プロセッサ間データ依存関係を緩和する並列ループ構文 for_slice の提案

平成8年2月15日
前山 浩二

高性能な分散メモリ型並列計算機の出現により、大規模科学技術計算を並列計算機上で実行する数多くの試みがなされている。その中の一分野として数値流体力学があり、その流体の式は偏微分方程式で構成される。離散化により実空間は格子空間に変換され、同時に偏微分方程式は差分方程式に変換される。定常状態の解を求める場合、差分方程式は疎係数行列を持つ連立一次方程式になり、主に反復法を用いる。

反復法のうちSOR法のような緩和法は計算は収束のために逐次改良を行っている。for文により運搬されるフロー依存が存在するため、そのまま並列化した時、一回の反復は逐次的にしか実行できない。そこで、並列化効率を得るため、次の反復処理と重ねてパイプライン的に処理を進める必要がある。しかし、通常反復法は収束判定が必要であり、この処理は普通に逐次言語で記述すると同期的な処理と見なされるため、並列化コンパイラでは高速化しにくい。よって、並列化効果を得るために、プログラマ自らの手によってパイプライン処理を阻害しないような収束判定の処理を実装せねばならない。

そこで、プロセッサ間で交換されるデータとして同一の改良での更新後の値を用いるのではなくどのような場合でも更新前の値を使用する方法が考え出された。この方法では各プロセッサが同一時刻に動作するため、収束判定が簡単に行え、起動時や終了時のパフォーマンスの低下がないため、高速に実行できる。この方法はプロセッサ内の依存はそのままに、プロセッサ間の依存を除去することと一致する。

一方、この並列化手法だと収束効率の悪化が予想されるが、反復法は途中の計算結果を問わないから収束さえすればよい。そして、計算速度の向上が収束効率の悪化分を上回れば、利用価値はあるといえる。しかし、欠点として逐次言語と並列化コンパイラを用いて生成することができないことがあげられる。

そこで、本研究ではプロセッサ間データ依存関係を緩和する並列ループ構文`for_slice`を提案する。`for_slice`はデータ分割に従い、依存をプロセッサ内のもの、プロセッサ間のものと分ける。プロセッサ間の依存はデータの分割次元の添字式と`for_slice`のループ変数が関係している場合除去され、ループ以前のデータを使用するように変更される。よって。各プロセッサは同一時刻に計算を行えるようになり、収束判定は並列化コンパイラを用いても速度が落ちないようになる。また、構文は`for`文に類似しており、`for`文を`for_slice`に変更するだけなので、プログラマの負担は軽減する。

また、`for_slice`を我々が開発している並列化コンパイラTINPARに実装した。TINPARの内部処理は並列化部、最適化部などから構成されている。そこで、`for_slice`の処理を並列化部で行うことにした。理由は、1) 通信の移動場所が明らかである 2) その後の最適化部を十分利用できる、である。また、`for_slice`のために、隣接する非所有データの一部を格納するオーバーラップエリアを実装した。

`for_slice`は収束に関する考察をした結果、SOR法で使用する大きな緩和係数を用いると収束しないなることがわかった。しかし、主な用途として緩和係数が1より小さい不足緩和を用いるナビエストークス方程式を考えており、その実験結果では収束効率の悪化はそれほど見られなかった。また、全体の計算時間も理想的なfor文の並列化プログラムに対して10%ほど向上した。

The Proposal of parallel loop construction “for_slice” relaxing data dependence relations between processors

February 15, 1996
Koji MAEYAMA

The large-scale science and technology calculations are to be executed on a memory distributed parallel computer. One of fields of large scale computing is computational fluid dynamics, and expressions of the fluid are constructed with partial differential equations. By discretisation, the real space becomes the lattice space and partial differential equations become difference equations. When the solution of steady states is solved, difference equations become coalition linear equations and we use iterative method chiefly for the cost.

Iterative methods like SOR method execute sequential improvement for the convergence and the parallelization of `for` construction orders the execution time of the processors. The processing can be advanced in the pipeline for the parallelized effect. However, because the convergence judgment is usually necessary and the judgment processing is a synchronous processing, the parallelized effect is not achieved easily. In order to achieve the effect of parallelization, it is necessary to implement a convergence judgment so that the pipeline processing is not blocked, that gives the programmer a big labors.

Then, the method of using data before renewal of data which is exchanged between processors, is proposed. It is possible to execute calculations in high speed because convergence judgment can be easily done because each processor works in this method at the same time and there is no decrease in the performance at the start and the end. The method is consistent with the method of the elimination of only data dependence relations between processors.

Moreover, though the deterioration of the convergent efficiency is forecast, iterative method only has to just converge. And, it is useful if the calculation speed of the improvement of the solution overcomes the deterioration of the convergent efficiency. However, it is not able to be generated by using the sequential language and the parallelizing compiler.

In this paper, we propose parallel loop construction “`for_slice`” relaxing the data dependence relations between processors. The condition of the dependence relations relaxed by `for_slice` specifies the loop variable. The solution by `for_slice` calculates each processor at the same time. Therefore, the convergent judgment can be described by the sequential language and the parallelizing compiler. Moreover, the `for_slice` construction resembles `for` and reduces programmer’s load.

And `for_slice` was implemented on parallelizing compiler TINPAR which we are developing. TINPAR is constructed with the parallelizing part, the optimizing part, and etc. The `for_slice` processing is executed on the parallelizing part, because the place on which communication codes are arranged is obvious before optimization, and it effectively uses the optimizing part after parallelization. And for `for_slice` the overlap area is implemented on TINPAR , this area is the place on which a part of neighboring non owner data are put.

The convergence efficiency deteriorates because the computation sequence is changed and is executed in parallel for `for_slice`. If the over relaxation coefficient used by a usual SOR method is used for `for_slice` solution, this may not converge. However, the main purposes of `for_slice` is to solve the simultaneous partial differntial equations like the Navier-Stokes equation which used the under relaxation coefficient which was more small than one. The deterioration of the convergence efficiency was not seen too much in the experiment result, and the speed of the entire computing time improves about 10% against `for` with high-degree convergence judgment which parallelizing complier cannot create.

1　はじめに

並列計算機が期待されている利用分野として数値流体力学がある。この分野では偏微分方程式を離散化することにより、空間を格子化している。楕円型偏微分方程式は離散化により巨大な疎係数行列の連立一次方程式を解くこととなる。これをガウス法などの直接法で解くのはコストがかかる、あるいは連立偏微分方程式の場合には直接法で解くことが不可能あるため、多くの場合反復法のアルゴリズムを用いて解を求めている。

反復法の逐次アルゴリズムでは解の収束効率向上のため、解の改良のための一回のイタレーションにおいて、格子点の解を順番に改良している。その結果、隣接する格子点に対するループ運搬依存が生じるようになっている。これを複数のプロセッサにデータを分割して並列化すると、データ依存のためのプロセッサ間の通信が生じる。したがって、一回の改良では各プロセッサが逐次的に動作するため、収束判定を工夫する必要があるなど、高速化のためには非常に多くの労力を要する。

そこで、並列計算機を用いるに当たって、プロセッサ間の通信タイミングをずらし、各プロセッサが解の改良のイタレーションを同時実行する方法が考え出されている[8]。この方法では、並列性を高めるためにプロセッサ間にまたがる依存関係を除去するが収束効率の悪化を防ぐためにプロセッサ内での依存関係は保存する。

この方法による利点は、プロセッサ内の計算により通信の待ちが隠蔽され、一回のイタレーションが高速になる。また、収束判定の処理が単純で済むため、並列実行による高速化が達成しやすい。

一方、欠点として計算順序の変更により収束効率が悪化するという欠点がある。しかし、収束効率の悪化よりも一回のイタレーションの実行時間が短縮が上回れば、全体の実行時間は短縮できる。

別のより重要な欠点は、逐次プログラムを大幅に書換え、手作業で並列化を行わねばならないことである。すなわち、逐次プログラムを並列化コンパイラで、自動的に並列化することはできない。したがって、プログラマの負担が大きいだけでなく、並列化コンパイラが備える高度な高速化手法も利用することができない。

そこで、我々は、プロセッサ間の依存が緩和される並列ループ構文`for_slice`を提案する。`for_slice`はプロセッサ内外のループ依存関係を区別して取扱うことを指示するもので、並列化コンパイラ TINPAR[3] によって自動的に適切なコードに変換される。したがって、逐次プログラムの for ループを`for_slice`ループに置き換えるだけで、反復法を効率よく並列化できる。

本稿では、まず 2 章ではループの並列化および並列ループについてついて述べた後、3 章では`for_slice`の提案とその主たる目的である反復法について、4 章では`for_slice`の仕様と TINPAR への実装について、5 章では`for_slice`を用いた時の収束効率の変化について、それぞれ述べる。続いて 6 章では、反復法を用いた`for_slice`の有効性について述べる。7 章で`for_slice`の拡張について述べた後、8 章でまとめる。

2　ループの並列化

2.1　逐次ループの並列化

分散メモリ型並列計算機を用いたデータ並列プログラミングでは、配列型のデータを各プロセッサに分散して配置するのが一般的である。このような場合、各々のプロセッサに固有のデータのみで計算が行われるならば、できるなら、完全に並列な動作が可能となる。しかし、他のプロセッサが所有しているデータが必要であると、メッセージ通信により他のプロセッサから送られたデータを受信しなければ処理を進めることができない。

ここでは、配列データを扱う基本的な構文である`for`を、分散型並列計算機で扱う時にどのような流れとなるかを述べる。本論文では断りのない限り、説明に使われるプログラムは TINPAR が対象とするプログラミング言語 tiny language[1] で記述される。

ここで、次のような一般的な構造で説明を行う。

```
for i = 1, n do
        $S_1$
        $S_2$
endfor
```

ここで、イタレーション`i` 内部の文 S_m に対する表現を S_m^i で表す。逐次計算機ではどのような依存があろうとも図 1 のような順番で実行される。しかしこれを並列化すると、各プロセッサの処理の開始時刻や終了時刻は、データの依存関係によって決まってくる。

$$S_1^1 \longrightarrow S_2^1 \longrightarrow S_1^2 \longrightarrow S_2^2 \longrightarrow$$
$$S_1^3 \longrightarrow S_2^3 \longrightarrow S_1^4 \longrightarrow S_2^4 \longrightarrow \ldots$$

