   calculate();
6 } else if (/* 中心領域 */) {
7   calculate();
8 }
```

リスト 2.3. ループタイリングを用いたステンシル計算のコード例

```
for( int bz = 0; bz < MaxZ ; bz += bsz )
  for( int by = 0; by < MaxY; by += bsy )
    for( int bx = 0; bx < MaxX; bx += bsx )
      for( int t = 0; t < tmax; t++) {
        tc = tmax - t - 1;
        xMin = …; xMax = …;
        yMin = …; yMax = …;
        zMin = …; zMax = …;
        for( int z = zMin; z < zMax; z++ ){
          for( int y = yMin; y < yMax; y++ ){
            for( int x = xMin; x < xMax; x++ ){
              /* Calculate on cell(x, y, z) */
}}}}}}}
```

第 3 章

関連研究

3.1 先行研究

本研究には先行研究として，Java を拡張した言語 PersianJ [3] が存在する．PersianJ は，ユーザの指定したメソッドと，それらの依存関係から適切な計算順序を生成，反映する言語である．一般に，通信オーバーラップやキャッシュヒット率を考慮したプログラムの最適化の際には，プログラマはメソッド間の依存関係を注意深く考慮しなければならない．しかしその一方で，計算順序 (メソッドの呼び出し順序) の制約には必要な制約と不必要な制約があり，容易に判断するのは難しい．PersianJ は，プログラマがメソッド間の計算順序を変更するとき，プログラムの性能に直接関係する計算順序に集中できるようにすることで，プログラムチューニング時のプログラマの負担を軽減する．

PersianJ では，メソッド内の計算順序が自動的に決定されるメソッドとして，dispatch 修飾

リスト 3.1. dispatch メソッドによる拡散方程式の記述例

```
1  public dispatch void sor(DealDataE east,
2                           DealDataW west,
3                           CalcCells calc, ...){
4      // 通信処理を終えてからデータを格納
5      async east.exchange() precedes east.arrange(result);
6      async west.exchange() precedes west.arrange(result);
7
8      calc.calcOuter() follows east.arrange(),
9                               west.arrange();
10      :
11     // 中心領域，袖領域の計算結果を必要とする計算
12     calc.calcMaxResid(inner, outer)
13                  follows calc.calcInner() => inner,
14                          calc.calcOuter() => outer;
15      :
16     conditionChecker.checkBreak(maxResid)
17                  follows calc.calcMaxResid() => maxResid;
18 }
```

子の付いたメソッド (dispatch メソッド) を導入している．プログラマはリスト 3.1 のように，この dispatch メソッド内に計算順序の制約を必要な分だけ記述する．この時，計算の依存関係に伴う計算順序の制約は直接記述するが，プログラマがよりパフォーマンスの良い計算順序を考え出すこともあるため，プログラムの性能に関する順序関係を直接記述するかはプログラマが選んでよい．プログラマが記述しない場合には，コンパイラが自動的にある程度パフォーマンスの良い計算順序を生成する．また，本質的な依存関係がないメソッド間においては，dispatch メソッド内で計算順序を記述しなくて良い．

メソッド間の計算順序の制約は，precedes/ follows 構文を使って記述する (リスト 3.1)．dispatch メソッドでのメソッド呼び出しの順序は，dispatch メソッド内に記述された precedes / follows 文に基づいて決められ，メソッドはそれに従って実行される．あるメソッドを他のメソッドより先に実行する場合には，precedes 文を用いて 5 行目のように記述する．ここで，east.arrange(result) によって，east.exchange メソッドの返り値を east.arrange メソッドに渡している．逆に，あるメソッドを他のメソッドより後に実行する場合には follows 文を使い，例えば 12〜14 行目のように記述する．ここでは，calc.calcInner メソッド，calc.calcOuter メソッドの返り値を calc.calcMaxResid メソッドに引数 (inner, outer) として与えている．

更に，PersianJ では同期処理を必要とするメソッドを指定する async 修飾子も導入している．プログラマがリスト 3.1 の 5，6 行目のように async 修飾子を用いてメソッドを指定すると，コンパイラが計算のオーバーラップについて考慮したうえで，同期処理の回数が最小になるように計算順序を決定する．例えばリスト 3.1 のようにコードを記述した場合，実行時にはリスト 3.2 と同様の動作が得られる．

PersianJ で扱っている問題は，計算順序という点では本研究と同じであるが，PersianJ ではメソッドそのものの間の依存関係，MoCoJ コンパイラではコレクション中のセル間の依存関係を扱うという点で異なる．部分領域毎の実行順序を処理と共に制御することで，MoCoJ では PersianJ に比べ，より細やか且つ柔軟にプログラムの動作をチューニングできる．

3.2 predicate dispatch を利用したツール

3.2.1 Merge フレームワーク

D. Linderman らが提案した Merge フレームワーク [4, 5] は，ヘテロジーニアスなクラスタ環境に向けたプログラミングモデルを提供している．Merge はライブラリシステムと map-reduce パターンを導入したライブラリ指向の並列プログラミング言語，コンパイラとランタイムの 3 つの要素から構成される．Merge が所有するライブラリシステムにはアーキテクチャ毎に用意された最適な関数群が用意されており，その中では predicate dispatch の利用により，実行アーキテクチャに応じて呼び出す関数を変化させることができる．各アーキテクチャに応じた API は，EXOCHI と呼ばれる既存の環境を使用して，幅広いヘテロジーニアスなアーキテクチャに渡って生成，実装されている．

リスト 3.3 の 4〜6 行目のようにユーザが Merge の提供する言語を用いてコード中に

リスト 3.2. 通信オーバーラップを考慮した実装

```
public void sor(DealDataE east, DealDataW west,
         CalcCells calc, Buffer buffer,
         DealCondition conditionChecker){
  while(true){
    // 非同期通信開始
    MPJ_AsyncResult eastResult = east.exchange();
    MPJ_AsyncResult westResult = west.exchange();

    // 可能な部分では通信を計算とオーバーラップ
    double innerResult = calc.calcInner();

    eastResult.sync(); // 同期処理
    westResult.sync();
    east.arrange(eastResult); // 通信で得たデータを格納
    west.arrange(westResult);

    double outerResult = calc.calcOuter(); // 袖領域での計算
    :
    :
  }
}
```

predicate を記述すれば，コンパイラは自動的にアーキテクチャとの対応付けを行い，様々な環境に向けて最適化された関数群の中から，実行時に対象のアーキテクチャにとって最適な関数を呼び出す．またこのとき，より複雑で効果的なマッピングを行うために API を生成することもある．これにより，ユーザはアーキテクチャに依存しない形でプログラムを記述でき，環境毎に最適な実装を選ばずに済む．

リスト 3.3. Merge プログラムの記述例

```
bundle km2 : public X3000 {
  void kmdp(Array2D<float> dp, ...) {
    // Dispatch control predicate
    predicate(dp.rows < 5000);
    predicate(dp.cols == 8 && cc.rows == 4);
    predicate(arch == X3000_arch);
      :
}};
```

また，Merge では少しずつ異なるバージョンの関数を統括して扱えるため，プログラムを Merge を用いて記述した場合，幅広いヘテロジーニアス環境において性能が飛躍的に向上する可能性がある．実際に性能実験では，8-core 32-thread Intel Graphics and Media Accelerator X3000 に対して 3.6 から 8.5 倍のスケールアップ，a homogeneous 32-way Unisys SMP system に対して 5.2 から 22 倍のスケールアップを実現している．

Merge は，本研究の MoCoJ コンパイラとは predicate dispatch を利用している点で共通して

いる．しかしながら，Merge での predicate dispatch はアーキテクチャの名前やコア数などのスペックを条件に，プログラムが動作する環境に応じて実装を切り替えるものである．そのため，Merge では本研究で提案するような柔軟な領域指定や順序関係はサポートしていない．

3.3 自動通信最適化機能を持つコンパイラ

3.3.1 bamboo

Nguyen らの提案する Bamboo [6] は，MPI 混じりの C 言語で書かれたコードからデータドリブンな実行モデルを持つコードを生成する変換器であり，元のコードに直感的なアノテーションをいくつか追加するだけでコード変換を行える．アノテーションはデータの送受信処理やメインの計算を記述した部分に，リスト 3.4 のように付加する．まず，コード中で通信オー

リスト 3.4. Bamboo が提供するアノテーションを付加したプログラムの記述例

```
1  MPI_Init(&argc, &argv );
2    :
3  #pragma bamboo olap
4    for ( it=0; it<num_iterations; it++ ){
5      #pragma bamboo send{
6        /* pack boundary values to message Buffer */
7        MPI_Isend(/* send ghost cells to left, right, up, down */);
8      }
9      # pragma bamboo receive {
10       MPI_Recv (/* Receive Ghostcells from left, right, up, down */);
11       /* unpack incoming data to ghost cells */
12     }
13     MPI_Waitall();
14     for ( j=1; j < N/numprocs_Y-2; j++ )
15       for ( i=1; i < N/numprocs_X-2; i++ )
16         V(j,i) = c * (U(j,i+1) + U(j,i-1) + U(j+1,i) + U(j-1,i));
17     swap (U, V);
18 }
19   :
20 MPI_Finalize();
```

バーラップが行える可能性のある部分は Olap-region として 3 行目のようにアノテーションを付加する．Olap-region 内では send 処理，recive 処理を行っている箇所にそれぞれアノテーションを付加，処理をブロック化する事で区別し (5，9 行目)，コンパイラに通信に関する情報を与える．Olap-region 内且つ Send ブロック，Receive ブロックに属さない箇所は計算処理を行う Compute ブロックと認識される．

Bamboo の変換器は，記述されたアノテーションを元に指定された領域 (olap-region) 内のコードについて通信オーバーラップが可能かどうかを推定し，出力コードに反映する．更にデータレイアウトを指定するディレクティブも用意されており，より繊細な最適化も施すことができる (リスト 3.5)．例えば 3 次元空間に対するヤコビ法の場合，その y 次元，z 次元のデー

タのデータ分解はリスト 3.5 の 4，5 行目のように記述できる．ここで利用されているレイアウト NN は，Bamboo に元から用意されているレイアウトである．しかしながら，bamboo で

リスト 3.5. データレイアウトを指示するディレクティブ

```
1 #pragma bamboo dimension NumProcGeomDims
2 #pragma bamboo olap (layout name (args)* )*
3
4 #pragma bamboo dimension 2
5 #pragma bamboo olap layout NN Y layout NN Z
```

適用される最適化は通信オーバーラップとデータレイアウトに関するもののみであり，またアノテーションを付加した領域内でしか最適化が行われない．そのため領域内での実行順序を柔軟に指定することはできず，本研究とは方向性が異なる．

3.4　ステンシル計算向け DSL

ステンシル計算や偏微分方程式向けに特化した専用言語 (DSL : Domain Specific Language) も数多く提案されている．しかし，多くの DSL では，シミュレーションに関わるオブジェクトや実行環境の抽象化を重視するため，本研究とは異なり反復計算自体は抽象化していない．そのため，依然として C や Java のような手続き型言語と同様の制御ループを用いてステンシル計算を記述する必要がある．

3.4.1　Liszt

DeVito らの提案する Liszt [7] は，Scala を拡張して実装された可読性の高い並列分散 DSL である．Liszt は頂点，エッジ，面，セルで構成されるメッシュを抽象化した構造体を持ち，ユーザは簡単な命令でメッシュに含まれるセルにアクセスできる．Liszt ではセルの持つ値はフィールドで管理する．フィールドはメッシュ内の各要素上のデータを持つオブジェクトで，実際には各要素を値にマッピングするような連想配列として実装されている．例えばリスト 3.6 は，平面上のある一点に温度を与えて，それがどのように拡散して行くかをヤコビ法で解くプログラムの例である．コード中では，まずセルを表すフィールドを生成し，位置と温度の初期値を与えている (1〜4 行目)．その後，6〜9 行目でメッシュ中のセルを検査し，ある一点に温度を設定する．ただしこの時，メッシュの走査順序をユーザが明示的に指定することはできない．また，メッシュの辺に関して走査を行うことも可能である．11 行目以降の while 文では，メッシュの各辺についてその両端のセルを取得し，位置と温度情報を用いて新たな値を計算・更新している．

Liszt では DSL の提供に加え，コンパイル時には記述されたコードを一度分析し，並列性や局所性を考慮した上で MPI，CUDA，pthreads の各プラットフォームに対応したコードを生成する．これにより，ユーザはバックエンドを意識すること無く，同一のプログラムを多様な環

リスト 3.6. Liszt を用いて記述した PDE ソルバ

