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

宗桜子[†1]　佐藤芳樹[†1]　千葉滋[†1]

並列コレクションを利用したステンシル計算に対して，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為の Java 言語拡張を提案する．Java や Scala に導入された並列コレクションはステンシル計算と相性が良い．ステンシル計算のプログラムは格子空間上のセルを配列として表現し，その各セル上で周囲のセルの値を使って同一計算を反復処理するものである．並列コレクションを利用した反復処理は，低レベルなループ文ではなく，各セルでの計算がリスト内包表記で記述されラムダメソッドのようなコードブロックとして分離されるため処理順序が隠蔽される．そのため，プログラマは実行スケジューリングの最適化を意識せずにステンシル計算を並列分散化することができる．しかし，処理順序の隠蔽された素朴な並列コレクションでは，セル間に依存関係を持ったステンシル計算の実装は難しい．並列分散化されたステンシル計算では全ての配列要素で同じ処理が行われるのではなく，各計算ノードが担当する領域境界点では隣接ノードとの間の袖領域のデータ送受信や同期を実装する必要が出てくる．また，ノード間通信をバックグラウンド化する通信オーバーラップや，通信回数を削減する時間ブロッキング手法のような最適化を施すためには，コレクション要素間の細やかな実行順序制御も必要となってくる．本研究では部分処理と部分順序を定義するための部分メソッドディスパッチ （partial method dispatch）を Java 言語に導入する．部分メソッドディスパッチは，データ範囲や実行順序を条件にメソッドを上書きできる言語機構である．これによりユーザは，袖領域での部分処理や，中心領域など他の部分領域との実行順序を，メソッド上書き条件として明示的に与えられる．これらは，上書きされるデータ範囲を付加する範囲指定構文と，上書きした部分メソッド間での実行順序を定義するための precedes 構文を用いて記述される．また，同名メソッドの上書きによって走査順序を定義するため，既存の並列コレクションをそのまま再利用することができ，かつ別の上書き条件が実装されたクラスへ容易に切り替えられるようになる．

1. 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[5]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し，3 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．4 章では 3 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 修飾子を提案し，続く 5 章では関連研究について述べ，6 章で本論文をまとめる．

†1 東京大学大学院情報理工学系研究科創造情報学科
Dept. of Creative Informatics, The University of Tokyo.

2. 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 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．

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

リスト 1 セル上での計算例

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，隣接ノードとの通信中に，交換データに依存しない計算を先に行う通信オーバラップは全体の実行時間を短縮できる事がよく知られている．

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]．シフト計算では，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．そのため，ステップ毎に全ての隣接ノードと袖領域のデータを送受信する必要が無くなり，片方向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（図 1）．

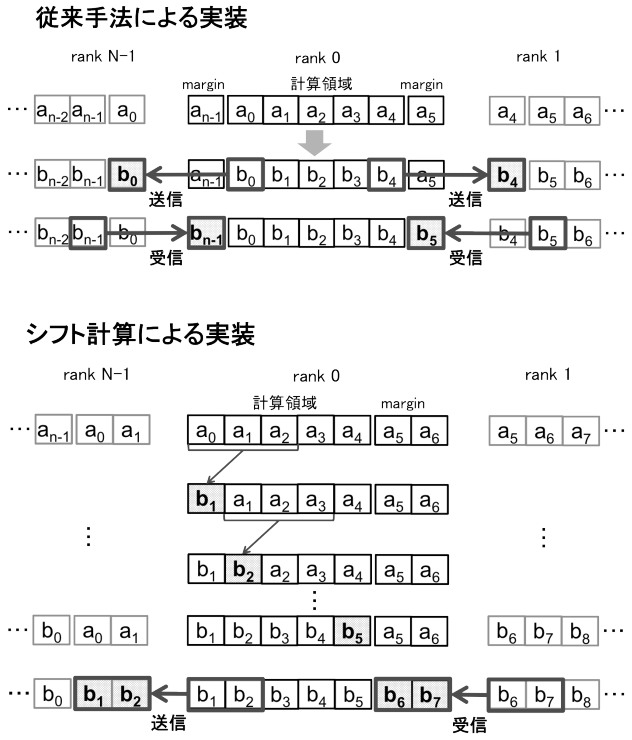

図 1 従来手法とシフト計算

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．

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（リスト 2）．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るよ

うなコード変換で実装される．そのようなコード変換によって，配列の走査順序やブロックサイズはリスト 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，これら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．

```
1:  if (/* 袖領域 */) {
2:    pack(); // 送信用データを用意
3:    exchange(); // データ通信
4:    unpack(); // 受信したデータを配列に格納
5:    calculate(); // 値の更新
6:  } else if (/* 中心領域 */) {
7:    calculate(); // 値の更新のみ行う
8:  }
```