図 1: 逐次計算機における実行順序

まず、次の場合を考える。

$$S_1 : a(i) = b(i) - 1$$
$$S_2 : b(i) = c(i) + 1$$

それぞれの S_1、S_2 はループの繰り返しをまたがるような依存 (ループ運搬依存) がない。そこで例えば、配列 a,b をブロック分割すると、各プロセッサは割当てられたブロックの大きさ分だけ繰り返すようなループを、並列に実行することができる。一方、同じイタレーション内では S_1 と S_2 の間に逆依存が存在するので順序を変えることができない。したがって、ブロックサイズが 1 である場合、処理の流れは図 2に示すものとなる。

次にループ運搬のフロー依存がある場合を考える。

$$S_1 : a(i) = b(i-1) - 1$$
$$S_2 : b(i) = c(i) + 1$$

このようにループ運搬依存がある場合、依存の伝播がプロセッサ間でのメッセージ通信に変換される。そのため、図 3に示すように、各プロセッサ処理が逐次化され、最悪の場合には全体の実行時間が逐次計算時間+通信時間となる。

次に逆依存の場合を考える。

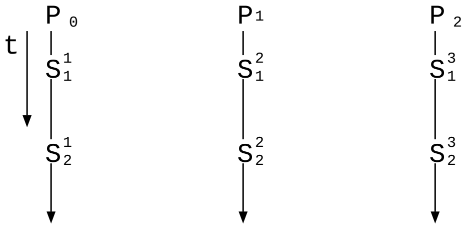


図 2: ループ運搬依存がない場合

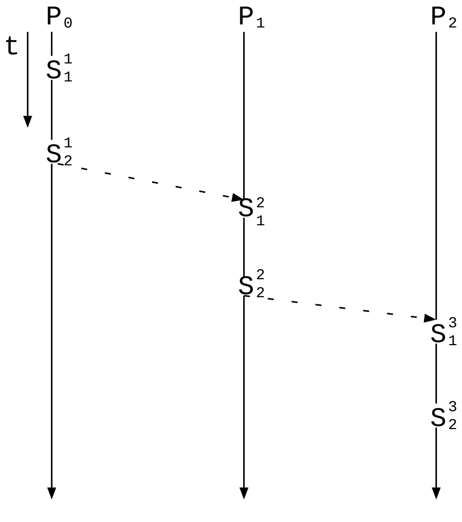


図 3: ループ運搬のフロー依存がある場合

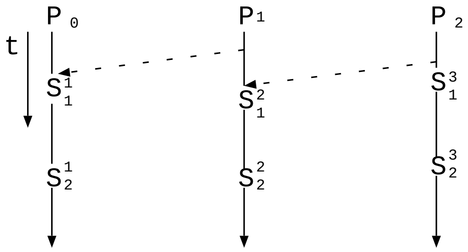


図 4: ループ運搬の逆依存がある場合

$$S_1 : a(i) = b(i+1) - 1$$

$$S_2 : b(i) = c(i) + 1$$

このような場合、図 4で示すように、先に通信を必要とするが計算部分は完全に並列化できる。

次にループ運搬依存があるループの外側にさらにループがある場合を考える。

```
for k = 1, m do
  for i = 1, n do
      $S_1$
      $S_2$
  endfor
   $S_3$
endfor
```

ここで、S_1, S_2, S_3をそれぞれ次のようにする。

$$S_1 : a(i) = b(i-1) - 1$$

$$S_2 : b(i) = c(i) + 1$$

$$S_3 :$$

このとき、流れは図 5のようになり、パイプライン的な並列処理が行われる。なわちループ運搬依存がある場合でもうまく並列化を行えば、外側のループでパイプライン効果を期待できることがある。しかし、S_3が空でなく、例えば、総和演算のようなプロセッサ間の同期を必要とするような処理があった場合、図 6に示すように、パイプライン効果はそこで途切れて並列化の効果は現れないことになる。

そのため、フロー依存があるループが連続して実行される場合、並列化効果を出すために同期処理を間引くか、うまくパイプラインに適合するような処理に変形することが必要となる。

2.2 並列ループ構文

通常のループ構文とは違い、ベクトル/並列処理を前提としたループ構文が存在している。代表的なものとして HPF(High Performance Fortran) における`FORALL` 構文がある [2]。

`FORALL` は`for` 文と同じ構造をしているがループ運搬依存が存在すると意味が全く異なる。例えば、tiny 風に記述した以下の`forall` ループを考える。

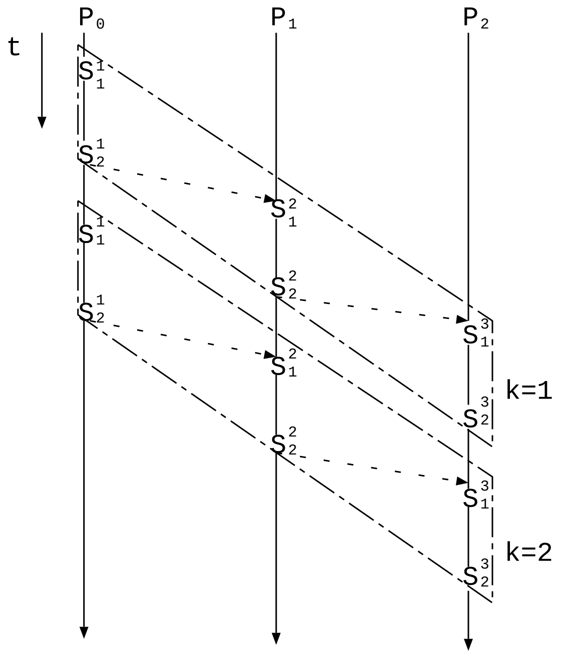


図 5: パイプライン処理の流れ

```
forall i = 1, n do
    $S_1$
    $S_2$
endfor
```

上記のループがforであるならば、ループ運搬フロー依存があれば、前のイタレーションで更新された値を使用し、逆依存があれば、後のイタレーションの更新される前の値を使用する。そのため、計算結果を保証するためにイタレーションの実行順序を変更してはならないことになる。

一方forallでは、$S_2^{i-1} \rightarrow S_1^i$や$S_1^{i-1} \rightarrow S_1^i$のようなループ運搬フロー依存が見かけ上依存しても、S_1^iの計算にはS_2^{i-1}やS_1^{i-1}が更新する前の値を使用する。ただし、$S_1^{i-1} \rightarrow S_2^i$や$S_1^i \rightarrow S_2^i$のように、文の前後関係とフロー依存関係が一致する場合には、S_2^iの計算にはS_1^{i-1}やS_1^iが更新した値を使用する。すなわち上記のforallは以下のものと等価である。

```
forall i = 1, n do
    $S_1$
endfor

forall i = 1, n do
    $S_2$
endfor
```

forallをブロックサイズ 1 で並列計算機で実行すると図 7の流れになる。また、ブロックサイズ 2 以上の場合、見かけ上のフロー依存を除去するために、配列の旧

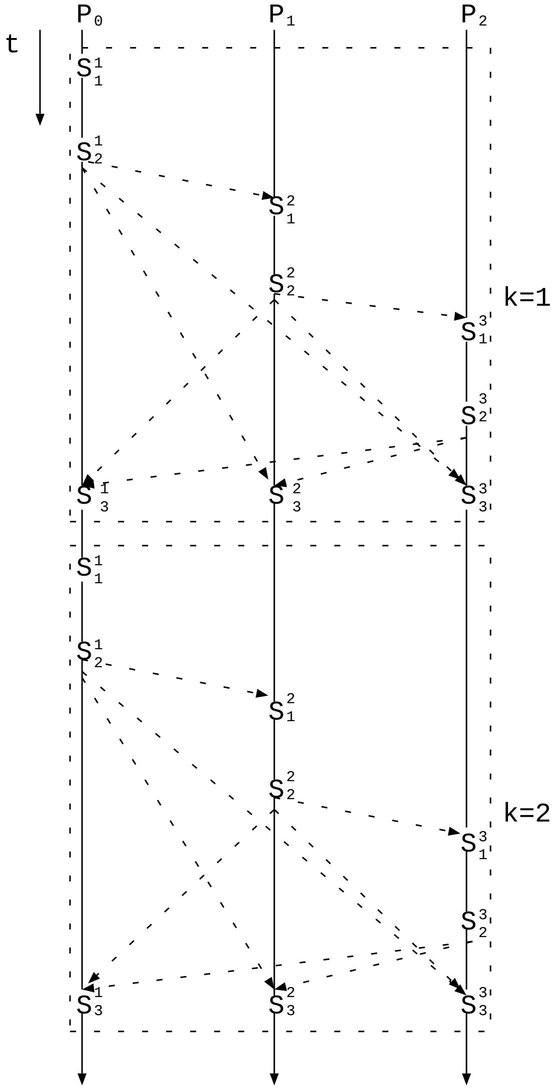


図 6: パイプライン処理が不可能な場合の流れ

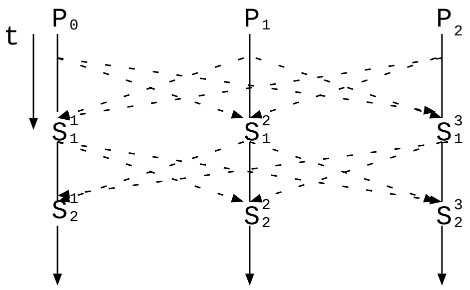


図 7: forall の流れ

値をループの実行開始前に保存することがしばしば必要である。

また、HPF では`INDEPENDENT` ディレクティブを仕様に盛り込んでいる。`INDEPEN-DENT` ディレクティブは`DO` や`FORALL` に用い、各イタレーションが他のイタレーションに対して独立であることを宣言する。従って、図 8に示すようにループの実行開始後は、プロセッサ間でのフロー依存は存在しない。なお、`INDEPENDENT` ディレクティブは、コンパイラが解析困難な依存関係を示すために用いられ、ループの意味を変えるものではない。したがって、実際にループ運搬依存が存在する場合には、`INDEPENDENT` ディレクティブを用いて依存関係を除去することはできない。