```
val Position
    = FieldWithLabel[Vertex, Float3](“position”)
val Temperature
    = FieldWithConst[Vertex, Float](0.f)
  :
for ( v <- vertices(mesh) ) {
  if(ID(v) == 1) Temperature(v) = 1000.0f
  else Temperature(v) = 0.0f
}
var i=0;
while ( i < 1000 ) {
  for ( e <- edges(mesh) ) {
      val v1 = head(e)
      val v2 = tail(e)
      val dP = Position(v2) − Position(v1)
      val dT = Temperature(v2) − Temperature(v1)
  }
    :
}
```

境へ簡単に移植することができる．

Liszt ではコードの変換を行う際，コンパイル時にメモリアクセスを静的に解析している．メモリアクセスの見積もりは，問題分割の方法や並列化の可能な部分を推測した上での計算の並列化手法の決定，ステンシルの形状の推測，各フィールドの状態の変化 (read-only, write-only, reduce-using-operator) の推測によって行う．与えられた PDE プログラムの並列化手法には，対象とする環境に応じて 2 種類のアプローチを用意している．分散メモリ環境に関しては，メッシュ分割法を適用する．メッシュ分割法は，メッシュを分割し，分割した境界をまたがる辺の数を最小化することで，ノード間通信を最小限に抑える手法である．一方共有メモリ環境においては，メッシュ彩色法を適用する．共有メモリ環境では，データへの書き込み衝突を防ぐため各処理を適切にスケジューリングする必要がある．そのためメッシュ彩色法では，同じデータへの書き込みを行う処理について，それらが一つの辺を共有する干渉グラフを生成して塗り分けることで，並列実行が可能な計算を推定する．ただし，PDE ソルバの並列性を正しく推定する為には，使用されているステンシルの形状を静的に把握できなければならない．静的推定が可能であることを保証するため，Liszt によるプログラムには，プログラム実行中にメッシュ構造をはじめ，メッシュの要素やフィールドの値は不変でなければならないなどの制限が課せられている．

3.4.2 XcalableMP acceleration device extension

李らが提案する XcalableMP acceleration device extension (XMP-dev) [8] は，並列プログラミング向け言語である XcalableMP [9, 10] のアクセラレータ向け拡張である．XMP-dev を用

いれば逐次コードに対してわずかな変更を施すだけで並列コードを作る事ができる．XMP の言語仕様に加えてホスト・デバイス間のデータ転送やループ文の並列化等の処理を OpenMP ライクな指示文で記述できる．リスト 3.7 は，デバイス上で配列の各要素の総和を求めるコード例である．デバイス上での変数宣言には 3，4 行目のように device replicate，device allocate 指示文を使う．device replicate は変数をホストとデバイスの両方で確保し，device allocate は変数をデバイス上でのみ確保する．ホスト・デバイス間のデータ転送が必要な場合には，device allocate で宣言された変数に対しては gmove 指示文を挿入する (20 行目)．この時，コンパイラは状況に応じてホスト・デバイス通信に加えノード間通信も生成する．一方 device replicate によって宣言した変数には device replicate_sync 指示文を用いる．各処理の実行に関する指定も可能である．あるループ文の各イテレーションが独立して実行できる事が分かっている場合には，8 行目や 14 行目のように loop 指示文で並列実行を指示できる．またこの他にも，計算を行うデバイスの指定や，bcast，reduction，reflect といったノード間通信が必要な処理をデバイス同士で直接行う事もできる．

リスト 3.7. XMP-dev を用いたコード例

```
1  #pragma amp align [i] with t(i) :: a, hb, db
2  #pragma xmp shadow a[*]
3  #pragma xmp device replicate (a)
4  #pragma xmp device allocate (db)
5  int a[N], hb[N], db[N];
6
7  void main(void) {
8    #pragma xmp loop on t(i)
9      for ( int i=0; i<N; i++) a[i] = i+1;
10   #pragma xmp reflect (a)
11
12   #pragma xmp device replicate_sync in (a)
13
14   #pragma xmp loop on t(i)
15     for ( int i=0; i<N; i++) {
16       db[i] = 0;
17       for ( int j=0; j<N; j++) db[i] += a[j];
18     }
19
20   #pragma xmp gmove
21     hb[:] = db[:];
22 }
```

3.4.3　Chapel

Chapel [11, 12] は，Cray による高生産性を目指した分散メモリ型並列計算機用の言語である．タスク並列をサポートしており，1 つのスレッドが実行を開始しした後，実行中の任意の場所で新しいタスクを sync 文や cobegin 文によって生成できる．また，並列ループ文 (forall)

もサポートされている.

Chapel ではデータの位置についての透過性が極めて高く，自プロセスのデータ、他プロセスのデータを文法上で区別しない．そのため，ユーザは遠隔ノードにあるオブジェクトや配列を, ノードの境界を意識することなくアクセスできる. ただし，参照は透過されているものの, 実際のデータや計算の配置は locale というオブジェクトを用いて明示的に記述する必要がある．locale は，あるプログラムの実行に参加している計算ノードを表すオブジェクトである．計算やデータ (変数) は，リスト 3.8 の 7 行目のように on 構文で特定の locale に移動して実行する.

加えて，一次元の列 (range)，多次元の矩形 (domain) を表すデータ構造も用意されている．これらと for 文を組み合わせて用いると，多次元矩形に対するデータの並列操作や，分割操作などを記述できる．配列は一般に domain から値への写像として定義されており，一次元配列, 多次元配列，連想配列などが全て配列として扱える．更に，Domain を多数の locale にマッピング (分割) したものとして mapped domain というデータ構造もサポートされている．これを domain とする配列は分散配列として利用できる (1〜4 行目).

リスト 3.8. Chapel のコード例

```
1 const TableDist = new damp(new Block[0..m-1]),
2       UpdateDist = new damp(new Block[0..N_U-1]));
3 const TableSpace: domain ⋯ dmapped TableDist = ⋯,
4       Updates: domain ⋯ dmapped UpdateDist = ⋯;
5   :
6 forall ( ,r) in (Updates, ⋯) do
7   on TableDist.idxToLocale (⋯) {
8     /* calculate using TableSpace and Updates */
9 }
```

第 4 章

並列コレクションを用いたステンシル計算の実装

4.1 Java におけるラムダメソッドと並列コレクション

scala や C# をはじめ，多くのプログラミング言語に取り入れられているラムダ式が，今後リリースされる Java にもラムダメソッドとして導入される [13, 14, 15]．これは，Java7 で導入された Fork-join [16] の機能をより発展させ，逐次実行プログラムと記述上で近い並列プログラミング機能をライブラリレベルで提供することを目的とするものである．

ラムダメソッドは，計算の処理内容を抽象化し，コレクションの走査順序を内包化できる機構である．例えば，リスト 4.1 のように，あるコレクション中の各要素に対して同一の処理やフィルタリングを行った場合を考える．リスト 4.1 は，大きさと重さの異なる立方体のリスト

リスト 4.1. リストに対して for 文を使って走査を行う場合

```
1 List<Box> boxes = …;
2 double maxWeight = 0.0;
3 for (Box b : boxes) {
4   if (b.size == 100)
5     if (b.weight == maxWeight) {
6       maxWeight = b.weight
7     }
8   }
9 }
```

に対して，一辺の大きさが 100 であるものの内，最も重い立方体の重さを計算するものである．ここではリストを走査する順序は表面化しており，走査順序はユーザが for 文中で自ら制御しなければならない．また，for 文の性質から通常走査順序はシリアルなものになってしまい，並列化には向かない．更に，イテレーションの最中に重さの最大値を保持する maxWeight 変数は可変であるためスレッドセーフでない．このことも for 文を使って並列化を行うことの欠点である．

上で述べた欠点は，差し当たっては匿名クラスを利用すれば解決できる．リスト 4.1 は，匿名クラスを用いてリスト 4.2 のように書き直せる．ここでは，boxes に対して filter メソッド

リスト 4.2. 匿名クラスを用いたコレクションの走査

```
1  ExtendedList<Box> boxes = ··· ;
2  double maxWeight =
3    boxes.filter(new Predicate<Box>() {
4      public boolean op(Box b)  {
5        return b.getSize() == 100;
6      }
7    }).map(new Mapper<Box, Double>() {
8      public Double extract(Box b) {
9        return b.getWeight();
10     }
11   }).max();
```

(3〜7 行目)，map メソッド (7〜11 行目) を適用している．filte メソッド，map メソッドそれぞれの引数にはその場で生成した匿名クラスを渡し，具体的な処理の内容を指示している．これにより，変数 maxWeight の値はリストに対するメソッド呼び出しの連なりで得られている．匿名クラスの利用により，コレクションの走査順序は filter メソッドや map メソッドの内部に隠蔽され，それらの定義に一任される．また，これらメソッドの定義によって，要素の並列走査，遅延評価も可能である．更に，今回変数 maxWeight の値はこのコード片の実行中には変化しないため，スレッドセーフと言える．

しかしながら，匿名クラスを使った場合，リストに対して一つの処理を行う度に多くの場合はここに挙げた以上の記述が必要になり，可読性が著しく低下してしまう．また並列実行と逐次実行を変更したい場合や遅延評価を適用したい場合など，プログラムの挙動を変更する際には逐一ライブラリを切り替えなければならないという制約も存在する．

リスト 4.2 を，更にラムダメソッドを用いて書き換えたものがリスト 4.3 である．ラムダメ

リスト 4.3. ラムダメソッドを用いたコレクションの走査

```
1  List<Box> boxes = ··· ;
2  double maxWeight =
3    boxes.filter((Box b) -> b.getSize() == 100)
4         .map((Box b) -> b.getWeight())
5         .max();
```

ソッドを利用することによって，匿名クラスを使った場合に比べて以下の点が改善される．特にラムダメソッドによる計算の抽象化によって可読性が向上し，それに伴ってプログラム全体の挙動を追い易くなる．

- コード全体の可読性が上がる
- 計算の抽象度が高くなる

- 可読性の向上によりエラーの発生を抑えられる
- 可変な変数を利用しない
- 並列化が容易に行える

ラムダメソッドは匿名関数と同様の機能であり，型付きの引数と返り値の型，メソッドボディを持つ (3，4 行目)．一方ラムダメソッドは関数型インターフェースのオブジェクトを生成する記述のシンタックスシュガーと捉える事もできる．関数型インターフェースは，一つのメソッドのみを持つインターフェースである．例えば，以下のような記述は，

```
Predicate<Box> p = new Predicate<Box>() {
  public boolean op(Box b)  {
    return b.getSize() == 100;
  }
};
```

ラムダメソッドを利用した次の記述と同値である．

```
Predicate<Box> p = (Box b) -> b.getSize() == 100 ;
```

また，ラムダメソッドの引数部分の型は省略可能となっている．これは，ラムダメソッドから生成される関数型インターフェースのメソッドシグネチャをコンパイル時に参照することにより，ラムダメソッド中の引数の型を推測できるためである．これに加え，Java においてはメソッド参照を用いた記述も可能であり，例えば以下のラムダメソッドは，

```
FileFilter x = (File f) -> f.canRead();
```

メソッド参照を用いて次のように書き直せる．

```
FileFilter x = File::canRead;
```

並列コレクションは，ラムダメソッドやメソッド参照と共に Java8 から導入される機構であり，ラムダメソッド，メソッド参照との併用によりリスト処理を簡潔に記述できる．Java8 には，Stream API [17] として実装される予定である．Stream を用いた並列コレクションの使用例をリスト 4.4 に示す．リスト 4.3 との主な相違点は，2 行目において，立方体のコレクション boxes に対して stream メソッドを使用している点である．これにより，boxes を Stream オブジェクトに変換して利用している．コレクションに stream メソッドを作用させて得られる Stream オブジェクトはストレージのようなデータ構造ではなく，要素の供給元 (データ構造やジェネレータなど) からデータを取得するための構造である．Stream オブジェクトを通して得られた要素に対して操作を行っても，元のデータそのものに変更は加えられない．例えば，リスト 4.4 の 2，3 行目のように Stream オブジェクトに対して filter メソッドを作用させ，コレクション中の特定の要素を取り出す時，filter メソッドの返り値は新たな Stream オブジェクトとなっている．Java のコレクションはそのままでは有限数の要素しか扱えないが，コレク

リスト 4.4. stream と共に並列コレクションを利用した場合のコード例

```
1 Collection<Box> boxes = ... ;
2 int sum = boxes.stream()
3                .filter(b -> b.getColor() == RED)
4                .mapToInt(b -> b.getWeight())
5                .sum();
```

ションを stream に変換して用いることで，無限個数の要素に対応できる．無限個の要素を扱うにあたり，要素は遅延評価される．

Stream オブジェクトに対する操作には，次の 2 種類が存在する．

- intermidiate operation
- terminal operation

intermidiate operation の返り値は Stream オブジェクト，terminal operation の返り値は単一の値である．例えば，filter メソッドや map メソッドは intermidiate operation，forEach メソッドや reduce メソッドは terminal operation である．(map メソッドと forEach メソッドの違いは，intermidiate operation か terminal operation かという点だけである．) intermidiate operation は常に遅延評価される．またこれらのメソッドには，上で述べたラムダメソッドやメソッド参照を渡すことができる．

Stream オブジェクトに対する処理はデフォルトでは逐次実行されるが，

```
int sumOfWeights = boxes.parallelStream()
                        .filter(b -> b.getColor() == RED)
                          :
```

のように Collection クラスのオブジェクトに対して parallelStream メソッドを作用させることで，その後の処理を並列実行に切り替えられる．また，逐次実行を明示的に指定したい場合には，BaseStream クラスに parallel メソッド，sequential メソッド共に用意されているため，次のような記述も可能である．