リスト 2 領域毎の処理の違い

```
 1: for( int bz = 0; bz < MaxZ ; bz += bsz )
 2:   for( int by = 0; by < MaxY; by += bsy )
 3:     for( int bx = 0; bx < MaxX; bx += bsx )
 4:       for( int t = 0; t < tmax; t++) {
 5:         tc = tmax - t - 1;
 6:         xMin = ...; xMax = …;
 7:         yMin = …; yMax = …;
 8:         zMin = …; zMax = …;
 9:         for( int z = zMin; z < zMax; z++){
10:           for( int y = yMin; y < yMax; y++){
11:             for( int x = xMin; x < xMax; x++){
12:               /* Calculate on cell(x, y, z) */
13: }}}}}}}
```

リスト 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

3. 並列コレクションを用いたステンシル計算の実装

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．並列コレクションは，ラムダメソッド[5]やメソッドリファレンスとの併用によりリスト処理を簡潔に記述でき，Scala や JavaScript をはじめ，次バージョンの Java でも搭載される機構である．例えばリスト 4 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．

```
1: Grid<Cell> localGrid = new Grid<Cell>(...);
2: do {
3:     double max = localgrid.map(Cell#calculate)
4:                               .max();
5: } while (max < THREASHOLD) // 定常状態で終了
```

リスト 4 並列コレクションを使ったコード例

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難しい．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 5 のようにステンシル計算に条件判定文を追加すれば所望の動作が期待できる．しかし，領域判定の記述が直接埋め込まれ，保守性，可読性が低くなることに加えて， Java 8 の並列コレクションの map メソッドでも複数のコードからなるブロックには対応しない．

```
 1:  double max = grid.map(
 2:    if (/* 中心領域 */) {
 3:      calculate(); // 値の更新のみ行えば良い
 4:    } else if (/* 袖領域 */) {
 5:      pack(); // 送信用データを用意
 6:      exchange(); // データ通信
 7:      unpack(); // 受信したデータを配列に格納
 8:      calculate(); //値を更新
 9:    }
10:  ).max();
```

リスト 5
並列コレクションによる
領域毎に異なる処理の記述

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反面，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再利用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ドする必要がある．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，現実的ではない．

4. 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

4.1 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 言語拡張を提案する．部分メソッドディスパッチを備えた Java 言語では，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 2 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，スーパー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リファレンスを，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 6）．オーバーライドされたメソッド群のうち，呼び出し時の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[a]．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[b]．

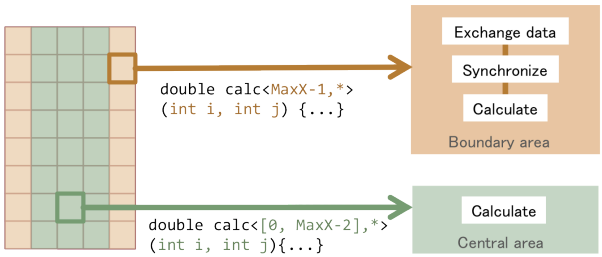

図 2 部分領域毎に異なる処理内容を持つメソッド

```
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 =
13:           (cell[i+1][j] + …… + cell[i][j])/6;
14:   cell[i][j] = next;
15:   return Math.abs(curr-next);
16: }
```

リスト 6 部分領域の記述

4.2 precedes 修飾子の導入

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，

a) 省略形として`<MaxX-1*, *>`または`<MaxX-1, *>`とも記述できる．
b) 省略形として`<MaxX-1?, ?>`，`<MaxX-1, ?>`も利用できる．

リスト 7 のように，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．

```
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: /* 袖領域 */
15: double calculate(int i, int j) <MaxX-1, *> {
16:   return default.calculate(i,j);
17: }
```

リスト 7 前処理の分離

さらに，動的な部分領域を指定し，実行時コンテキストに依存して変化する相対的な領域間での順序を与えることもできる．例えば，先述のようにステンシル計算では，任意のセルの計算は，1 ステップ前の隣接セルの計算を待つ必要がある．一方，時間と空間のブロッキングを考慮した場合，さらにブロッキング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，リスト 8 に示すように，メソッドの仮引数を用いて動的な部分領域を指定し,その順序付けを定義可能とした. 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を表現できる．ただし，ブロッキングサイズに基づいた順序関係の記述は，領域記述が煩雑になるため，現時点では並列コレクションやコンパイラに走査ポリシーを与えられるようなシステムを検討している[c].

```
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:  }
```

リスト 8 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

c) 動的な部分領域は，計算領域全体に対するメソッド（default メソッド）のみに記述できる．

4.3 コンパイラ拡張による実装

部分メソッドディスパッチ機能を備えた Java 言語は，Java のコンパイラ拡張によって実装する．コンパイラ拡張は，Java コンパイラ拡張フレームワークである JastAddJ[2]を利用する．JastAddJ では，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．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 3 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．解析時には，順序関係の制約だけでなく通信オーバーラップも考慮した走査順序を決定すると共に，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．