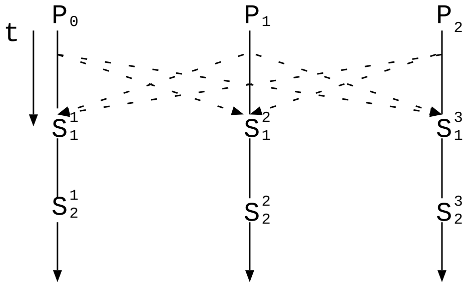


図 8: INDEPENDENT ディレクティブを用いたループの流れ

3 反復法とその並列化

本研究で提案する並列ループ構文`for_slice` は、2 章で述べた`for` と`forall` の中間に位置し、プロセッサ内外のループ運搬依存を区別して取扱うことができる。本章では、`for_slice` の意義を示すために、その主な適用例である連立一次方程式の反復解法の並列化について述べる。

3.1 反復法

`for_slice` の主な適用例の一つに、楕円型の偏微分方程式の求解がある。数値流体力学などの分野では、空間を離散化し、各格子点上の物理量に関する (連立) 差分方程式を解くことにより、偏微分方程式の近似解を求める。この差分方程式の求解は、疎係数行列を持つ巨大な連立一次方程式を解くことと一致する。この連立方程式を直接法で解くのは一般に困難であるため、多くの場合には反復法が用いられる。

まず、楕円型偏微分方程式の例として、二次元のラプラス方程式、

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad (1)$$

を用いる。

これを計算機で解く場合、離散化を行って近似値を求める。ここで、j, kを格子の xy座標とする時、二次精度の中心差分を用いると、ラプラス方程式は、

$$\frac{u_{j+1,k} - 2u_{j,k} + u_{j-1,k}}{(\mathit{\Delta} x)^2} + \frac{u_{j,k+1} - 2u_{j,k} + u_{j,k-1}}{(\mathit{\Delta} y)^2} = 0 \qquad (2)$$

のように表され、

$$u_{j,k} = \frac{1}{4}(u_{j-1,k} + u_{j+1,k} + u_{j,k-1} + u_{j,k+1}) \qquad (3)$$

となる。ここで格子点を辞書順に並べると、

$$\begin{pmatrix} -4 & 1 & & 1 & & & & & \\ 1 & \ddots & \ddots & & \ddots & & & & \\ & \ddots & \ddots & \ddots & & \ddots & & & \\ 1 & & \ddots & \ddots & \ddots & & \ddots & & \\ & \ddots & & \ddots & \ddots & \ddots & & \ddots & \\ & & \ddots & & \ddots & \ddots & \ddots & & 1 \\ & & & \ddots & & \ddots & \ddots & \ddots & \\ & & & & \ddots & & \ddots & \ddots & 1 \\ & & & & & 1 & & 1 & -4 \end{pmatrix} \begin{pmatrix} \vdots \\ u_{j-1,k} \\ \vdots \\ u_{j,k-1} \\ u_{j,k} \\ u_{j,k+1} \\ \vdots \\ u_{j+1,k} \\ \vdots \end{pmatrix} = \mathbf{0} \qquad (4)$$

となり、係数行列が疎の連立一次方程式を解くことと同じであることが分かる。

格子点が少ない場合、ガウス消去法などの直接法で解かれる場合もあるが、多く場合巨大な疎行列係数の連立一次方程式なるため、反復法を用いて解かれることとなる。

まず並列化手法を説明する前に基本となる緩和アルゴリズムを取り上げる。

ヤコビ法 式 (3) の左辺を更新後の値、右辺を更新前の値と見なすと次のような式になる。

$$u_{j,k}^{n+1} = \frac{1}{4}(u_{j-1,k}^n + u_{j+1,k}^n + u_{j,k-1}^n + u_{j,k+1}^n) \qquad (5)$$

ここで、n は解の改良の反復回数である。また、これに緩和係数を用いるとことができ、緩和係数を用いると次のようになる。

$$\begin{aligned} u_{j,k}^{n+1} &= (1-\omega)u_{j,k}^n \\ &\quad + \tfrac{1}{4}\omega(u_{j-1,k}^n + u_{j+1,k}^n + u_{j,k-1}^n + u_{j,k+1}^n) \\ &= u_{j,k}^n + \tfrac{1}{4}\omega(u_{j-1,k}^n + u_{j+1,k}^n \\ &\qquad -4u_{j,k}^n + u_{j,k-1}^n + u_{j,k+1}^n) \end{aligned} \qquad (6)$$

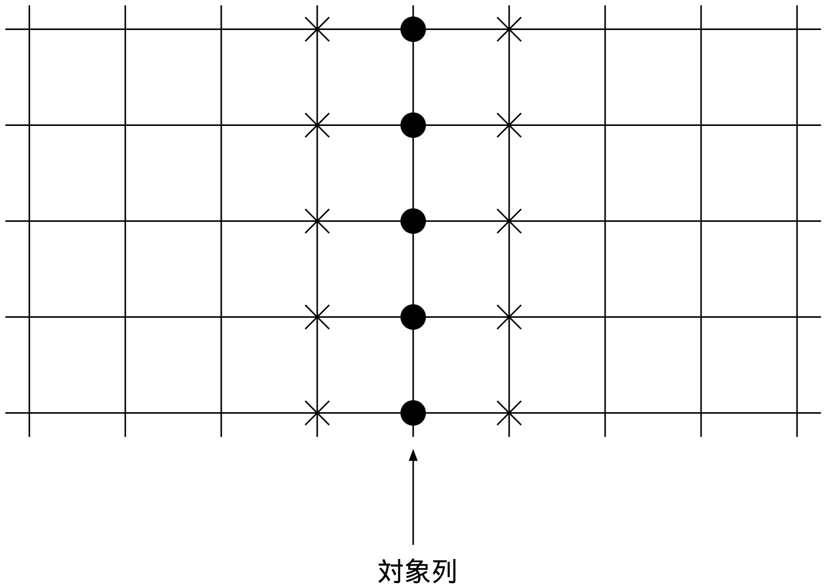


図 9: 線アルゴリズム

このアルゴリズムの特徴として、ループ依存がないため`forall`を用いて簡単に並列化できること、その反面収束効率が非常に悪いことが挙げられる。

ガウスザイデル法 反復改良によって更新された値は、前回の値より正しい値に近付いていると考えられるので、収束効率を改善するためにヤコビ法を

$$u^{n+1}_{j,k} = \frac{1}{4}(u^{n+1}_{j-1,k} + u^{n}_{j+1,k} + u^{n+1}_{j,k-1} + u^{n}_{j,k+1}) \quad (7)$$

のように変更する。このアルゴリズムはヤコビ法に比べ、約二倍の収束速度を持つが実用的な速さとはいえない。また、$u^{n+1}_{j-1,k} \rightarrow u^{n+1}_{j,k}$,$u^{n+1}_{j,k-1} \rightarrow u^{n+1}_{j,k}$ のループ運搬依存が生じるため簡単に並列化することはできない。

SOR 法 ガウスザイデル法にも、ヤコビ法と同様の緩和係数を用いることができる。すなわち、式 (7) に緩和係数を加えて、

$$\begin{aligned} u^{n+1}_{j,k} &= u^{n}_{j,k} & +\tfrac{1}{4}\omega(u^{n+1}_{j-1,k} + u^{n}_{j+1,k} \\ & & -4u^{n}_{j,k} + u^{n+1}_{j,k-1} + u^{n}_{j,k+1}) \end{aligned} \quad (8)$$

とする。この方法は逐次過緩和法 (SOR:Succesive Over Relaxion) と呼ばれる。緩和係数は $0 < \omega < 2$ が用いられる。$\omega < 1$ の場合を不足緩和と呼び、$1 < \omega$の場合を過緩和緩和と呼ぶ。

三重対角行列解法を用いた反復法 SOR 法などでは、一点一点解を改良していくため、点アルゴリズムと呼ばれるが実際には効率が悪い。そこで、収束効率をよくするために、三重対角の連立一次方程式を用いる線アルゴリズムという方法がある。

ここで三重対角の連立一次方程式は一般に次の式で表される。

$$a_jT_j = b_jT_{j+1} + c_jT_{i-1} + d_j \quad (9)$$

この方程式を直接解くのが三重対角行列解法 (TDMA:Tri-Diagonal Matrix Algorithm) である。

二次元問題以上ではこの解法では直接解くことが不可能なため格子を一次元の線領域に分割し、図 9に示す計算を行う対象列を計算する際にその列に隣接する列を一時的に定数として扱う。そのことはつまり、式 (3) を例として考えると、ある $k = K$に対して、

$$\begin{gathered} a_j = 1, b_j = c_j = \tfrac{1}{4}, d_j = \tfrac{1}{4}(u_{j,K-1} + u_{j,K+1}), \\ T_j = u_{j,K}, j = 1, \ldots, N \end{gathered} \quad (10)$$

であることを示している。

この方法を用いて、二次元以上の空間の解の改良を、ガウスザイデル法と組み合わせたり、SOR 法と組み合わせて行っている。また、SOR 法と組み合わせたものを SLOR 法 (Succesive Line Over Relaxion) 呼ぶ。一般的にこの種のアルゴリズムでは安定した解を得るため、緩和係数を不足緩和 ($\omega < 1$) にする。

3.2 反復法の並列化手法

ここでは、基本的なアルゴリズムである SOR 法を例として、並列化の手法について概説する。SOR 法では収束効率を増すために逐次計算を行うため、ループ運搬依存が生じる。したがって、並列化による効果を出すには以下に示すような工夫が必要である。

3.2.1 パイプライン方式

まず、SOR 法の反復改良の一回のイタレーションを並列化した時のプロセッサの処理の流れは図 10のようになる。図から明らかのように、一回のイタレーションのみだと並列化による効果が全く見られない。また、通常の SOR 法の場合、イタレーション毎に収束判定を行うと、

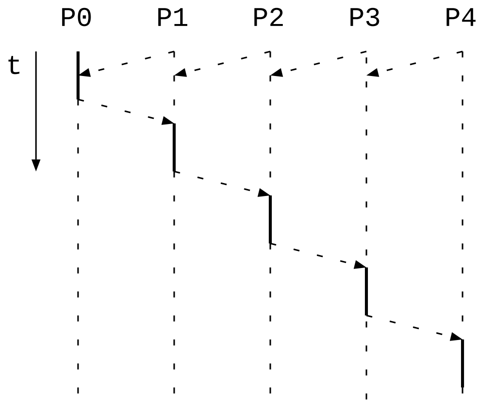


図 10: SOR 法の一回のイタレーションの流れ

2.1 節で述べたように全く並列化による効果が現れないことになる。

そこで、収束判定に工夫をして、図 11のように異なるイタレーションを並列に実行することが考えられる。このようにパイプライン的に処理を行うことにより、処理の最初の部分と最後の部分を除いては、かなりの並列化による効果が期待できる。この方法による利点は、逐次プログラムと同じデータの流れを保証しているので、緩和係数が同じであることが保証できる。