```
int sumOfWeights = boxes.parallelStream()
                        .filter(b -> b.getColor() == RED) // 並列実行
                        .sequential()
                        .mapToInt(b -> b.getWeight) // 逐次実行
                        .parallel()
                        .sum(); // 並列実行
```

これにより，処理の一部を並列実行，一部を逐次実行と，実行形態を使い分けることができる．

4.2　並列コレクションによるステンシル計算実装の問題点

上で述べた並列コレクションを利用するとステンシル計算を簡潔に記述できる可能性がある．並列コレクションによる実装では，for 文による実装に比べ以下のような点が改善される．

- コレクション内の走査順序が隠蔽され，可読性，保守性が高まる
- メソッド参照の利用により，要素上で計算を行うメソッドが簡単に指定できる

例えばリスト 4.5 に示したように，セルの並列コレクション localgrid が保持する全 Cell オブジェクトに対するステンシル計算は，それを実装した calculate メソッド参照を map メソッドに与えるように記述できる．ループ文による素朴な実装と比較すると，各要素上での計算がリスト内包表記で記述され，コードブロックの形で分離される．全セルに対する calculate メソッドの呼び出し結果の最大値は，max メソッドによるリダクション処理で求められる．したがって，二次元水面に垂らしたインクの濃度変化が閾値を下回るまで，シミュレーションの時間ステップを進めるようなコードが簡潔に記述できる．更に map メソッドが生成する calculate メソッドの実行順序は隠蔽され，スケジューラによって自動的に並列実行されるため，ユーザは実行順序を意識することなく効率的なタスクスケジューリングの達成も期待できる．

リスト 4.5. 並列コレクションを使ったコード例

```
1 Grid<Cell> localGrid = new Grid<Cell>(...);
2 do {
3   localGrid.exchange(); // 隣接ノード間でデータを交換
4   double max = localgrid.parallelStream()
5                         .map(Cell::calculate) // 全セル上で計算
6                         .max();
7 } while (max < THREASHOLD) // 定常状態で終了
```

しかし，単純に並列コレクションを利用するだけでは，並列分散化したステンシル計算をうまく扱う事は以下に挙げる理由によってやはり困難である．

- コレクションの特定の要素に対するメソッドの使い分けが難しい
- ユーザがコレクションの走査順序を簡単に制御できない

並列コレクションでは，全要素に対して同一の処理を適応することが前提であるため，通信を要するか否かといったセル毎の処理の違いが表しにくい．例えば，リスト 4.6 のようにステンシル計算に if 文を追加し，担当領域の部分領域毎に場合分けを行えば，袖領域では通信によるデータ交換と計算を行い，中心領域では計算処理のみ行うという所望の動作が期待できる．しかしこのような記述では領域判定の記述が直接埋め込まれ，保守性，可読性が低くなってしまう．加えて， Java 8 の並列コレクションの map メソッド側においても，複数のコードからな

るブロックを引数として渡すことはできない.

リスト 4.6. 並列コレクションによる領域毎に異なる処理の記述

```
1  Grid<Cell> localGrid = new Grid<Cell>(...);
2  do {
3    localGrid.exchange(); // 隣接ノード間でデータを交換
4    double max = localGrid.parallelStream()
5                          .map(
6        if (/* 中心領域 */) {
7          calculate(); // 値の更新のみ行えば良い
8        } else if (/* 袖領域 */) {
9          pack(); // 送信用データを用意
10         exchange(); // データ通信
11         unpack(); // 受信したデータを配列に格納
12         calculate(); //値を更新
13       }
14   ).max();
15 } while (max < THREASHOLD) // 定常状態で終了
```

また，並列コレクションはライブラリとしてブラックボックス化され利便性が高い一方で，利用者自身が走査順序を工夫してプログラムを最適化するようなシチュエーションは想定されていない．例えば，通信オーバーラップには，前もって袖領域でのデータ交換を開始し，通信が終わるまでの間に中心領域で新しい状態を計算するような実行順序が期待される．同様に，時間，空間ブロッキングにも，前述のように要素間の実行順序と，それを決定するための各方向のブロックサイズといった，要素間の複雑な依存関係を考慮したコードをループの制御文中にハードコーディングする必要がある．しかし並列コレクションでは通信処理や実行順序は隠蔽されているため，これらの最適化はライブラリ内部のコードを利用者が直接改変して実装しなければならず，ユーザによるプログラムの最適化は現実的ではない.

第 5 章

実行順序が制御可能な部分メソッドによる並列コレクションの拡張

5.1 部分メソッドディスパッチの導入

本研究では，並列コレクションを利用したステンシル計算のために，領域毎に異なる計算や走査順序をユーザが柔軟に定義するために部分メソッドディスパッチ（partial method dispatch）を導入した Java 言語拡張，Mosaic Collection for Java (MoCoJ) を提案する．部分メソッドディスパッチを備えた MoCoJ では，袖領域，中心領域といった部分領域毎に異なる処理を実行する複数のメソッド定義（部分メソッド）を同じメソッド名で定義できるようになり，セルが属する領域毎の微妙な計算の違いを簡潔に記述できる．部分メソッドディスパッチとは，同名のメソッドに対してメソッド呼び出しを行う時，レシーバの型や状態に応じて実際に呼び出すメソッドボディを動的に切り替える述語ディスパッチ（predicate dispatch）と呼ばれる機能を部分領域の指定に特化させた言語機能である．部分領域を特定するために，メソッド引数として渡される配列インデックス値に対するパターンマッチを行い，領域毎にメソッドをオーバーライドできる．オーバーライドしたメソッド内からは，元のメソッド default.[**メソッド名**] として呼び出すことができる．これにより，メソッドを呼び出した Cell オブジェクトがどの領域に属するかによって，そのセル上で行う処理の上書きや拡張が可能となる．例えば，図 5.1 のように，袖領域に対する calculate メソッドではセルの値更新の前にデータ通信を行い，中心領域に対する calculate メソッドではセルの新しい値の計算のみ行うよう記述すると，任意の Cell オブジェクトに対して calculate メソッドが呼び出された時にそのセルが属する領域に応じた適切な振る舞いを得ることができる．さらに，メソッドオーバーライドで部分領域を定義させることで，親クラスが定義した部分領域を継承し上書き再定義する事もできる．そのため，並列コレクションへ受け渡すメソッド参照を，サブクラスのそれに切り替えれば部分領域やその実装を図 5.2 のように容易に切り替えられる．

部分領域は，メソッドの引数に対するパターンとして指定する（リスト 5.1）．メソッド呼び出し時には，オーバーライドされた部分メソッド群のうち，引数の値がパターンと一致したメソッドが呼び出される．例えば，大きさが MaxX* MaxY であるような二次元配列にお

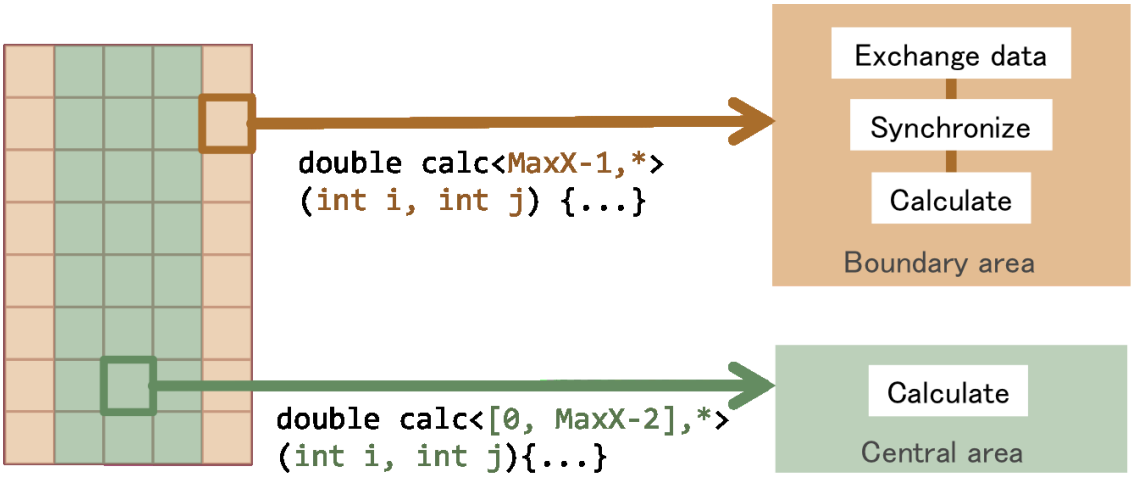

図 5.1. 部分領域毎に異なる処理内容を持つメソッド

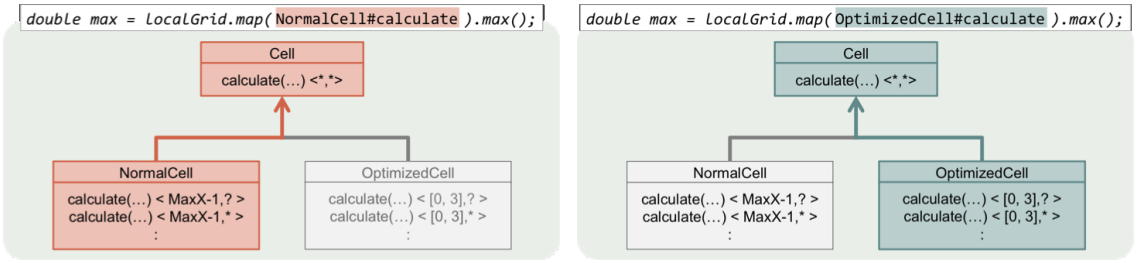

図 5.2. メソッド参照の切り替えによる実装の変更

いて，部分領域 <MaxX-1, [0, MaxY-1]> は配列の右端一列を表す．また，袖領域 <MaxX-1, [0, MaxY-1]> の全セル上でのメソッド呼び出しは，ワイルドカードを用いて，<MaxX-1*, [0, MaxY-1]*> と記述する．これは省略形として <MaxX-1*, *> または <MaxX-1, *> とも記述できる．袖領域内での通信処理のように一度だけ呼び出したいメソッドは“？”を用いて，<MaxX-1?, [0,MaxY-1]?> のように記述する．ここでも同様に省略形として <MaxX-1?, ?>，<MaxX-1, ?> という記述も可能である．

5.2 precedes 修飾子の導入

MoCoJ では部分メソッドディスパッチに加え，ユーザが部分領域間での実行順序を明示的に与えるための precedes 修飾子を導入し，コレクション中の要素の走査順序を制御できるようにした．実行順序は，部分メソッド定義の領域指定に続けて < **メソッドが適応される部分領域** > precedes < **ブロックする部分領域** > のように指定する．例えば，<MaxX-1, *> precedes <*, *> とすれば，右端の袖領域がそれ以外の領域での部分メソッドの実行をブロックする．また，precedes とワイルドカード“？”を使った部分領域を組み合わせて用いれば，部分領域での計算の前処理，後処理をメインの計算から分離して記述できる．例えば，リスト 5.2 のよう

リスト 5.1. 部分領域の記述

```
1  double calculate(int i, int j)
2      <MaxX-1, [0, MaxY-1]> {
3    // default の calculate メソッドを呼び出す
4    return default.calculate(i,j);
5  }
6
7  /* default method */
8  double calculate(int i, int j)
9      <[1, MaxX-2], [0, MaxY-1]> {
10   ……
11   double curr = cell[i][j];
12   double next = (cell[i+1][j] + …… + cell[i][j])/6;
13   cell[i][j] = next;
14   return Math.abs(curr-next);
15 }
```

に，袖領域でのデータ送受信処理を前処理として分離して定義できる．precedes による実行順序の指定がない領域は，並列コレクションのスケジューラによって適宜実行される．

リスト 5.2. 前処理の分離

```
1  /* 袖領域(First entry) */
2  double calculate(int i, int j)
3      <MaxX-1,?> precedes <MaxX-1, *> {
4    ……
5    Request[] reqs;
6    if (eastID >= 0) {
7      pack(); // 送信データを準備
8      reqs = exchange(…); // データ通信
9      Request.Waitall(reqs); // 同期処理
10     unpack(…); // 受信データを配列へ格納
11   }
12   return default.calculate(i,j);
13 }
14
15 /* 袖領域 */
16 double calculate(int i, int j) <MaxX-1, *> {
17   return default.calculate(i,j);
18 }
```

さらに，precedes 修飾子を用いると，実行時コンテキストに依存して変化する動的な領域間の順序を指定することもできる．例えば，先述のようにステンシル計算では，任意のセルの計算には，1 ステップ前の隣接セルの計算結果が必要なのでその計算終了まで待つ必要がある．一方，時間と空間のブロッキングを考慮した場合，さらにブロックサイズに応じて時間方向と空間方向で依存関係は動的に変化する．本研究では，このような動的な部分領域についてもリスト 5.1 と同様にメソッドの仮引数を用いてメソッドシグネチャで指定し，その順序付けを定

義可能とした（リスト 5.3）．これによって，実行時の引数の値を基底とした相対的な部分領域とその依存関係を表現できる．例えばリスト 5.3 では，任意のセル（位置 x，時刻 t+1）での値更新には，自身と右隣 2 つのセルにおいて 1 ステップ前の計算が終わっていなければならないという順序制約を記述している．ただし，ブロックサイズに基づいた動的な順序関係は，ブロックサイズ分の計算を進める毎に空間方向と時間方向とで動的に切り替わる上，領域記述も非常に煩雑になるため，現時点では，走査順序を決定するブロックサイズは並列コレクションやコンパイラに外部から与えられるようなシステムを検討している．現在の仕様では，動的な部分領域は，計算領域全体に対する部分メソッド（default メソッド）のみに記述できる．

リスト 5.3. コンテキストに応じて位置が変動する領域の実行順序の指定