各要素の実行順序は Java のグラフライブラリである jgraphT を用いて導出する．まず，拡張コンパイラによって，部分領域間に指定された部分順序から，並列コレクションが保持する要素を頂点と有向辺から成る有向グラフへ変換する．次に，得られた有向グラフに jgraphT のトポロジカルソートを適用し，制約を満たす実行順序が作成される．また，コンパイラの実行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．並列コレクションが，2 章で述べた時間ブロッキング，空間ブロッキング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なる．

5. 関連研究

D. Linderman らが提案した Merge フレームワーク[3]は，ヘテロジーニアスなクラスタ環境に向けに predicate dispatch を導入し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 機構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 では，ユーザがコード中に predicate を記述すれば自動的にアーキテクチャとの対応付けが行われ，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せる．しかし，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動的ディスパッチの条件とするが，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されていない．

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．DeVito らの提案する Liszt[4]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

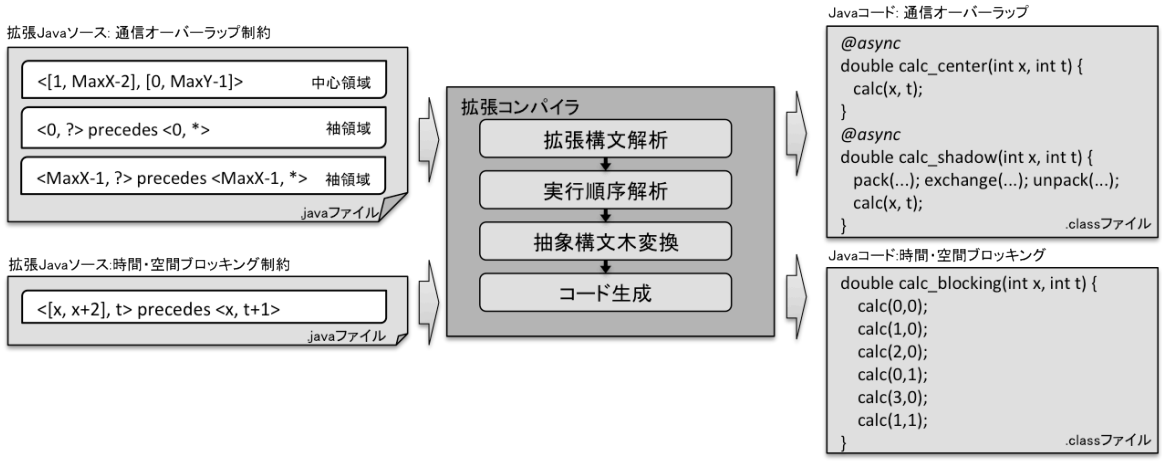

図 3 拡張コンパイラの概要

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．また，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境へ簡単に移植することができる．しかし，Liszt を初めとする多くの 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．

6. 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を制御できる 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関係の制約から実行順序を決定する．その際には通信オーバーラップ，走査順序のシリアル化，メソッド呼び出しのインライン展開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．

今後の課題は，言語仕様の検証と実装，実験である．メソッドの上書き拡張が可能な機構は柔軟な実装を可能とする反面，セキュリティホールやデッドロックを誘発する可能性がある．そのため，実装の前に改めて言語仕様を見直し，セキュリティ面の脆弱性が無いか確認，修正する必要があると考えている．実行順序の解析自体はコンパイル時に行うため実行性能に影響を与えないものの，同じステンシル計算を実装した場合，並列コレクションや述語ディスパッチの利用により，素朴な実装よりも実行時オーバーヘッドが増加することも考えられる．従って実装後には，具体的なステンシル計算を題材として実行性能の計測を行い，オーバーヘッドが現れた場合にはその原因を考察することで提案言語の実用性を高めていきたい．

謝辞 本研究の推進には，東京大学情報基盤センターの富士通 PRIMEHPC FX10 System (Oakleaf-FX) を利用した．

参考文献

1) W. Augustin, V. Heuveline, and J. Weiss.: Optimized stencil computation using in-place calculation on modern multicore systems, In Euro-Par 2009, pp. 772–784, 2009.
2) T. Ekman and G. Hedin.: The JastAdd system - modular extensible compiler construction, Science of Computer Programming, Vol. 69, pp. 14-26, 2007.
3) M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng.: Merge: A programming model for heterogeneous multi-core systems, ASPLOS 2008, pp. 287–296, 2008.
4) Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan.: Liszt: A domain specific language for building portable mesh-based PDE solvers, In SC'11, pp. 1–12, 2011.
5) Project Lambda, http://openjdk.java.net/projects/lambda/