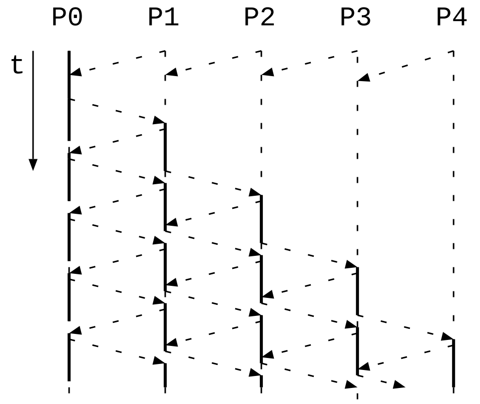


図 11: SOR 法のパイプライン並列処理の流れ

ここで、このパイプライン効果を得るために行う収束判定に関して述べる。その方法として、次の二つの方法が考えられる。

間引き方式
: 文字通り収束判定を間引くものであり、一定回数の反復に一回収束判定を行う。この方法は、真の解との誤差が反復法は単調に減少し、余分に反復しても解は改良されるだけであるという考えに基づいている。一定回数の反復の間パイプライン効果が持続するために、ある程度の並列化効率は期待できる。

  この方法の利点は、逐次言語によるプログラムをそのまま並列化コンパイラより並列プログラムに変換できることである。欠点として、パイプライン効果を得ようとすると余分に実行する回数が多くなり、余分に実行する回数を減らそうとすると収束判定前後の低稼働率の影響が大きくなることがあげられる。

終了連絡方式
: 各プロセッサが収束判定のために同期を取らなければ、理想的なパイプライン効果が得られる。そこで、収束判定をイタレーションの中で一番最後に計算を行うプロセッサにまかせ、他のプロセッサはそこからの終了の連絡を受け停止する方法が考えられる。

  具体的には次の方法で示される。

  1. P_0は所有領域の残差の総和を P_1へ送信する。
  2. P_1から P_{N-2}はそれ以前のプロセッサが所有する領域の残差の総和を受信後、所有領域の残差の総和と受け取ったデータの和をとり、次のプロセッサに送信する。
  3. P_{N-1}は非所有領域の残差の総和を受信後、所有領域の残差の総和と受け取ったデータとの和をとり、収束判定を実行する。収束したら P_0に収束したことを連絡するが、収束していなかったら P_0に対して何も連絡しない。
  4. P_0は P_{N-1}からの収束の連絡が来たら他のプロセッサに収束したことを知らせ、処理を中断する。一定時間受信待ちして P_{N-1}からの連絡が来ないなら収束していないとしてそのことを他のプロセッサに知らせる。
  5. 各プロセッサは P_0からの連絡に従い、収束していなかったら通常通り処理を行う。収束したなら、各プロセッサは連絡が P_0から来ているため P_0と同じ反復回数で停止する。

  この方法では、収束判定のための計算コストは微量でほぼ理想的な実行時間を得られる。ただし、実際に収束してからそれを P_0がしるまでに時間を要するため、逐次的に実行する場合に比べて余分な繰り返しが行われる。

  また、より重大な欠点として以下のものがあげられる。

  - 逐次プログラムで記述された収束判定を、自動的にこの方式に変換することは極めて困難である。
  - 計算の流れが単純なものに限られる。例えば、SOR を一回改良した後次に逆順にたどって計算する SSOR 法 (Symmetric SOR) や反復のたびに交互に対象列が変わる線アルゴリズムである ADI 法のように計算の流れが常に変化するものの場合、この方式は使えない。
  - 計算の終了時刻が様々な要因により非決定的になる。

  そのため、この方式は計算が単純であること、並列化コンパイラを使わず手動で並列化を行うことが前提となる。

3.2.2 プロセッサ間の依存を緩和した並列化技法

これまでに述べてきた並列化手法では、収束効率は変化しないが一般的なものに適応しにくい欠点があった。

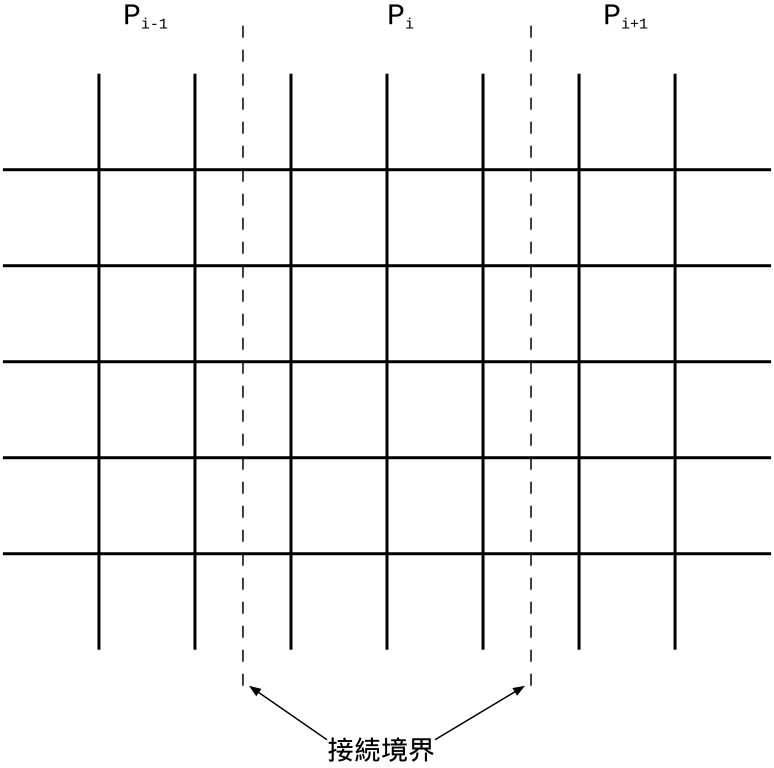


図 12: for_slice で扱われる領域分割法

そこで、プロセッサ内は収束効率向上のため逐次改良を行い、プロセッサ間改良前の値を使用する方法を考える。プロセッサ間に序列がないため、収束判定は単純なものとすることができる。この方法を仮に部分逐次緩和法とする。

部分逐次緩和法は、数値流体力学で用いられる領域分割法の一つの方法と見なすことができる。数値流体力学では領域分割法を物体周りの空間を一つ格子で構成せず、いくつかの部分領域に分割する。計算は部分領域内である程度独立して行う。隣接する領域に対する物体の一貫性を保つために問題によって必要とされるあるタイミングでデータ交換を行う。それを繰り返すことにより全体の解を求める。

部分逐次緩和法では領域分割法の内、図 12のように各領域に重なりがなく格子の中間を接続境界とするものを扱う。そして、部分逐次緩和法は領域分割法の通信タイミングのうち、同一改良時間でない最新の値を用いるタイミングで通信を行う。すなわち、更新中のものは使わず、また昔のデータは使用しない。

4 並列ループ for_slice 構文

先に述べた通り`for_slice`はプロセッサ間の依存関係を緩和するために用いる並列ループである。この章では`for_slice`の目的および`for_slice`の仕様、そして我々が開発している自動並列化コンパイラ TINPAR への実装方法を述べる。

4.1 for_slice の目的

並列化コンパイラは、逐次言語で記述されたプログラムを用いることにより、繁雑な作業を軽減しつつ高い効果を出すことを目的としている。しかし、逐次言語は本来逐次的なアルゴリズムを記述するためのものであり、高度に並列化を施したプログラムを記述することは必ずしも容易ではない。

その代表的な例として、ループ運搬依存によって生じるプロセッサ間の通信があげられる。SOR 法のように、配列データの隣接する部分に依存があると、その結果、プロセッサの処理の開始時刻、終了時刻が依存により順序付けされる。各プロセッサが同一時刻に処理を行えず、並列化の効果を得ることができなくなる。

また、パイプラインによって並列効果を得ようとすると、収束判定のような同期処理がパイプライン処理を妨げないようにするために多大な苦労を必要とする。

一方、領域分割法による並列化のようにプロセッサ間のループ運搬依存を除去すると、各プロセッサの、開始時刻や終了時刻をほぼ同じにすることができる。同期処理も簡単であり、並列化効果を十分得ることができる。

しかし、このような並列処理は、`for`でも`forall`でもだめである。したがって、プロセッサ間のみ依存を緩和するような特殊な並列処理を行いたいユーザは、自分で全てを行わなくてはならない。そこで、逐次言語で書かれたプログラムに若干の変更を施すだけで、追加で簡単にプロセッサ間の依存を緩和できるような構文を考える必要がある。それが、今回提案する並列ループ`for_slice`である。

`for_slice`ではループ中の見かけ上のループ運搬依存が二種類のものに区別している。それはプロセッサ内依存とプロセッサ間依存である。プロセッサ内依存の場合

は従来通りの効率のよい計算を行うため依存をそのままにしている。一方、プロセッサ間依存の場合は各プロセッサが同一時刻に計算を行うため、依存を除去する。しかし、計算には隣接するプロセッサのデータが必要なため、本来依存を生じさせる更新後のデータを使用せず、更新前のデータを使用する。これは、データ分割によって生じる通信をループの前方に移動することと一致する。

4.2 for_slice の仕様

`for_slice` の構文を以下に示す。

```
for_slice ループ変数 = ループ開始値, ループ終了値, ループステップ値 do
    ループ本体
endfor
```

上記のように、for 文が`for_slice` 文へと変わっただけで、その他のループの制御値の記述方法には変化がない。

次に、`for_slice` がどのようにな動作するのかを、等価な逐次プログラムを用いて示す。例として、次のプログラムを用いる。

例 1

```
const n = 8
processors 4
integer a(0:n - 1:2)
integer b(0:n - 1:2)

for_slice i = 1, n - 2 do
    a(i) = a(i) - a(i - 1)
    b(i) = a(i + 1) + a(i)
endfor
```