```
1 // default の calculate メソッド
2 double calculate(int x, int t)
3     <[x, x+2], t> precedes <x, t+1>{
4   return stencil(x); // cell[x]上で値を更新
5 }
```

第 6 章

MoCoJ コンパイラの実装

6.1 JastAddJ によるコンパイラ拡張

部分メソッドディスパッチ機能を備えた Java 言語，MoCoJ は，JastAddJ [18] を用いた Java のコンパイラ拡張によって実装する．JastAddJ は，コンパイラのコンパイラである JastAdd [19, 20] によって実装された Java コンパイラ拡張フレームワークである．JastAddJ では，以下に挙げるファイルに拡張した構文規則や抽象構文木の変換規則を追記するだけで Java を拡張することができる．

- flex ファイル: 追加する文法で使用されるキーワードを指定するファイル
- ast ファイル: 抽象構文木における各ノード (AST ノード) を表すクラス群の継承関係を定義するファイル
- parser ファイル: 構文解析のアルゴリズムを記述するファイル
- jrag ファイル: AST を構成するノードを表す各クラスに対して，新たなメソッドを追加したり，あるノードを他の種類のノードと交換するなどの処理を記述するファイル

本研究においては，JastAddJ に部分領域と precedes 修飾子それぞれの文法を追加し，コンパイル時には図 6.1 に示したように，precedes によって記述された部分領域間の依存関係から実行順序を解析して，その結果を元に抽象構文木を標準の Java で構成された構文木に変換する．これについては以下で詳細に説明を加える．また現時点では未実装であるが，解析時には，部分的な順序のシリアル化やメソッド呼び出しのインライン展開を行うことで素朴な実装よりも良い性能のコードを生成することを予定している．

6.1.1 AST ノードと構文木生成規則の追加

今回新たに作った AST ノードと，そのノードが継承するノード，保持する子ノードについて，ast ファイルに記述した定義と共に述べる．ある AST ノードを表すクラスがどのクラスを継承するか，またどのようなノードを子ノードとして持ち得るかは以下のように記述できる．まず，MoCoJ の文法を用いて記述された部分メソッドと，それらをクラスメンバとして

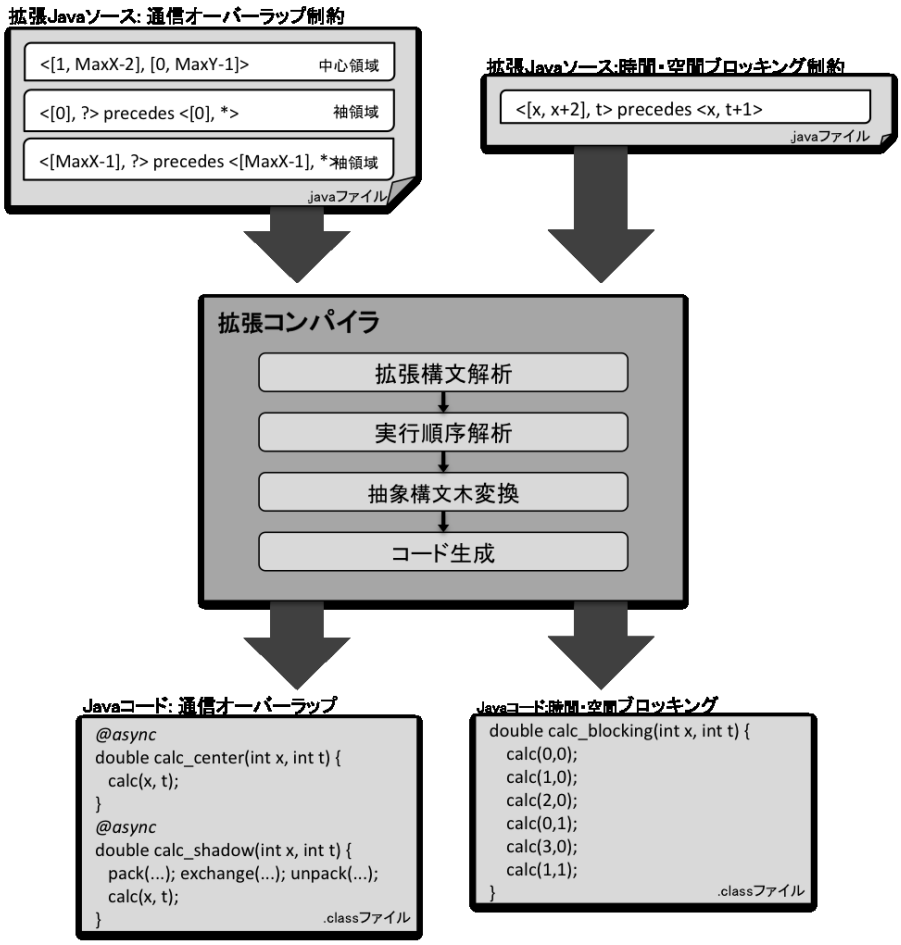

図 6.1. 拡張コンパイラの概要

持つようなクラスを表す AST ノードとして，MOSAIC_ClassDecl と MOSAIC_MethodDecl をリスト 6.1 のように定義した．これらは通常のメソッドやクラスの亜種と言えるため，それぞれクラス宣言を表す ClassDecl(11〜13 行目)，メソッド宣言を表す MethodDecl(15〜17 行目) を継承している．これらは jastAddJ に元から定義されている．従って保持する子ノードも ClassDecl，MethodDecl と共通するが，メソッドシグネチャに新たに記述する次の 2 つの情報を含まなければならない．

- 自メソッドが対応付けられている部分領域
- 自メソッドの担当する部分領域より後に走査される部分領域群

そのため，MOSAIC_MethodDecl はメソッドシグネチャに合わせ，部分領域とその部分順序の情報を子ノード MainArea，FollowsAreas として持っている．MainArea は，その部分メソッドに対応付けられた部分領域の情報を持つ．FollowsAreas は precedes 句によって指定された部分領域を表すノードである．MainArea は部分領域を表す SubArea クラスのオブジェクトである．一方 FollowsAreas は複数の部分領域の情報を持たなければならないため，SubArea

リスト 6.1. 部分メソッドとクラスの AST ノードクラス定義

```
1  MOSAIC_ClassDecl : ClassDecl ::=
2    Modifiers <ID:String> [SuperClassAccess:Access]
3    Implements:Access* BodyDecl*;
4
5  MOSAIC_MethodDecl : MethodDecl ::=
6    Modifiers TypeAccess:Access <ID:String>
7    Parameter:ParameterDeclaration*
8    MainArea:SubArea FollowsAreas:SubArea*
9    Exception:Access* [Block];
10
11 ClassDecl : ReferenceType ::=
12   Modifiers <ID:String> [SuperClassAccess:Access]
13   Implements:Access* BodyDecl*;
14
15 MethodDecl : MemberDecl ::=
16   Modifiers TypeAccess:Access <ID:String>
17   Parameter:ParameterDeclaration*  Exception:Access* [Block];
```

オブジェクトのリストとなっている．なお，MethodDecl は BodyDecl を継承しているため，MOSAIC_MethodDecl も BodyDecl として扱われる．

部分領域を表す AST ノードである SubArea クラス，部分領域の各次元の値の範囲を表す AST ノードである IndexScope クラスは，リスト 6.2 のように定義している．SubArea はメソッドシグネチャで言う所の

`<[1, MaxX-2], [0, MaxY-1]>`

にあたり，一方 IndexScope はその中の

`[1, MaxX-2]` **または** `[0, MaxY-1]`

にあたる．SubArea クラスは ASTNode を継承している．ASTNode クラスは全ての AST ノードクラスの元になる抽象クラスであり，特に継承したいノードのないクラスは ASTNode クラスを継承することになっている．部分領域では複数の引数に対する値の範囲を扱う必要があるため，SubArea クラスは IndexScope クラスのリストを子ノードとして持つ (1 行目)．一方 IndexScope クラスも ASTNode クラスを継承する．値の範囲を定義するには最小値と最大値が得られれば良いので，式を表す Expr クラス 2 つをそれぞれ子ノード From, To として持つ (2 行目)．実際には構文木の変換時に部分領域の属性 (*, ?) を判断するため，これら属性に対応した IndexScopeForAll クラス，IndexScopeForFirst クラスを定義して IndexScope を継承し，IndexScope を抽象クラスとして利用している．IndexScopeForAll クラス，IndexScopeForFirst クラスの子ノードは IndexScope クラスと同一であるので，子ノードの情報は省略している (3，4 行目)．

リスト 6.2. 部分領域を表す AST ノードクラス定義

```
1 SubArea: ASTNode ::= IndexScope*;
2 IndexScope : ASTNode ::= From:Expr To:Expr;
3 IndexScopeForAll : IndexScope;
4 IndexScopeForFirst : IndexScope;
```

新たに導入したノードを用いて，部分メソッド，すなわち部分領域と precedes 節を持つメソッド，クラスを含む構文木の生成規則を parser ファイルに追加した．追加した生成規則の一部をリスト 6.3 に示す．コンパイル時には，パーサが構文解析をする過程で各ノードを表すオブジェクトを生成し，構文木を作っていく．

リスト 6.3. 構文木の生成規則 (一部)

```
1  ClassDecl class_declaration =
2    modifiers.m? CLASS IDENTIFIER super.s?
3                       interfaces.i? mosaic_class_body.b
4    {: mod = new Modifiers(m);
5       id = IDENTIFIER;
6       return new MOSAIC_ClassDecl(mod, id, s, i, b); :};
7
8  IndexScope mosaic_indexScope =
9      LBRACK expression.e1 COMMA expression.e2 RBRACK MULT?
10           {: return new IndexScopeForAll(e1, e2); :}
11   | LBRACK expression.e1 COMMA expression.e2 RBRACK QUESTION
12           {: return new IndexScopeForFirst(e1, e2); :}
13   | LBRACK expression.e RBRACK  MULT?
14           {: return new IndexScopeForAll(e, e); :}
15   | LBRACK expression.e RBRACK  QUESTION
16           {: return new IndexScopeForFirst(e, e); :}
17   | MULT  {: return new IndexScopeForAll(null, null); :}
18   | QUESTION
19           {: return new IndexScopeForFirst(null, null); :};
```

6.1.2 構文木の変換

パーサーによって MoCoJ 独自の AST ノードを使った構文木が構成されたら，自分で定義したノード (MOSAIC_ClassDecl や MOSAIC_MethodDecl など) を標準の Java に用意されているノードに置換しなければならない．そのような操作 (rewrite) は，jrag ファイルに記述する．今回行う変換は，MoCoJ 特有の情報を取り除いた部分メソッドを，通常のメソッドとして置換するものである．ただし，その過程では部分領域毎の部分順序関係から全部分領域の実行順序を計算しなければならない．

構文木の変換は，

1. MOSAIC_ClassDecl の ClassDecl への置換

2. MOSAIC_MethodDecl の MethodDecl への置換

の 2 ステップで行う．以下でそれぞれのステップについて述べる．

■1. ClassDecl への書き換え　MOSAIC_ClassDecl の書き換えには以下の手順が必要である．

1. そのクラス内に存在する部分メソッドを把握して部分順序間の全順序を計算する．
2. 自身と同じ子ノードで ClassDecl オブジェクトを作り直し，自ノードと置換する．

rewrite 処理を行うコードをリスト 6.4 に示す．まず，自身が持つクラスメンバはリストとし

リスト 6.4. ClassDecl への変換