このプログラムではプロセッサ台数が 4 となっており、配列 a、b はブロックサイズ 2 で分割される。`for_slice` 構文ではプロセッサ間での`a(i-1)`→ `a(i)` が除去される。例 1 と等価な逐次プログラムを例 2 に表す。

例 2

```
const n = 8
integer a(0:n - 1)
integer tmp_a(0:3, 0:n - 1)
integer b(0:n - 1)

for j = 0, 3 do
    for i = 0, n - 1 do
        tmp_a(j, i) = a(i)
    endfor
endfor
for j = 0, 3 do
    for i = max(j * 2, 1), \
            min(j * 2 + 1, n - 2) do
        tmp_a(j, i) = tmp_a(j, i) \
                    - tmp_a(j, i - 1)
        b(i) = tmp_a(j, i + 1) \
             + tmp_a(j, i)
    endfor
endfor
for j = 0, 3 do
    for i = j * 2 , j * 2 + 1 do
        a(i) = tmp_a(j, i)
    endfor
endfor
```

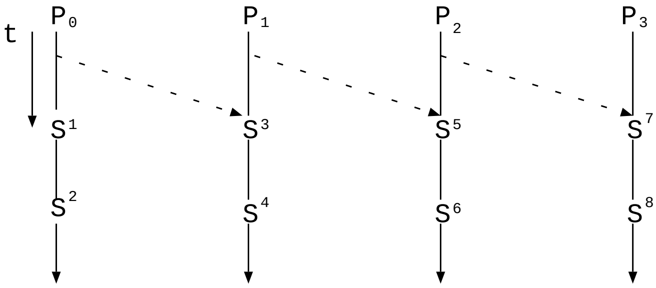


図 13: プロセッサ数 4、ブロックサイズ 2 の時の for_slice の処理

この逐次プログラムにおいて、ループ変数`j` の各 for 文がプロセッサ毎の処理を表している。ループ変数`i` で行われる処理は各プロセッサに分割されたイタレーションを行うためのループである。

まず、最初の j ループの中ではプロセッサ間の依存を緩和するために前もって`a(i)` の値を通信しておくことと等価なことを実行している。ここでは、各プロセッサに対して全ての要素をコピーしている。次の j ループの本体は実際の計算の部分を表している。ここでは、計算に使うのは一時配列である。最後の j ループでは所有データに真の値を書き戻している。これらはあくまで等価な動作をする逐次プログラムのための処理で、実際の並列プログラムでは各プロセッサ毎に`a` を全て所有することができるので、`tmp_a` をわざわざ用意して計算部分に使用したり、書き戻したりする必要はない。分散メモリ型並列計算機ではプロセッサが所有していないデータの受渡しは全て通信で行わねばならないため、実際には必要なデータしか通信を行っていない。したがって、処理の流れは図 13に示すものとなる。

また、例としては示さないが`for_slice` のループ変数とデータ分割された次元の添字式が関係ない場合、`for_slice` の機能は働かない。

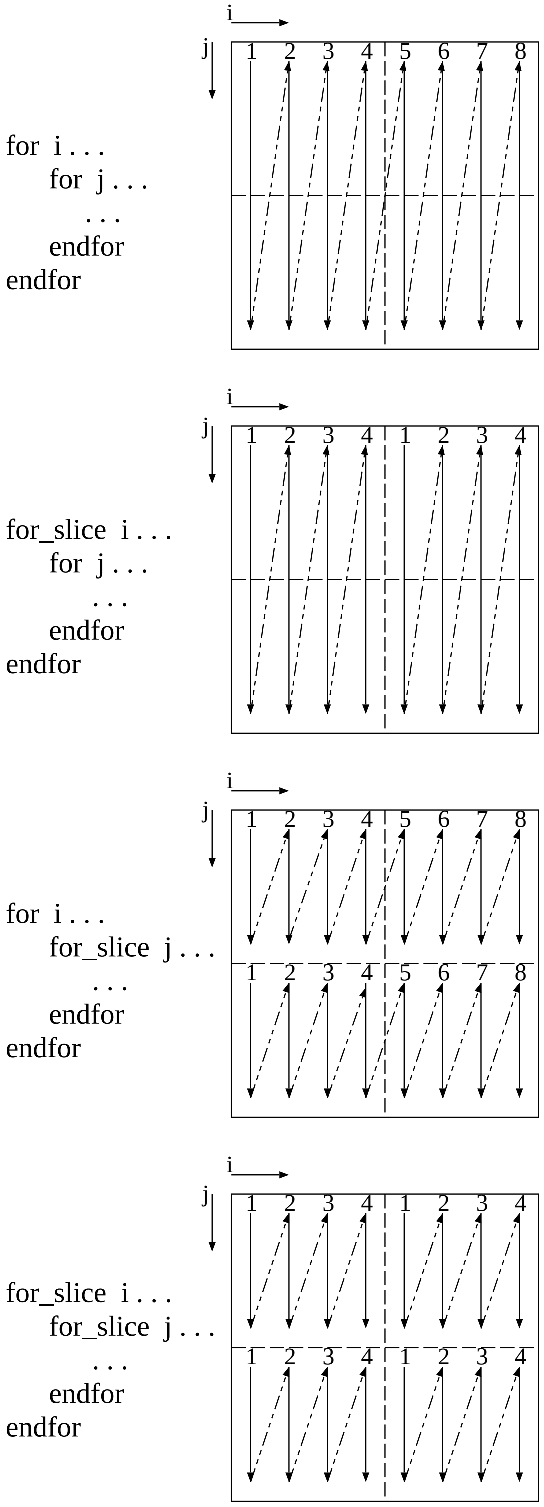
4.2.1 for_slice 構文の入れ子

先ほどの例では一次元配列のみの扱ったが、二次元配列を二次元分割した時の扱いを考えてみる。

ここで次の例をあげてみる。

例 3

```
for i = 1, n - 2 do
  for j = 1, m - 2 do
    u(i, j) \
    = (u(i - 1, j) + u(i, j - 1) \
```



```
for i ...
    for j ...
        ...
    endfor
endfor
```

```
for_slice i ...
    for j ...
        ...
    endfor
endfor
```

```
for i ...
    for_slice j ...
        ...
    endfor
endfor
```

```
for_slice i ...
    for_slice j ...
        ...
    endfor
endfor
```

図 14: for_slice と for の組み合わせによる計算順序の違い

```
    + u(i , j + 1) + u(i + 1, j) \
    - g(i, j)) * (omega / 4) \
    + u(i, j) * (1 - omega)
  endfor
endfor
```

これは SOR 法の改良の本体部分である。仮に四つのプロセッサに均等に二次元分割を施したと考える。そこで実際にどのような流れで処理が行われているかを図 14 に示す。図に示される通り、for 文と`for_slice` 文の組み合わせにより四種類の処理方式が存在することになる。これは二次元分割をした場合、`for_slice` による依存の緩和がどこに対して行われるかを表している。そのため、二次元分割を行った場合に依存をどう残すか、全て同時実行するかを選択できる。

ここで二次元分割を行った場合、自らが所有するデータ領域に対して`for_slice` がどこの領域の非所有データとの依存を緩和するかを図 15に示しておく。このような形式をとっているため、自分に対して斜めの格子点の計算に使用する場合、片方の`for_slice` では`for_slice` による並列実行はできないため、`i` と`j` の両方のループを`for_slice` にしておく必要がある。

以上が`for_slice` 文が並列計算機上でどのような実現することについて述べた。`for_slice` による並列化は HPF の`INDEPENDENT` ディレクティブを使用しての実現はおそらく可能であろう。その時は、プロセッサ数、データ分割幅などの並列計算機特有の煩わしい処理を自分で行うことになる。これは並列化コンパイラの存在意義に反すると思われるので、`for_slice` のようなものは必要であると考えている。

4.3 for_slice の TINPAR への実装

4.3.1 TINPAR の概要

TINPAR は拡張 Tiny language で記述された逐次プログラムを並列化する。Tiny language とはループ再構成ツール Tiny[1] のために設計された言語であり、これに我々はデータ分割ディレクティブ、`while` 構文およびスコープ規則を追加した。なお現在指定できるデータ分割は、単一次元方向に対するブロック、サイクリックおよびブロックサイクリック分割である。

TINPAR のオブジェクトコードは、C 言語による SPMD プログラムである。これはさらにコンパイルされ、専用通信ライブラリとリンクされて実行形式となる。こうすることにより、1) プロセッサに依存した最適化を C コンパイラに任せることができる、2) 対象並列計算機の変更が比較的容易である、などの利点が生じる。