```
1  rewrite MOSAIC_ClassDecl {
2    to ClassDecl {
3      /* 1. generate the invokation order */
4        :
5      for(int i = 0; i<bodyDeclList.getNumChild...; i++){
6        BodyDecl b = ... /* get a class member */
7        if(b instanceof MOSAIC_MethodDecl){
8          String name = b.getMosaicName();
9          subAreaList.add(name);
10         Iterator<String> followsIter =
11                       b.getFollowsNameList().iterator();
12         while(followsIter.hasNext()){
13           /* Generate rules of partial orders */ }
14       }
15     }
16     ArrayList<String> sortResult =
17       topologicalSort(subAreaList.toArray(…), ruleList);
18     /* 2. return new ClassDecl obj with the same contents */
19     return new ClassDecl(this.getModifiers(),...);
20   }
21 }
```

て部分メソッド，それ以外の区別無く保持しているので，リストの各要素に対して instanceof 演算子を用いて部分メソッドであるかどうかを検査する (5〜7 行目)．MOSAIC_MethodDecl を見つけたら，そのメソッド固有の名前を getMosaicName メソッドにより生成し，部分領域を集めるリスト subAreaList に追加する (8，9 行目)．ある部分メソッド固有のメソッド名は，getMosaicName メソッドによって以下の規則に従って生成される．

- 固有メソッド名は，元のメソッド名の後ろに部分領域の各次元の範囲 (下限，上限) と，その部分領域の属性を付加させて生成する．各単語は_で区切るものとする．
- 属性を表す文字列は，*に対しては ALL，?に対しては FIRST とする．
- 部分領域の値の範囲を文字列にする際，演算子 −，+，*，/ はそれぞれ MINUS，PLUS，MULT，DIV と読み替える．
- 部分領域に一つの要素 expr だけで構成された次元がある場合，その部分は_expr_expr

と読み替える．

- 部分領域に省略された次元がある場合，値の代わりにその部分領域の属性を用いて_ALL_ALL または_FIRST_FIRST を付加させる．

例えばメソッド名が calc であり，部分領域 <[Max-2, Max-1]?,[MaxY-2, MaxY-1]?> を持つ部分メソッドの固有メソッド名は, calc_MaxXMINUS2_MaxXMINUS1_MaxYMINUS2_MaxYMINUS1_FIRST となる．10，11 行目では，現在注目している MOSAIC_MethodDecl の表す部分メソッドの後続の部分メソッド群をリストで取得しており，続く 12 行目の while 文では詳細は省略しているが，8 行目と同様に部分領域毎に固有のメソッド名を生成しながら部分順序のリスト ruleList を作っている．各リストは String または要素 2 つの String の配列のリストである．部分順序のリストを全ての MOSAIC_MethodDecl について作り終えたら，Java 向けのグラフライブラリ JGraphT が提供するトポロジカルソートを用いて全順序を生成する (16，17 行目)．これについては，詳細を次節で述べる．MOSAIC_ClassDecl の書き換えそのものは，元々 ClassDecl オブジェクトの生成に必要な子ノードを保持しているため，19 行目のように自ノードの持つ各子ノードをそのまま引数としてオブジェクト生成し，返り値とすれば良い．

■2. MethodDecl への書き換え MOSAIC_MethodDecl は自メソッドに対応付けられた部分領域と，precedes 節で指定された部分領域群の情報を持っているが，これらの情報は構文木上での 2 階層上に位置する MOSAIC_ClassDecl によって総括，利用されるため，そのメソッド固有のメソッド名を生成する以外に MOSAIC_MethodDecl 上で扱うことはない．従って，rewrite 処理の際には自身の部分領域から固有メソッド名を生成し，自身と同じ子ノードと新たなメソッド名を用いて MethodDecl オブジェクトを作り直し，自ノードと置換すれば良い (リスト 6.5).

リスト 6.5. ClassDecl への変換

```
1  rewrite MOSAIC_MethodDecl {
2    to MethodDecl {
3      /* 1. generate new method name */
4      String newName = getMosaicName();
5      /* 2. create normal MethodDecl object */
6      return new MethodDecl(this.getModifiers(), …, newName, …);
7    }
8  }
```

6.2 JGraphT を用いた実行順序の導出

6.2.1 JGraphT

各要素の実行順序は Java のグラフライブラリである JGraphT [21] を用いたトポロジカルソートによって導出する．トポロジカルソートとは，閉路のない有効グラフにおいて，全ての

頂点が自身から出る辺の指している頂点よりも先に走査されるように順番付けを行うソート方法である．JgraphT では無向グラフ、有向グラフをはじめとした様々なグラフを表現するクラス構造やそれらを用いたソーティング手段が用意されており，MoCoJ コンパイラではその中のトポロジカルソートを行うイテレータを利用している．

■JGraphT でのソーティングアルゴリズム　JGraphT が提供するトポロジカルソート向けイテレータでは，next メソッドによって要素を取り出す過程で，同時に残った要素のソートを進める．ソートは Kahn のアルゴリズム [22] に基づいて行われている．Kahn のアルゴリズムは，最初に取り出す頂点から考え始め，次に取り出すべき頂点を探していくというものである．まず，自身に入る辺を持たない頂点を保持する集合 S(実装時には，一般的にキューやリストが利用される) を用意する．集合 S に存在する頂点は，次に取り出される頂点の候補となる．集合 S と，ソートした解を保持するためのリスト L を用いて，以下のように頂点のソートを行う．

1. S から頂点を一つ選び，S から削除する．この頂点を v とする．
2. L に頂点 v を追加する．
3. 頂点 v から出る各辺 e に対して，以下の a)，b) を行う．
   a) 辺 e を元のグラフから削除する．
   b) 辺 e が指す先の頂点 w が，e 以外の辺に指されていなければ，頂点 w を S に追加する．
4. S が空であれば L を解として計算を終了する．そうでなければ手順 1 に戻る．

このアルゴリズムを用いたソートには，V を頂点数，E を全ての辺の数として，$O(|V|+|E|)$ の実行時間を要する．なお，各辺は一度しか訪問されないが，頂点は複数回訪問される可能性がある．

トポロジカルソートに利用する有向グラフは，例えばリスト 6.6 のように生成する．ここでは，まず 2，3 行目で空の有向グラフのオブジェクト graph を，9 行目〜10 行目で各部分領域と対応する頂点を用意している．その後，12〜14 行目，16 行目以降で，有向グラフを表すクラスに用意されている addVertex メソッド，addEdge メソッドによって頂点や頂点同士を繋ぐ有向辺を graph に追加している．

また、リスト 6.6 で生成した有向グラフに対して，リスト 6.7 のようにトポロジカルソートを施した順番でノードを取り出すことができる．この例では 2，3 行目でリスト 6.6 のコード中で生成した有向グラフ graph に対するトポロジカルソートを行うイテレータである TopologicalOrderIterator オブジェクトを生成し，6〜8 行目でこれを用いて各要素を取り出している．

リスト 6.6. 有向グラフの生成

```
// 新しい有向グラフを生成
DirectedGraph<String, DefaultEdge> graph =
  new DefaultDirectedGraph<String, DefaultEdge>(DefaultEdge.class);

// 各頂点を生成
String [] subArea = new String[6];
subArea[0] = "ALL_ALL";
subArea[1] = "0_FIRST";
subArea[2] = "0_ALL";
  :
// グラフに頂点を追加
for(int i=0; i<subAreas.length; i++){
    graph.addVertex(subArea[i]);
}
// グラフにエッジを追加
graph.addEdge(subArea[1], subArea[2]); // "0_FIRST"->"0_ALL"
 :
```

リスト 6.7. 生成した有向グラフに対するトポロジカルソート

```
// トポロジカルソートを行うイテレータを生成
Iterator<String> iterator =
  new TopologicalOrderIterator<String, DefaultEdge>(graph);

// トポロジカルソートを行った順番で要素を取り出す
while(iterator.hasNext()){
  System.out.println(iterator.next());
}
```

6.2.2 部分領域毎の実行順序の生成

MoCoJ で記述された部分領域間の実行順序を導出する際には，まず，MoCoJ コンパイラによって，部分領域間に指定された部分順序を，部分領域を表す頂点と有向辺から成る有向グラフへ変換する必要がある．

部分領域の有向グラフは，MoCoJ によってオーバーライドされた各部分メソッドのメソッドシグネチャから生成する．具体的には図 6.2 のように，メソッドシグネチャからその部分メソッドの担当する部分領域，precedes 節以降からその部分メソッドより後に実行しなければならない部分領域を取得し，各部分領域間の順序制約を表現するリストを生成する．得られたリストを元に有向グラフを作り，JGraphT のトポロジカルソートを適用することで順序制約を満たす実行順序を作成する．

ただし，リストを作る際，実際には部分領域から生成したメソッド名を頂点として扱っている．jastAdd によるノード変換の過程で，部分メソッドはそれぞれ部分領域に応じた固有のメ

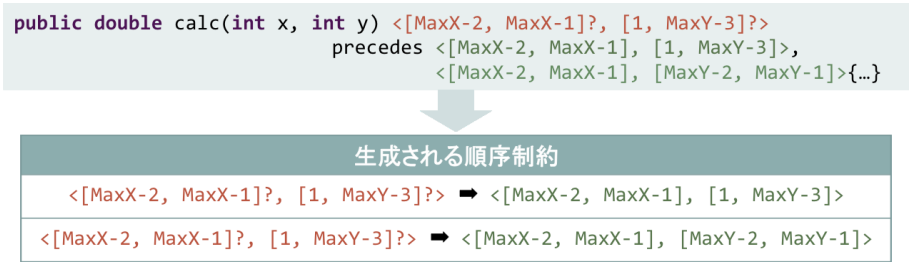

図 6.2. メソッドシグネチャによる順序制約の生成

ソッド名を与えられる．部分領域はこれらメソッド名と対応付けられているため，有向グラフの各頂点として利用できる．

6.2.3　空間・時間ブロッキング適用時の走査順序の生成

プログラムに時間ブロッキング・空間ブロッキングを適用する場合には，コンパイラに実行時パラメータとして各ブロックサイズやソーティングポリシーを与えるようにして，並列コレクションの走査順序を，時間・空間ブロッキングを適用した場合と等価にスケジュールされた実行順序へシリアル化できるようにする．この場合での各セルの実行順序の生成にも jgraphT を利用した．

トポロジカルソート実行時の結果は，一般に一意には定まらない．先の説明におけるリスト L に頂点を追加する際に，集合 S の保持する頂点が複数ある場合，どの頂点を選ぶかによって最終的に解となるリスト L 内の頂点の並びは微妙に異なったものになる．これは，集合 S が持つ各頂点の間には順序制約がないためであるが，どの頂点を選んでも解としては妥当なものである．ただし，空間・時間ブロッキングを適用する場合には，集合 S から頂点を選ぶ場合にも「できるだけ配列の先頭から，ブロックサイズに従う連続アクセスができるような順番で」頂点を選ばなければいけないという，リスト 5.3 に示した条件 (強条件) とは別の弱い条件が発生する．

JGraphT の TopologiclaOrderIterator クラスでは，コンストラクタに自作の PriorityQueue を実装したクラスのオブジェクトを渡す事でこのような弱い制約をトポロジカルソートに反映できる．まず，セルをタイムステップ数毎に区別しなければならないため，セルの情報を扱うクラスとして CellInfo クラスを定義した．CellInfo クラスから生成されるオブジェクトが持つ情報は，そのセルの x 座標，y 座標，時刻 t の三つである．今回自作した空間・時間ブロッキング向け PriorityQueue は，次に走査されるセル候補の集合 S の役割を担うことになる．そのため，セルをポールする時に上記で述べた条件

- できるだけ配列の左上にあるセルから選択する
- できるだけ指定されたサイズのブロック状にセル続けて選択する

を考慮して動作するよう実装した．

リスト 6.8. PriorityQueue のコンストラクタ

```
public ST_PriorityQueue(…) {
    Qs = new TreeSet[2];
    Qs[0] = new TreeSet<CellInfo>(new BFS_Comparator());
    Qs[1] = new TreeSet<CellInfo>(new DFS_Comparator());
      :
  }
```

実装した PriorityQueue のクラス, ST_PriorityQueue クラスではリスト 6.8 に示すように, 幅優先探索，深さ優先探索を行うための TreeSet オブジェクトをそれぞれ一つずつ持ち，状況によって切り替えながら探索を行う．次に選択するべきセルを含む新たなブロックを探す際，既に計算が進んでいる部分で更にタイムステップを進める場合には，ブロックの先頭セルは配列の左上方向にあるものから選ぶ必要があるため深さ優先探索を行う．一方，そのようなブロックが確保できない場合には，幅優先探索を行って t の値が小さいセルを選択する．各 TreeSet オブジェクトが持つ要素群は完全に一致しなければならない．そのため，ST_PriorityQueue クラスに対して要素の削除や追加が依頼された時にはどちらの木にも削除，追加が行われるように各メソッドをオーバーライドした.

リスト 6.9. poll メソッド

```
public CellInfo poll() {
  /* ブロックの中盤であれば続きのセルを出力 */
  if(!toBePolledList.isEmpty()){
    CellInfo toBeNext = toBePolledList.poll();
    TreeSet<CellInfo> tailSet = Qs[1].tailSet(toBeNext, true);
    CellInfo next = tailSet.pollFirst();
      :
    return next;
  }

  /* 次のブロックの先頭になるセルを取得 */
  CellInfo headOfBlock = Qs[1].first(); // 深さ優先探索
  prepareList(headOfBlock); // toBePolledListの中身を充填