次に、TINPAR の構成を示す。TINPAR は以下の 6 つのモジュールから構成される。

1. 構文解析部

   ソースプログラムを構文解析し、処理系の内部表現に展開する。

2. 前処理部

   逐次プログラムには、そのままでは並列化に適さない部分が存在することがある。そこで本モジュールでは、このような部分にプログラム変換を行って並

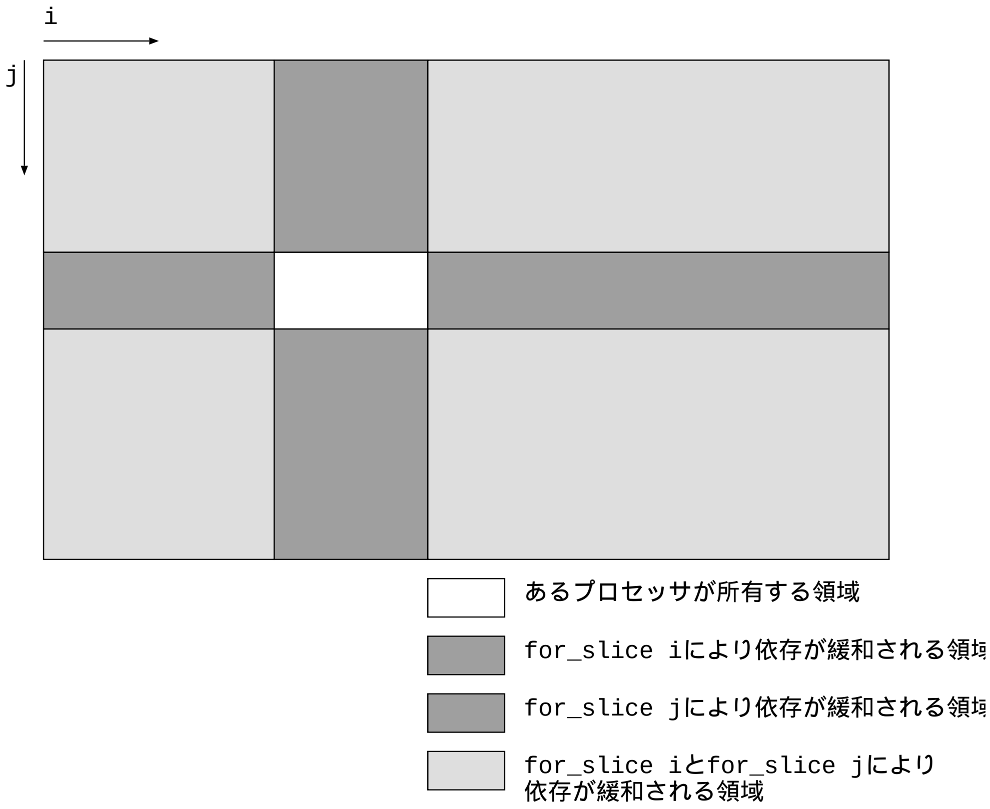


図 15: for_slice の組み合わせによる for_slice 処理の有効範囲

列化に適した形に変形する。この変換の例としては、総和演算のイディオム変換、リモートデータに対する一時的なローカルコピーの作成/利用などが挙げられる。

3. 並列化部

   owner computes rule に従ってソースプログラムを並列化する。具体的には以下の処理を行う (図 16 参照)。

   (a) `if` 文の挿入

      分割されたデータに対する代入は、当該データの所有者プロセッサのみが行う。そこでこのようなデータに対する代入文には、所有者プロセッサを判定するための`if` および`endif` 文を前後に挿入する。

   (b) `SEND`/`RECV` 文の挿入

      分割されたデータに対する参照は、当該データの所有者プロセッサと参照者プロセッサの間のメッセージ通信で実現される。そこでこのようなデータに対する参照には、所有者プロセッサによる送信のための`SEND` 文、参照者プロセッサによる受信のための`RECV` 文、および両者を判定するための`if`/`endif` 文を挿入する。

   またこの時点では、分割されたデータに対して、分割前のデータ全体を格納できる大きさのメモリ領域を確保する。この結果、分割されたデータに対するアクセスは分割前と同様のアドレッシングで可能となるが、実際に使用されるメモリ領域は各プロセッサに割り付けられたデータを格納する部分のみとなる。

4. 最適化部

   並列化部で用いている並列化手法は極めて単純であり、コード的に無駄が多く効率もよくない。そこで本モジュールでは、不用な`if` 文の削除や`SEND` 文のリスケジューリングなどの最適化を行う。

5. メモリ再割当部

   分割されたデータに対する並列化部のメモリ割り当て手法は、最適化部によるデータ依存関係解析を容易にするが、確保されるものの全く使用されないメモリ領域が存在することになって無駄も多い。そこで本モジュールでは、分割されたデータに対するメモリ割り当て手法の改善およびそれに伴うアドレッシングの修正を行う。

6. コード生成部

   処理系の内部表現から C によるオブジェクトコードを出力する。

4.3.2 for_slice の実装

TINPAR の内部処理構造は並列化部、最適化部、メモリ再配置部などからなっているが、今回の実装では`for_slice` のための処理は並列化部で行うこととした。この段階では、最適化を行っていないため、どのループのどの計算式が for_slice に適用されるかが明白である。ま

```
real a(0:n - 1:*)
real b(0:n - 1:1)
/*  a() はブロックク分割、b() はサイクリック分割 */

for i = 0, n - 1 do
  b(i) = a(i)
endfor
```

⇓

```
real a(0:n - 1:*)
real b(0:n - 1:1)

for i = 0, n - 1 do
  if OWNER(a(i)) == GET_CELL_ID() then
    SEND(OWNER(b(i)), a(i))
  endif
  if OWNER(b(i)) == GET_CELL_ID() then
    RECV(OWNER(a(i)), tmp_a)
    b(i) = tmp_a
  endif
endfor
```

図 16: 並列化

た、並列化部に`for_slice`のための処理を置くことにより、その後の最適化部をそのまま利用できることになる。

TINPAR の並列化部は、左辺値のインデックスと右辺の変数の各々のインデックスを調べ、それら個別に非所有データかどうかを判定し、通信を挿入する処理を行っている。そのため、`for_slice`でも同様に非所有データの判定を流用することができる。すなわち、ある非所有データの分割された次元のインデックスにループ変数が含まれている場合、そこで起こる通信をループの前方に移動すればよい。なお、`SEND`文のみを前方に移動すると、`for_slice`に関するもの以外の通信が間に入る可能性があるため、`RECV`文の順序が狂う可能性がある。そこで、`for_slice`が適用される通信に関しては、`RECV`文も前方に移動して通信の順序を保証している。

並列化部の処理の流れは 1 パスで処理を行っているため、`for_slice`により前方に移動させられる通信が発生した場合、それが適切な場所へ移動できるように記憶しておく必要がある。そのため、`for_slice`のブロック内で記憶されるデータは次のようになる。

- `for_slice`により移動される通信の挿入場所
  `for_slice`は前方に自分と同じ形の for 文を作って、そこに通信を挿入している。これにより必要なデータが確保できるのだがこれはループの構造に対して正しい場所に挿入できるようにするために、記憶されている。
- `for_slice`のループ変数
  非所有データの分割次元が`for_slice`のループ変数に関係していることを探さねば通信処理を変更できない。そのため、あらかじめ`for_slice`のループ変数を取り出しておく。
- `for_slice`の入れ子のためのポインタ
  `for_slice`の入れ子を認めているため、次の入れ子レベルの`for_slice`の挿入ポイントを記憶しておく。

次に、`for_slice`を含めた並列化の手法を示す。

for_slice 文

- 外側に`for_slice`文がない場合:
  図 17(a) から (b) に至る過程で示す。`for_slice`文により通信を前方に移動するための for 文を`for_slice`文の前方に挿入し、ループ変数を記憶する。通信の挿入場所は for 文の中である。
- 外側に`for_slice`文がある場合:
  図 17(b) から (c) に至る過程で示す。外側の`for_slice`文が前方に挿入した通信のための for 文に同じ for 文の入れ子と同じ形になるように for 文を挿入する。外側の`for_slice`の通信の挿入場所は新しく挿入した for 文の中とする。あとは外側に`for_slice`がない場合と同様な処理を行う。すなわち、自分の前方にも for 文を挿入する。

for 文

- 外側に`for_slice`文がない場合:
  何もしない。
- 外側に`for_slice`文がある場合:
  外側の`for_slice`文が前方に挿入した通信のための for 文に同じ for 文の入れ子と同じ形になるように for 文を挿入する。外側の`for_slice`の通信の挿入場所は新しく挿入した for 文の中とする。

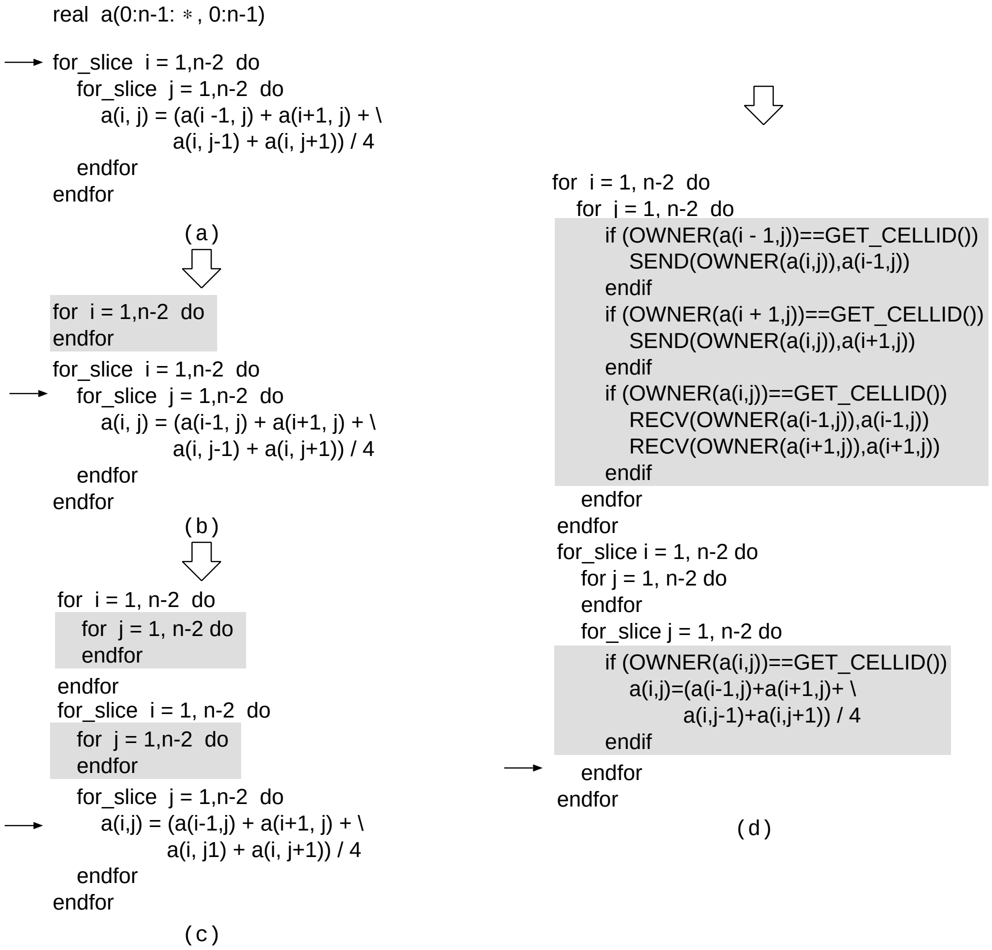
endfor 文

for 文と`for_slice`文は同様に endfor 文を用いているが、内部では区別されている。

- 外側に`for_slice`文がない場合:
  何もしない。
- 外側に`for_slice`文がある場合: `for_slice`のための挿入記憶リストの全てのノードの挿入ポイントを自分の分身の endfor の後ろへずらす。また、ループヘッダが`for_slice`文の場合、対応する`for_slice`文用の挿入記憶ノードを解放する。