  /* headOfBlockを先頭とするブロックを作れなかった場合 */
  if(toBePolledList.isEmpty()){
    CellInfo newHead = Qs[0].first(); // BFSの先頭から取る
    prepareList(newHead); // toBePolledListの中身を充填
      :
    return newHead;
  }
    :
  return headOfBlock;
}
```

弱条件を考慮した走査を行う poll メソッドをリスト 6.9 に示す．priorityQueue からセルを

一つ取り出す時の状態には次の 3 状態が挙げられる．

1. 現在ブロックに含まれるセルを出力している最中であり，
   次も同じブロック内のセルを選択すれば良い．
2. ブロック一つ分のセルを出力し終わった所であり，
   タイムステップを進めるブロックが確保できる．
3. ブロック一つ分のセルを出力し終わった所であり，
   タイムステップを進めるブロックが確保できない．

リスト 6.9 では，まず 3〜8 行目が状態 1 での処理を行っている．if 文の条件分岐で使われている toBePolledList は，現在出力しようとしているブロックに含まれるセルのリストである．toBePolledList にまだ要素が残っているならば状態 1 であるとわかる．toBePolledList から次のセルを取り出した後，そのセルと同じ情報を持つセルを TreeSet から取得，削除しなければならない．この時，TreeSet オブジェクト内の要素間の関係が壊れないようにするため，目的のセルを根とする部分木を取得し，その木の先頭をポールするようにしている (5，6 行目)．

状態 1 でなければ，ブロックが出力された直後である．この時には状態 2 であると仮定して，次に取り出すセルの候補群を深さ優先探索でソートしてある TreeSet からセルを選び，そのセルを先頭としたブロック分のセルが確保できるかどうかを prepareList メソッドにより検査する (12，13 行目)．prepareList メソッドは，ブロックの先頭となるセルを受け取り，1 ブロック分のセル群 (の情報) を toBePolledList に格納する (リスト 6.10)．まず 8〜12 行目で，

リスト 6.10. prepareList メソッド

```
1  private void prepareList(CellInfo head) {
2    int startX = …;   int endX = …;
3    int startY = …;   int endY = …;
4    int = head.getTime();
5
6    while(endX>0){
7      // 候補のブロック
8      for(int y=startY; y<=endY; y++){
9        for(int x=startX; x<=endX; x++){
10         toBePolledList.offer(new CellInfo(x, y, t));
11       }
12     }
13     if(this.containsAll(toBePolledList)){ …  return; }
14     endX--; // ブロックの幅を縮める
15       :
16   }
17 }
```

与えられたブロックサイズと先頭セルの情報から，同一ブロック内のセルを表す CellInfo オブジェクトを新しく生成する．続く 13 行目で，今生成したセル群が TreeSet に含まれているかどうかを調べ，含まれていれば計算を終了する．含まれていない場合には，ブロックの幅を 1 縮

めて再度ブロック内セル群を生成し，TreeSet に含まれるかを検査する (14 行目)．prepareList メソッドを呼び出したものの，toBePolledList が空で返ってきた場合には，タイムステップを進めるべきブロックが確保できなかった事になる．そのため，状態 3 として処理を行う (リスト 6.9 の 16〜21 行目)．この処理は殆ど状態 2 と同じであるが，今度は幅優先探索によってセル群をソートした TreeSet オブジェクトの先頭セルを元にブロックの情報を用意する．

なお，今回作成した PriorityQueue は，2 次元平面上で配列の左上方向にシフトを行う場合を想定している．3 次元空間の問題に対しては非対応であるが，3 次元の問題でシフト計算を行うと通信回数がシフト計算を行わない場合より増加してしまうため，少なくとも In-Place 手法においては 3 次元配列を用いた実装は現実的な手段ではないと考えている．

第 7 章

実験

第 5 章で示したように，MoCoJ では部分メソッドディスパッチの導入により，コードに対する最適化を述語を使って宣言的且つ簡潔に記述できるようになった．しかし，MoCoJ コンパイラによって導出されるコードの実行時性能の善し悪しは，言語の記述性の高さとはまた別の話である．そこで実験では，実際に MoCoJ の述語を用いて書いた最適化によってプログラムの実行効率が向上するかどうかを，MoCoJ を用いて記述した最適化コード，素朴なコード間の実行性能の差を調べることで確かめた．

7.1 実験概要

実験に向け，複数ノード間で分散化した拡散方程式を解くプログラムを，MoCoJ のコンパイラが生成するコードを想定して，最適化を施すもの，施さないものを合わせて以下の 6 種類用意した．実行時間の計測は，予備実験をローカル上で，本実験 (マイクロベンチマーク) を FX10 [23] 上でそれぞれ行った．なお，6 種類のコードの中にはノード内で並行実行するもの，しないものがあるが，ノード間での通信は全ての場合で行っている．

- S1. シリアル実行 (最適化無し)
- S2. S1 + シフト計算
- S3. S1 + シフト計算 + 空間・時間ブロッキング
- P1. ノード内並行実行
- P2. P1 + シフト計算
- P3. P1 + シフト計算 + 通信オーバーラップ

(各コードについて期待される高速化の程度を述べる) 上記 6 種類のコードの間では、最適化によって通信回数が変化する．S1 と P1 では担当領域に対して上下左右に隣接するセルを持つ 4 つのノードとデータの送信・受信を行うため，1 タイムステップあたり計 8 回の通信を行っている．一方 S1，P1 以外の実験コードでは左上向きのシフト計算を採用しているため，データ交換に伴う通信は担当領域の上，左上，左の三方向への送信，下，右下，右の三方向からの受信の合わせて 6 回行う．従って，S2，P2，P3 の 1 タイムステップあたりの通信回数は 6 回

である．S3 では更に空間・時間ブロッキングを行うため，1 イテレーションで 4 ステップ進んでいる．そのため，1 タイムステップあたりの通信回数は，6 ÷ 4=1.5 回と見なすことができる．本実験では，このような通信回数の変化や通信オーバーラップの効果が実行時間に現れることが好ましい．

リスト 7.1 は，実験コード S1 での通信処理，全セルに対する走査，計算を含むイテレーション部分を示したものであり，この記述は 6 種類の実験コード間で殆ど共通している．実験では，全実行時間のうちリスト 7.1 に示した do-while 文にかかる時間のみを計測した．

リスト 7.1. 実験コード S1 でのイテレーション部分

```
1    :
2  Grid grid = new Grid(xNumber, …);
3    :
4  double delta;
5  double[] maxDelta = new double[1];
6  do {
7      grid.exchange(); // 4方向へ，計8回の通信を行う
8      // 通信が終わったら全セルを走査・計算
9      delta = (Double)grid.mapToDouble(func)
10                          .max(); // 値の変化のうち最大のものを取得
11
12     // 全ノード間でのdeltaの最大値を取得
13     double[] sendDelta = {delta};
14     MPI.COMM_WORLD.Allreduce(sendDelta, 0, maxDelta,
15                                 0, 1, MPI.DOUBLE, MPI.MAX);
16
17     grid.moveOn(); // ダブルバッファリング
18 } while ( maxDelta[0]>THRESHOLD );
19   :
20   :
```

リスト 7.1 では，具体的には以下に述べるように拡散方程式を解いている．まず 2 行目で，自ノードが担当するセル群を含んだ並列コレクション，Grid クラスのオブジェクト grid を生成する．ただし並列コレクションはまだリリースされていない機能である為，Grid クラスは ArrayList を拡張して作った自作のコレクションである．コレクションに対して行われる処理の並行化は，並行プログラミング用 API として用意されている Concurrency Utilities [24] を用いて行っている．6 行目から始まる do-while 文で，拡散方程式を解く計算のイテレーションを表現する．実験コード S1 では通信と計算はオーバーラップさせないため，7 行目で前もって通信を開始している．通信オーバーラップを行う実験コード P3 では，部分領域毎に通信処理を行うためこの 1 行は記述していない．また通信処理は，例えばリスト 7.1 における exchange メソッドでは，リスト 7.2 のように Isend，Irecv 命令を用いて記述している．これら通信命令はローカル実行時には MPJ Express[25, 26, 27]，FX10 上での実行時には OpenMPI Java Binding[28, 29, 30, 31] を利用して記述した．これら 2 つの API はそれぞれ記述様式は同じで例外の扱いが異なるのみであるので，ローカルまたは FX10 上でプログラムを実行する際

には殆ど同じソースコードで対応する事ができる．

リスト 7.2. 通信部分の実装

```
1  public void exchange() throws MPIException {
2      :
3    /* 1. 上方向の通信 */
4    // send (上へ)
5    double[] aboveSendBuf = new double[ xSize+2 ];
6    for (int i = 0; i < xSize + 2 ; i++) {
7      aboveSendBuf[idxCounter++] = get(i, 1).getVal();
8    }
9    pareReqs[0] = MPI.COMM_WORLD.Isend(aboveSendBuf, 0, ...);
10   // receive (下から)
11   double[] belowRecvBuf = new double[ xSize+2 ];
12   pareReqs[1] = MPI.COMM_WORLD.Irecv(belowRecvBuf, 0, ...);
13   // 一度通信
14   Request.Waitall(pareReqs);
15     :
16     :
17   /* 受信したデータを配列に反映 */
18   // 1. 上
19   for (int i = 0; i < xSize + 2; i++) {
20     get(i, 0).setReceivedVal(aboveRecvBuf[idxCounter++]);
21   }
22     :
23 }
```

9 行目では，grid 中の全セルに対して計算と値の更新を行っている．このとき各セルの新旧の値の差のうち最も大きいものを 10 行目で取得し，続く 13〜15 行目で全ノード間での値の差の最大値を得ている．ここで mapToDouble メソッドの引数 func は，セルのオブジェクトに関するダブルディスパッチを行う為のインターフェースであり，インターフェースクラスはリスト 7.3 の 2〜6 行目，リスト 7.1 で使用しているオブジェクト func は 9〜11 行目のように定義している．このように生成した func を与えた mapToDouble メソッドを grid に対して呼び出す事で，最終的には func に定義された doItToDouble メソッドが呼び出され，呼び出し元を変更せずとも，セルそれぞれ固有の calc メソッドを実行できる．

最後に，17 行目で現在着目している配列をスイッチする．これも S2，S3 ではシフト計算によりダブルバッファリングの必要がないため，この一行は記述していない．

7.2　予備実験

イテレーション回数に対するおおよその実行時間と，コード毎の実行速度の傾向を調べるため，実験コード S3，P2，P3 について，それぞれローカル (Mac OS X (10.7.5)，1.8GHz Intel Core i7(4 コア)， Java 1.6.0) で予備実験を行った．図 7.1 は，イテレーション回数 900 回，利用ノード数 3 × 3 とした時の実験結果である．実験コード f においては，通信から通信までの

リスト 7.3. FunctionIF の実装

```
/* インターフェースの定義 */
public interface FunctionIF {
  public double doItToDouble(Cell c) throws InterruptedException;
  public boolean filter(Cell c);
    :
}