代入文

- 外側に`for_slice`文がない場合:
  送信データ、受信データの所有者を示す条件を代入文の直前に挿入し、その条件文の中に通信コードを挿入する。この時、代入文は受信条件のための if 文に挿入される。



```
real  a(0:n-1: ∗ , 0:n-1)

for_slice  i = 1,n-2  do
   for_slice  j = 1,n-2  do
      a(i, j) = (a(i -1, j) + a(i+1, j) + \
               a(i, j-1) + a(i, j+1)) / 4
   endfor
endfor
```

(a)

```
for  i = 1,n-2  do
endfor
for_slice  i = 1,n-2  do
   for_slice  j = 1,n-2  do
      a(i, j) = (a(i-1, j) + a(i+1, j) + \
               a(i, j-1) + a(i, j+1)) / 4
   endfor
endfor
```

(b)

```
for  i = 1, n-2  do
   for  j = 1, n-2 do
   endfor
endfor
for_slice  i = 1, n-2  do
   for  j = 1,n-2  do
   endfor
   for_slice  j = 1,n-2  do
      a(i,j) = (a(i-1,j) + a(i+1, j) + \
              a(i, j1) + a(i, j+1)) / 4
   endfor
endfor
```

(c)

```
for  i = 1, n-2  do
   for  j = 1, n-2  do
      if (OWNER(a(i - 1,j))==GET_CELLID())
         SEND(OWNER(a(i,j)),a(i-1,j))
      endif
      if (OWNER(a(i + 1,j))==GET_CELLID())
         SEND(OWNER(a(i,j)),a(i+1,j))
      endif
      if (OWNER(a(i,j))==GET_CELLID())
         RECV(OWNER(a(i-1,j)),a(i-1,j))
         RECV(OWNER(a(i+1,j)),a(i+1,j))
      endif
   endfor
endfor
for_slice i = 1, n-2 do
   for j = 1, n-2 do
   endfor
   for_slice j = 1, n-2 do
      if (OWNER(a(i,j))==GET_CELLID())
         a(i,j)=(a(i-1,j)+a(i+1,j)+ \
                a(i,j-1)+a(i,j+1)) / 4
      endif
   endfor
endfor
```

(d)

図 17: for_slice による変換の推移

- 外側に`for_slice`文がある場合:
  非所有データと判定された配列データの分割次元に関する添字式との判定を行う。一番外側の`for_slice`文から判定を行い、前方に挿入したfor文の挿入データの示す場所に送信データ、受信データの所有者を示すif文を挿入し、そのif文の中に通信コードを挿入する。この時、代入文を囲うように受信条件と同じ形のif文が挿入される。

  ループ変数と関係のない場合、通常の処理を行う。

その他

while文やif文などは条件によって通信が変化するが、通信を前方に移動してしまう`for_slice`の場合、データを使用しなくても先に通信を行ってしまうだけなので、特別な処理は行っていない。

4.3.3 オーバーラップエリアの実装

係数行列が疎である連立一次方程式を解く反復法のアルゴリズムでは、参照されるデータが所有しているデータの近傍に限られているため、オーバーラップエリアという非所有データのために領域を確保する方法が提案されている[4]。このオーバーラップエリアを用いると、通信のために一時変数を除去でき、所有変数の一時変数への代入や、非所有変数のために一時変数を用いる例外処理のための分岐といった、余分なコードが不要になる。

ところで、`for_slice`はプロセッサ間の依存を緩和するために送信を前方に移動する。しかし、`for_slice`と関係のない通信が間に入る可能性があり、その時には受信順序が保てない。そのため、それぞれのデータを正しい場所に格納するための場所が必要である。したがって、通常とは違い通信のための一時変数は配列型である必要がある。TINPARの並列化部はブロックが違うと必要に応じて別の一時変数を作っている。オーバーラップエリアを利用せず、この仕様に従うと無駄な一時配列変数を大量に作る可能性がある。よって、TINPARにおいて`for_slice`のためにオーバーラップエリアを実装するのはメモリ利用効率の面で重要である。

そこで、オーバーラップエリアのサイズを指定するディレクティブを追加し、TINPARに実装した。

```
real a(0:n - 1:3:1)
/*  a() はサイズ 3 でブロック分割、

   オーバーラップエリアのサイズは 1 */

for i = 2, n - 1 do
  a(i) = a(i - 2)
endfor
```

図18: オーバーラップエリアの追加ディレクティブ

図18はオーバーラップエリアの例でありはブロックサイズ3で分割される配列`a`にサイズが1のオーバーラップエリアを付加している。

次にオーバーラップエリアがある場合の分割領域のメモリ再配置後のインデックスを求め方について述べる。まず、すべてのプロセッサのインデックス計算を同様に扱うため、通常の分割時と違い、図19の下の図に示すように両端のプロセッサにも使用されないオーバーラップエリアを確保しておく。

次に所有者をOWNER()、オーバーラップエリアを持つデータの分割後の添字をLOCAL_INDEX_OLA()とする。ここで、分割された次元の添字をi_k、分割開始位置のオフセット値をOS、ブロックサイズをBS、プロセッサ数をPN 計算ownerとの依存距離をD、オーバーラップエリアサイズをOLAのそれぞれの変数で与える。これにより、OWNER()、LOCAL_INDEX_OLA()は以下の式で与えられる。

$$\mathrm{OWNER}(i_k) = \lfloor \frac{i_k - OS}{BS} \rfloor \bmod N$$

$$\begin{aligned}
&\mathrm{LOCAL\_INDEX\_OLA}(i_k)\\
&= \lfloor \frac{i_k - D - OS}{BS \times PN} \rfloor \times (BS + OLA \times 2)\\
&+ (i_k - D - OS) \bmod BS + OLA + D
\end{aligned}$$

一方データ分割が施されたk次元めに関して、各プロセッサが所有する配列要素数N_LOCAL_ELEMENTSは分割された次元の配列要素数を変数Nとすると、次の式で与えられる。

$$\begin{aligned}
\mathrm{N\_LOCAL\_ELEMENTS} &= \lceil \tfrac{N}{PN} \rceil\\
&+2 \times OLA \times \lceil \tfrac{N}{PN} \rceil \div BS
\end{aligned}$$

また、従来のTINPARではメッセージ通信による受信処理を受信用一時変数を用いて行っていた。しかしそこで、並列化部で以下の処理を追加した。

- 外側に`for_slice`文がない場合:
  オーバーラップエリアが指定されていれば、通信対象データと左辺値との依存距離を計算する。この距離がオーバーラップエリアサイズを越えていないなら、一時変数を使用せず、オーバーラップエリアを使用するように変更する。

  ディレクティブが無指定、あるいは依存距離がオーバーラップエリアサイズを越えていれば、オーバーラップエリアの処理を行わない。

- 外側に`for_slice`文がある場合:
  まず、オーバーラップエリアサイズの指定の有無に関わらず、通信のための一時変数を使用しない。

  そして、オーバーラップエリアが指定してある場合、通信対象データと左辺値との依存距離を計算し、この距離がオーバーラップエリアサイズを越えていないなら、オーバーラップエリアを使用するようにプログラムの内容を変更する。

  一方、ディレクティブが無指定、あるいは依存距離がオーバーラップエリアサイズを越えていれば、メモリ再割り当て部における処理を該当する配列に関して禁止する。この結果、`for_slice`を使用している場合、オーバーラップエリアの指定が適格でないならば、本来所有していない全ての領域を確保することになる。

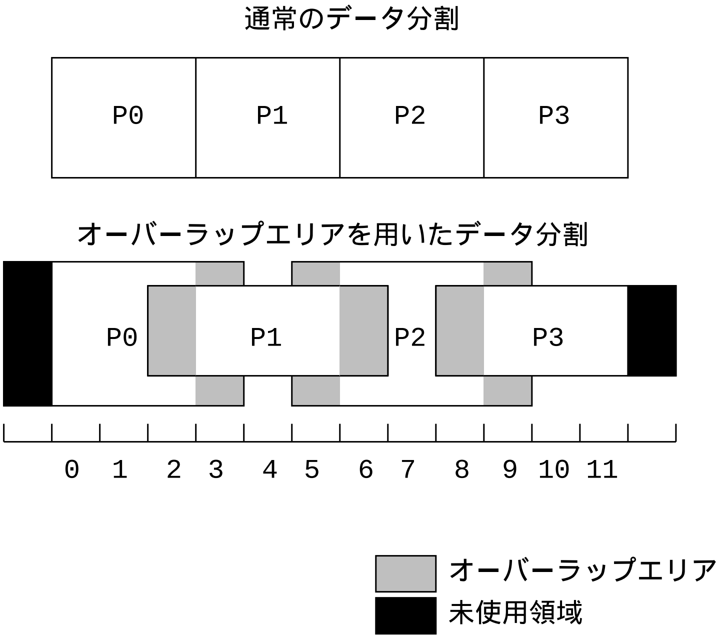


図 19: オーバーラップエリアを用いた時のメモリ再割り当て方式

5 for_slice 使用による収束効率の変化に対する考察

`for_slice` を用いて連立一次方程式の反復求解を行うと、収束効率が悪化することが予想される。したがって、収束効率の悪化がどの程度のものかを示すことができれば、`for_slice` を適切に利用するための指針を得ることができる。しかし、収束効率を解析的に求めるのは極めて困難である。そこで、ここでは SOR 法における緩和係数が収束効率に与える影響を定性的に議論し、さらに実験によりその影響を明白にする。

まず、二次元問題を扱うと複雑になるので一次元の場合を考える。一次元のラプラス方程式を、離散化した結果の連立一次方程式は格子点数を M とすると

$$\begin{pmatrix} -2 & 1 & & & & & \\ 1 & \ddots & \ddots & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & \ddots & \ddots & 1 \\ & & & & & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_M \end{pmatrix} = \mathbf{0}$$