/* 利用時に各メソッドを適宜オーバーライド */
FunctionIF func = new FunctionIF(){
  @Override
  public double doItToDouble(Cell c) throws InterruptedException{
    return c.calc();
  }
  @Override
  public boolean filter(Cell c){ return c.getVal()>5.0; }
    :
};
```

間に進むタイムステップ数を 3 とし，do-while 文を 300 回行う事で全体のイテレーション回数を 900 としている．空間ブロックの大きさは 3 × 2 とした．また P2，P3 では，タスクの並行実行時にはスレッドプールを環境に合わせて 4 つ生成している．

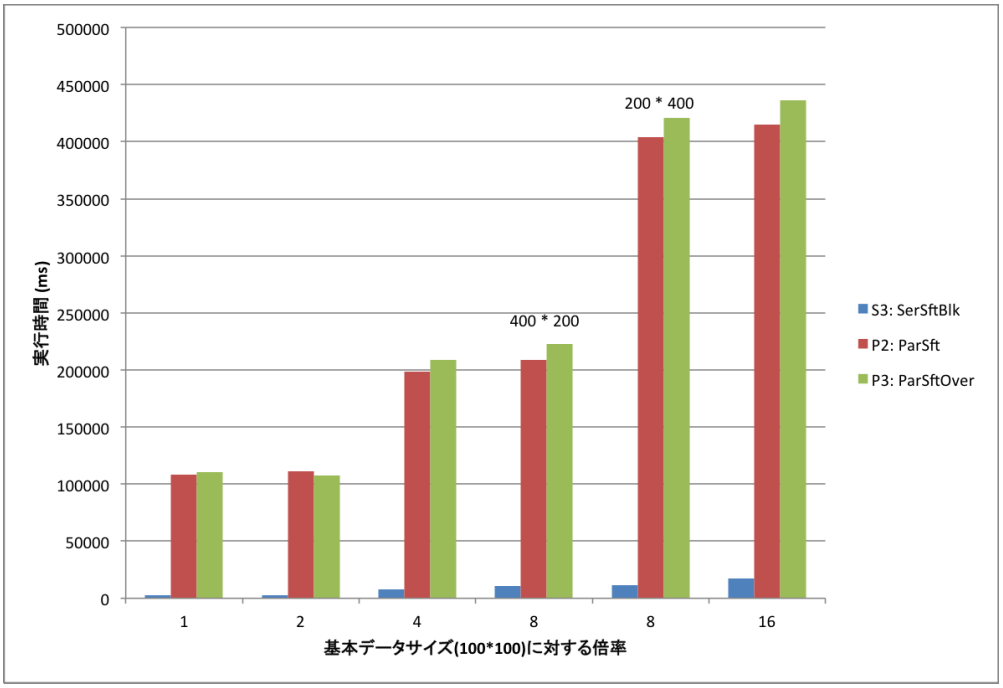

図 7.1. 予備実験の結果

実行結果としては，ノード内での計算がシリアル実行であるにも関わらずコード S3 の実行

速度が他の二つより著しく高かった．対してタスクを並行処理している P2，P3 は期待した実行速度を得られなかった．二つを比べると，通信オーバーラップを採用した P3 が，P2 よりも最大 6.9 %程度遅い結果となった．S3 の通信回数は P2，P3 の通信回数の三分の一であるため，これらの実行速度の差は通信時のオーバーヘッドに起因しているものと思われる．P3 では各方向への通信が部分領域毎に分散しているので，P2 以上のオーバーヘッドがかかっていると考えられる．また，問題サイズが 400 × 200，200 × 400 の場合については，セルの総数は同じ 80000 であるものの，x 方向に長い問題より y 方向に長い問題に，P2, P3 はほぼ二倍の実行時間を要した．

7.3　FX10 上での実験

FX10 上での実験では，各コードの実行時間の計測値を元に，MoCoJ を用いて最適化を施したコードが素朴な実装 (S1，P1) に比べてどの程度高速化されたかを算出した．実行時には，拡散方程式を解く総タイムステップ数を 120 とし，ノードを 3 × 3=9 ノード使用した．時間・空間ブロッキングを使用している S3 については，時間方向のブロックサイズを 4 とし，30 回のイテレーションを行うことで総タイムステップ数を他の 5 種と合わせている．空間方向のブロックサイズは 10 × 10 とした．なお，実行時間の増減率の計算には，各コードを 5 回ずつ実行した平均の実行時間を用いている．

全体で扱う問題サイズを 100 × 100，1000 × 1000 とした場合の結果を図 7.2，7.3，7.4，7.5 に示す．各系列に付加した数値は，5 回分の実行時間の平均値 (ms) である．逐次実行の場合は問題サイズが 1000 × 1000 の時に比べ 100 × 100 の時に，より大きな割合で実行時間が削減されている (図 7.2，7.3)．問題サイズが大きい場合に実行時間の削減率が低下してしまうのは，一つのノードが扱うセル数は問題サイズの大小によって 100 倍変化するため，1000 × 1000 のケースにおいて a) データの送受信時に，交換するセルの数が多くキャッシュに乗り切らずキャッシュヒット率が低い，b) 空間・時間ブロッキングを行う S3 においては，1 イテレーションでアクセスするセルが多く，やはりキャッシュヒット率が低いことが原因として考えられる．一方，ノード内で並行実行を行う場合には，元々問題サイズが 100 × 100 の場合でも実行時間の削減率が低いだけでなく，問題サイズを大きくすると，P2，P3 共にほぼ P1 と同じ実行時間を要している (図 7.4，7.5)．前者は，ノード内での計算並行化のオーバーヘッドが，シフト計算や通信オーバーラップによる恩恵を打ち消してしまっていることに起因すると考えられる．後者に関しては，並行化オーバーヘッドに加え，問題サイズが増加したことによって逐次実行するコードと同様にデータ交換時のキャッシュヒット率が低下してしまうためと考えた．

ここまでの実験において，ノード内並行実行を行うコードに対する実験結果からはコード最適化の効果が実行時間に現れているとは言い難いと考えた．そこで，更に通信オーバーラップの効果が出易いように問題サイズを 1000 × 100，100 × 1000 といった細長い領域として追加実験を行った (図 7.6，7.7)．問題サイズを横長 (1000 × 100) にした場合には，シフト計算と通信オーバーラップによって 10 %程度の高速化が達成されている．通信によって交換される

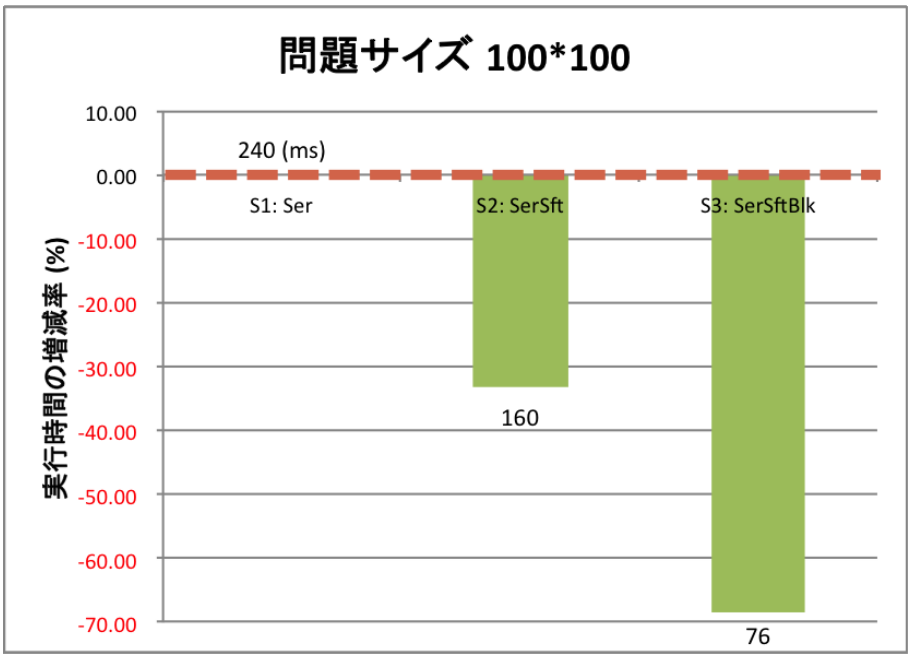

図 7.2. コード S1-S3 を問題サイズ 100*100 で実行した場合の実行時間増減率

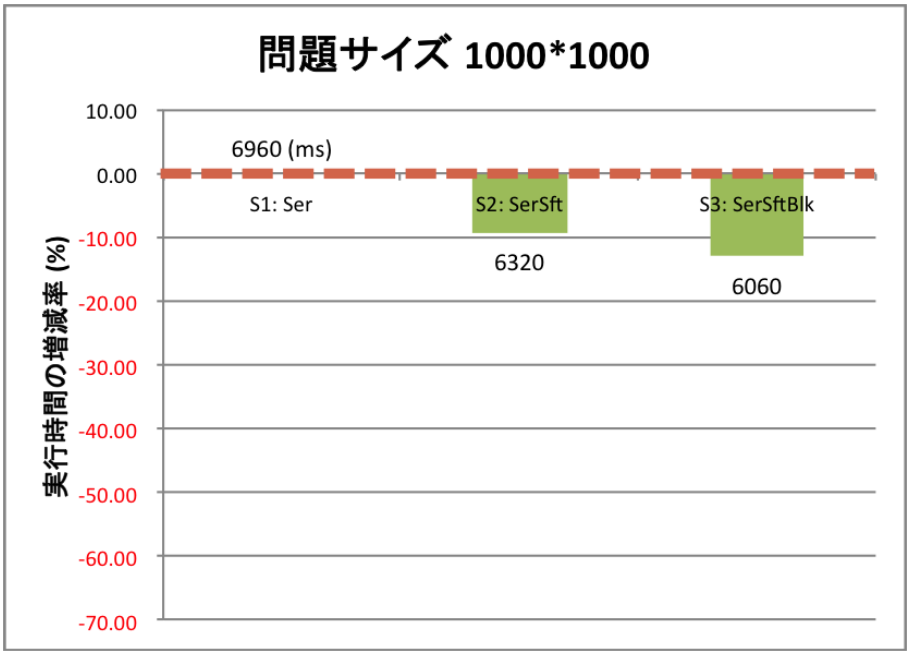

図 7.3. コード S1-S3 を問題サイズ 1000*1000 で実行した場合の実行時間増減率

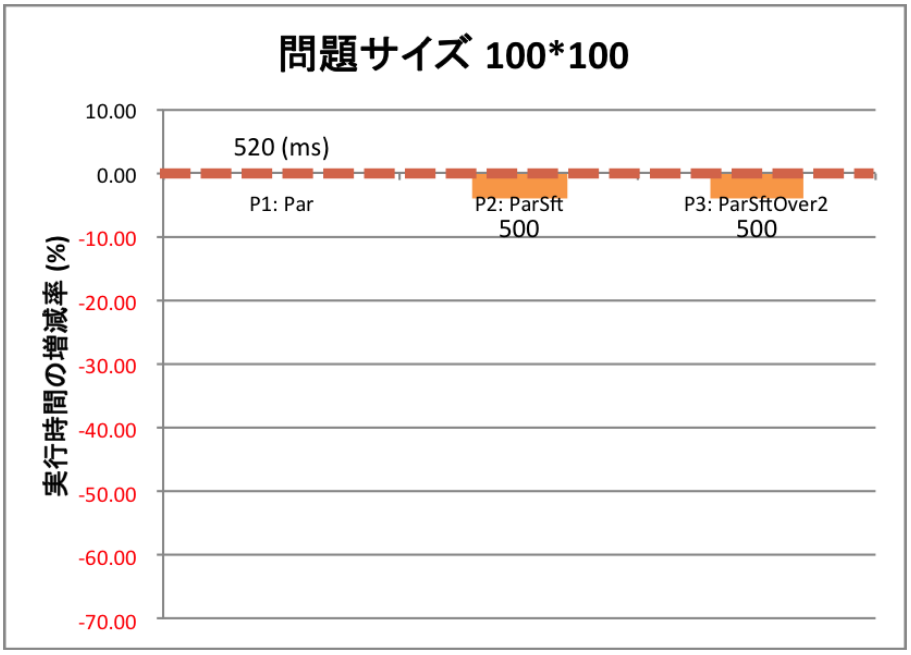

図 7.4. コード P1-P3 を問題サイズ 100*100 で実行した場合の実行時間増減率

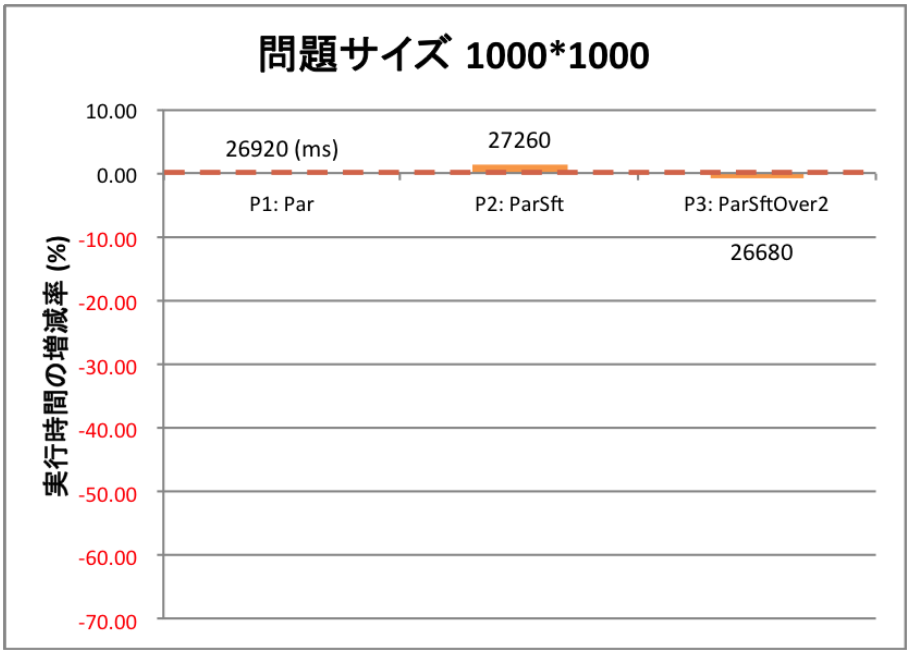

図 7.5. コード P1-P3 を問題サイズ 1000*1000 で実行した場合の実行時間増減率

セルの割合を全体に対して大きくしたことで，シフト計算や通信オーバーラップの効果が実行時間に現れるようになったものと思われる．一方，問題サイズを縦長 (100 × 1000) とした場合は，実行性能の改善は見られない．これは，実装上の仕様から，縦長の問題において，長辺の境界領域上での参照効率が低下する為と考えられる．今回自作したコレクション API では，二次元平面を，縦方向のマス目を表す ArrayList に横方向のマス目を表す ArrayList を入れ子にしたものとして表現している．そのため，領域を横方向に走査する場合にはメモリ上で連続した番地へのアクセスが行えるが，縦方向の走査では，セルにアクセスする度に不連続な番地を参照しなければならない．従って，データ交換時に縦方向の走査が多くなるため，問題サイズを 100 × 1000 とした場合により大きなオーバーヘッドがかかっていると考えた．

以上の結果により，MoCoJ の述語を使って実装した最適化が，期待通りの効果を発揮する場合があることが確認できた．ただし，MoCoJ による最適化効果が，手で実装した最適化効果と比べて同等であるかどうかは本実験の結果からは判らない．例えば，P1，P2，P3 を問題サイズ 1000 × 1000 として実行した場合は最適化の効果が見られないが，その原因は MoCoJ が生成したコードの実装方法，あるいは最適化手法そのものによる効果の限界の二つが考えられるものの，現時点ではどちらであるか考察することができない．このような，MoCoJ の最適化の効果と，手による最適化で得られる効果との比較は今後の課題となっている．

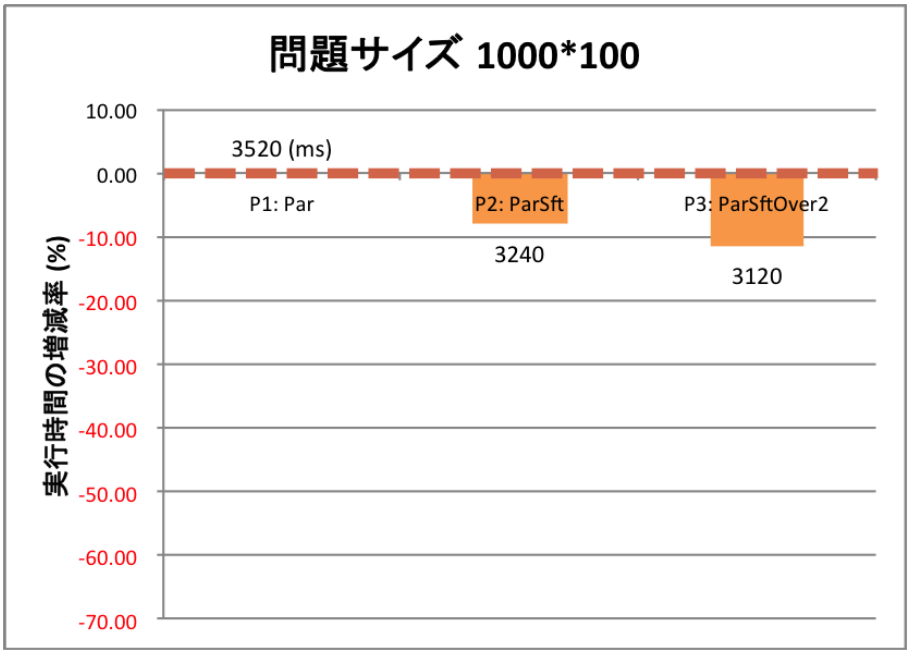

図 7.6. コード P1-P3 を問題サイズ 1000*100 で実行した場合の実行時間増減率

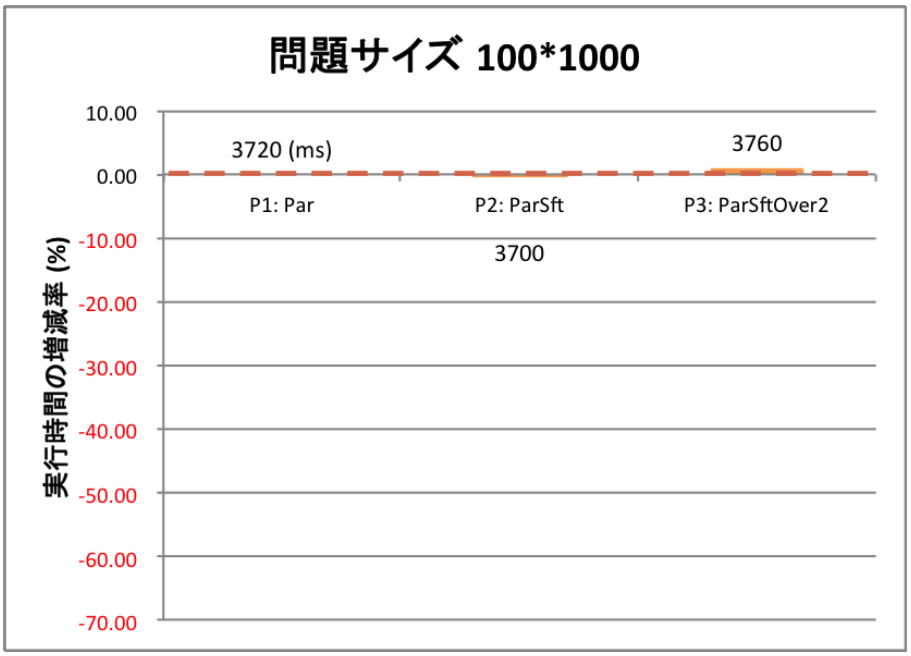

図 7.7. コード P1-P3 を問題サイズ 100*1000 で実行した場合の実行時間増減率

第 8 章

まとめ

本研究では，領域毎の処理の差異を表現可能な部分メソッドディスパッチと，コレクションの走査順序制御を可能にする precedes 修飾子による，並列コレクションのステンシル計算向け拡張，Mosaic Collection for Java (MoCoJ) を提案した．部分メソッドディスパッチは述語ディスパッチ（predicate dispatch）を部分領域指定に特化させて部分領域毎のメソッドオーバーライドを可能とした機構であり，precedes 修飾子は部分領域毎に再定義したメソッドの実行順序を指定するものである．実装は JastAdd を用いた Java 拡張によって行い，コンパイル時に順序制約から実行順序を決定する．実行順序の決定には，Java 向けのグラフライブラリである JGraphT を利用した．また実行順序生成の際には通信オーバーラップ，走査順序のシリアル化により素朴な実装よりも良い性能のコードを生成する．これにより，ユーザは並列コレクションを使ったステンシル計算プログラムの最適化をより柔軟に行える．また，実験では MoCoJ の述語を用いて記述した最適化プログラムの実行効率を調べるため，MoCoJ による最適化コードと素朴なコード間の実行性能の差を，ローカルと FX10 上で調査した．結果としては，MoCoJ の述語による最適化が，期待通りの効果を発揮する場合があることが確認できた．MoCoJ の最適化の効果と，手による最適化で得られる効果との比較は今後の課題となっている．

本研究の今後の方向性としては，言語としての機能に関するもの，実装に関するものの 2 種類が挙げられる．前者については，より良い効果が期待できたり，一般的に使われている最適化手法の更なるリサーチ，導入により，MoCoJ がサポートする最適化手法の幅を広げるという道がある．また，現在の言語仕様にはまだ洗練の余地がある．例えば，MoCoJ では空間・時間ブロッキングをサポートしているものの，そのブロックサイズはプログラムの本文中には記述できず，実行時にパラメータとしてコンパイラに渡さなければいけない．このような MoCoJ の記述力の限界に対して，言語として十分な記述性を有しているかの検討を含め，各種の最適化を簡潔且つ統一的に扱えるような構文仕様を追求することも有用であると考える．実装の面については，現在完成しつつある実装では，部分メソッドやそれらの実行順序の生成部など，MoCoJ の基本的な機能のみ実現するに留まっている．そのため，動的な部分領域や，時間・空間ブロッキングも扱えるようコンパイラを拡張することは一つの目標であるが，これに伴って現在よりもオーバーヘッドが少ないような実装手法を考案することも，本研究を深め

る一つの手段である．更に，コンパイラ実装の際には，時間・空間ブロッキングを施したコードの，ブロック単位での並列化，あるいはブロック内での並列化については考慮していない．このような複雑な並列化手法を反映できるような実装の検証も，MoCoJ の実用性の向上に繋がると考えられる．

発表文献と研究活動

(1) 宗 桜子，佐藤 芳樹，千葉 滋．メソッドのオーバーライドにより部分的実行順序が定義可能な HPC 向け言語，PPL2013，2013.03.04. (ポスター)

(2) 宗 桜子，佐藤 芳樹，千葉 滋．処理の差異と順序を考慮した並列コレクション向け Java 言語拡張．並列/分散/協調処理に関するサマー・ワークショップ (SWoPP2013)，2013.07.31.

参考文献

[1] Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss. Optimized stencil computation using in-place calculation on modern multicore systems. In Henk J. Sips, Dick H. J. Epema, and Hai-Xiang Lin, editors, *Euro-Par*, Vol. 5704 of *Lecture Notes in Computer Science*, pp. 772–784. Springer, 2009.

[2] Thomas Pohl, Markus Kowarschik, Jens Wilke, Klaus Iglberger, and Ulrich Rde. Optimization and profiling of the cache performance of parallel lattice boltzmann codes in 2d and 3d. *PARALLEL PROCESSING LETTERS*, Vol. 13, p. 2003, 2003.

[3] 宗桜子, 武山文信, 千葉滋. メソッド間の依存関係から適切な計算順序を生成する言語. 日本ソフトウェア科学会第 29 回大会 (2012 年度) 講演論文集. 日本情報科学会, 2012.

[4] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, Vol. 43, No. 3, pp. 287–296, March 2008.

[5] Merge framework a programming model for heterogeneous multi-core systems (slide). `http://www.hpcsw.org/presentations/thurs/linderman.pdf`, 2008.

[6] Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan, and Scott B. Baden. Bamboo: Translating mpi applications to a latency-tolerant, data-driven form. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 39:1–39:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 9:1–9:12, New York, NY, USA, 2011. ACM.

[8] 李珍, チャン トゥアンミン, 小田嶋哲哉, 朴泰祐, 佐藤三久. Pgas 並列プログラミング言語 xcalablemp における演算加速装置を持つクラスタ向け拡張仕様の提案と試作, 2012.

[9] Jinpil Lee and Mitsuhisa Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. *2012 41st International Conference on Parallel Processing Workshops*, Vol. 0, pp. 413–420, 2010.

[10] Xcalablemp. http://www.xcalablemp.org/.

[11] Hans Zima, B. L. Chamberlain, and D. Callahan. Parallel programmability and the chapel language. *International Journal on HPC Applications, Special Issue on High Productivity Languages and Models*, Vol. 21, No. 3, pp. 291–312, 2007.

[12] The chapel parallel programming language. http://chapel.cray.com/.

[13] Openjdk: Project lambda. http://openjdk.java.net/projects/lambda/.

[14] Project lambda: To multicore and beyond(slide). http://www.oracle.co.jp/javaone/2012/download-docs/file.php?name=JS1-31_DL.pdf.

[15] Jsr 335: Lambda expressions for the javatm programming language. https://www.jcp.org/en/jsr/detail?id=335.

[16] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pp. 36–43, New York, NY, USA, 2000. ACM.

[17] Java platform standard ed. 8 draft ea-b123. http://download.java.net/jdk8/docs/api/java/util/stream/Stream.html, 2013.

[18] T. J. Team. Jastadd. http://jastadd.org.

[19] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pp. 884–885, New York, NY, USA, 2007. ACM.

[20] Torbjörn Ekman and Görel Hedin. The jastadd system - modular extensible compiler construction. *Science of Computer Programming*, Vol. 69, pp. 14 – 26, 2007.

[21] Barak Naveh. Jgrapht. http://www.jgrapht.org/.

[22] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, Vol. 5, No. 11, pp. 558–562, November 1962.

[23] Fx10 スーパーコンピュータシステム (oakleaf-fx). http://www.cc.u-tokyo.ac.jp/system/fx10/.

[24] Doug Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, Vol. 58, No. 3, pp. 293 – 309, 2005. ¡ce:title¿Special Issue on Concurrency and synchonization in Java programs¡/ce:title¿ ¡xocs:full-name¿Special Issue on Concurrency and synchronization in Java programs¡/xocs:full-name¿.

[25] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. Mpj: Mpi-like message passing for java. *Concurrency: Practice and Experience*, Vol. 12, No. 11, pp. 1019–1038, 2000.

[26] Mark Baker and Bryan Carpenter. Mpj: A proposed java message passing api and environment for high performance computing. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pp. 552–559, London, UK, UK, 2000. Springer-Verlag.

[27] Mark Baker Bryan Carpenter, Aamir Shafi. Mpj express. http://mpj-express.org/.

[28] Open mpi: Open source high performance computing. http://www.open-mpi.org/.

[29] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In Proceedings, 11th European PVM/MPI Users ' Group Meeting, pp. 97–104, 2004.

[30] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of open mpi: Enabling third-party collective algorithms. In *In Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pp. 167–185. Springer, 2004.

[31] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open mpi: A flexible high performance mpi. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM'05, pp. 228–239, Berlin, Heidelberg, 2006. Springer-Verlag.

謝辞

本研究は，多くの方々のご指導，ご助力なしには成り立ちませんでした．千葉滋教授，情報基盤センター特認講師の佐藤芳樹氏には，日頃から研究方針や論文に関して様々な点で熱心にご指導いただきました．特に佐藤芳樹氏には度々の長時間に渡る議論にもお付き合いいただき，その中で言語仕様を初め論文の構成方法，実装，実験手法など多岐に渡って多くのアドバイス，ご意見をいただきました．深く感謝いたします．また，千葉研究室の皆様には，日々共に研究活動を行う中で多くの知識や助言をいただいただけでなく，多くの場面で叱咤激励していただきました．この場を借りて，お礼申し上げます．有り難うございました．