となる。この方程式はを係数行列が三重対角であるので、直接法で解くことは可能であるが、あえて反復法を用いてで解を求めることとする。

SOR 法を用いると、$n+1$ 回目の反復で得られる u_j の近似解は

$$u_j^{n+1} = (1-\omega)u_j^n + \frac{\omega}{2}(u_{j-1}^{n+1} + u_{j+1}^n)$$

となる。ただし ω は緩和係数である。この式から以下の関係式を得る。

$$\begin{aligned} u_1^{n+1} &= (1-\omega)u_1^n + \tfrac{\omega}{2}u_2^n \\ u_2^{n+1} &= \tfrac{\omega}{2}(1-\omega)u_1^n + (\tfrac{\omega^2}{4} + (1-\omega))u_2^n + \tfrac{\omega}{2}u_3^n \\ u_3^{n+1} &= \tfrac{\omega^2}{4}(1-\omega)u_1^n + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega))u_2^n \\ & \quad + (\tfrac{\omega^2}{4} + (1-\omega))u_3^n + \tfrac{\omega}{2}u_4^n \\ & \vdots \end{aligned}$$

ここで、反復法の一般式は以下のように表すことができる。

$$\mathbf{u}^{n+1} = \mathbf{S}\mathbf{u}^n + \mathbf{b}$$

この変換行列 S の最大固有値が SOR 法のスペクトル半径と呼ばれ、誤差ベクトルが 0 に近付くためには最大固有値が 1 より小さい必要がある。そのため、この行列の最大固有値は収束に関わりのある値となる。

同様にして、u_{j-2} と u_{j-1} にデータの境界があり、その間に生じるプロセッサ間の通信を `for_slice` で緩和したと仮定した場合、以下のように表せる。

$$\begin{aligned}
u_1^{n+1} &= (1-\omega)u_1^n + \tfrac{\omega}{2}u_2^n \\
u_2^{n+1} &= \tfrac{\omega}{2}(1-\omega)u_1^n + (\tfrac{\omega^2}{4} + (1-\omega))u_2^n + \tfrac{\omega}{2}u_3^n \\
u_3^{n+1} &= \tfrac{\omega^2}{4}(1-\omega)u_1^n + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega))u_2^n \\
&+(\tfrac{\omega^2}{4} + (1-\omega))u_3^n + \tfrac{\omega}{2}u_4^n \\
&\vdots \\
u_{j-1}^{n+1} &= \tfrac{\omega}{2}u_{j-2}^n + (1-\omega)u_{j-1}^n + \tfrac{\omega}{2}u_j^n \\
u_j^{n+1} &= \tfrac{\omega^2}{4}u_{j-2}^n + \tfrac{\omega}{2}(1-\omega)u_{j-1}^n \\
&+(\tfrac{\omega^2}{4} + (1-\omega))u_j^n + \tfrac{\omega}{2}u_{j+1}^n \\
u_{j+1}^{n+1} &= \tfrac{\omega^3}{8}u_{j-2}^n + \tfrac{\omega^2}{4}(1-\omega)u_{j-1}^n + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega))u_j^n \\
&+(\tfrac{\omega^2}{4} + (1-\omega))u_{j+1}^n + \tfrac{\omega}{2}u_{j+2}^n \\
&\vdots
\end{aligned}$$

またこの関係式から得られる変換行列 $\mathbf{S}^{'}$ とすると、$\mathbf{S}$ と $\mathbf{S}^{'}$ の最大固有値を比較することにより、収束の速さ、数値の安定性を推測することができる。しかし、$\mathbf{S}$ や $\mathbf{S}^{'}$ の最大固有値を解析的には困難であるので、その指標としてゲルシュゴリンの円の定理に基づく固有値の上限を利用する。

行列 $\mathbf{A}$ の固有値の上限は

$$|\lambda - a_{ii}| \leq \sum_{j \neq i}^{N} |a_{ij}| \tag{11}$$

で与えられる。この式は、中心が対角成分 a_{ii} で、半径が各行の非対角成分の絶対値の和により与えられている複素平面上に存在する円の内部に $\mathbf{A}$ の固有値が存在することを示している。

この定理を用い、変換行列 $\mathbf{S}$ と $\mathbf{S}^{'}$ の固有値の上限について考察を行う。まず、SOR 法に関する変換行列 $\mathbf{S}$ については、緩和係数が $0 < \omega < 2$ の範囲であるので、ゲルシュゴリンの円の半径は以下の値となる。

$\omega \leq 1$ の場合

$$\begin{aligned}
r_1 &= \tfrac{\omega}{2} \\
r_2 &= \tfrac{\omega}{2}(1-\omega) + \tfrac{\omega}{2} = \omega(1-\tfrac{\omega}{2}) \\
r_3 &= \tfrac{\omega^2}{4}(1-\omega) + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}-\tfrac{\omega^2}{8}) \\
r_4 &= \tfrac{\omega^3}{8}(1-\omega) + (\tfrac{\omega^4}{16} + \tfrac{\omega^2}{4}(1-\omega)) \\
&+(\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}-\tfrac{\omega^3}{16}) \\
&\vdots
\end{aligned}$$

$\omega > 1$ の場合

$$\begin{aligned}
r_1 &= \tfrac{\omega}{2} \\
r_2 &= \tfrac{\omega}{2}(\omega-1) + \tfrac{\omega}{2} = \tfrac{\omega^2}{2} \\
r_3 &= \tfrac{\omega^2}{4}(\omega-1) + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{3}{4}\omega+\tfrac{3}{8}\omega^2) \\
r_4 &= \tfrac{\omega^3}{8}(\omega-1) + (\tfrac{\omega^4}{16} + \tfrac{\omega^2}{4}(1-\omega)) \\
&+(\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}-\tfrac{\omega^2}{4}+\tfrac{3}{16}\omega^3) \\
&\vdots
\end{aligned}$$

それぞれの場合、$r_2, r_3, r_4, \ldots$ を半径とする円は中心は全て等しく $\frac{\omega^2}{4} + (1-\omega)$ であり、$0 < \omega < 2$ では正の実数である。

$\omega \leq 1$ の場合、$r_2, r_3, r_4, \ldots$ の各半径の最大値は $\omega = 1$ の時でその値は $\frac{3}{4}$ を越えない。また、中心との和をとっても 1 を越えない。すなわち、絶対に安定である。一方、$\omega > 1$ の場合、ω が大きくなると中心と半径の絶対値の和は安定値 1 を越える場合がでてくることが分かる。しかし、すぐにこれが不安定な解を生成することを意味するわけではない。

解析的には、SOR 法の固有値は次の式で与えられている。

$$\lambda_m = \frac{1}{4}\left(\omega\cos(\frac{m\pi}{M+1}) \pm \sqrt{\omega^2\cos(\frac{m\pi}{M+1})^2 + 4(1-\omega)}\right)^2$$

この式が与えられていることより SOR 法は安定であることはいえる。ゲルシュゴリンの円の定理は円内に固有値が存在しているだけで、必ずしも円周上の値をとるわけではない。

次に、`for_slice` の場合は以下のような結果になる。

$\omega \leq 1$ の場合

$$\begin{aligned}
r_1 &= \tfrac{\omega}{2} \\
r_2 &= \tfrac{\omega}{2}(1-\omega) + \tfrac{\omega}{2} = \omega(1-\tfrac{\omega}{2}) \\
r_3 &= \tfrac{\omega^2}{4}(1-\omega) + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}-\tfrac{\omega^2}{8}) \\
r_4 &= \tfrac{\omega^3}{8}(1-\omega) + (\tfrac{\omega^4}{16} + \tfrac{\omega^2}{4}(1-\omega)) \\
&+(\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}-\tfrac{\omega^3}{16}) \\
&\vdots \\
r_{j-1} &= \tfrac{\omega}{2} + \tfrac{\omega}{2} = \omega \\
r_j &= \tfrac{\omega^2}{4} + \tfrac{\omega}{2}(1-\omega) + \tfrac{\omega}{2} = \omega(1-\tfrac{\omega}{4}) \\
r_{j+1} &= \tfrac{\omega^3}{8} + \tfrac{\omega^2}{4}(1-\omega) + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}) \\
r_{j+2} &= \tfrac{\omega^4}{16} + \tfrac{\omega^3}{8}(1-\omega) + (\tfrac{\omega^4}{16} + \tfrac{\omega^2}{4}(1-\omega)) \\
&+(\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}) \\
&\vdots
\end{aligned}$$

$\omega > 1$ の場合

$$\begin{aligned}
r_1 &= \tfrac{\omega}{2} \\
r_2 &= \tfrac{\omega}{2}(\omega-1) + \tfrac{\omega}{2} = \tfrac{\omega^2}{2} \\
r_3 &= \tfrac{\omega^2}{4}(\omega-1) + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{3}{4}\omega+\tfrac{3}{8}\omega^2) \\
r_4 &= \tfrac{\omega^3}{8}(\omega-1) + (\tfrac{\omega^4}{16} + \tfrac{\omega^2}{4}(1-\omega)) \\
&+(\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}-\tfrac{\omega^2}{4}+\tfrac{3}{16}\omega^3) \\
&\vdots \\
r_{j-1} &= \tfrac{\omega}{2} + \tfrac{\omega}{2} = \omega \\
r_j &= \tfrac{\omega^2}{4} + \tfrac{\omega}{2}(\omega-1) + \tfrac{\omega}{2} = \tfrac{3}{4}\omega^2 \\
r_{j+1} &= \tfrac{\omega^3}{8} + \tfrac{\omega^2}{4}(\omega-1) + (\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{3}{4}\omega+\tfrac{\omega^2}{2}) \\
r_{j+2} &= \tfrac{\omega^4}{16} + \tfrac{\omega^3}{8}(\omega-1) + (\tfrac{\omega^4}{16} + \tfrac{\omega^2}{4}(1-\omega)) \\
&+(\tfrac{\omega^3}{8} + \tfrac{\omega}{2}(1-\omega)) + \tfrac{\omega}{2} \\
&= \omega(1-\tfrac{\omega}{4}-\tfrac{\omega^2}{4}+\tfrac{\omega^3}{4}) \\
&\vdots
\end{aligned}$$

`for_slice` の場合も $\omega \leq 1$ では SOR 法の時と同様の議論で絶対安定であることがわかる。しかし、`for_slice` で依存が緩和された部分では確実に半径は大きくなっており最大固有値が若干大きくなっていることが予想される。

```
const n=12
const m=5
real u(0:n - 1:*:1, 0:m - 1)
real g(0:n - 1:*, 0:m - 1)
real omega

for i = 0, n - 1 do
  for j = 0, m - 1 do
    u(i, j) = 0.0
    g(i, j) = 0.0
  endfor
endfor
g(5,3)=1.0
for k = 1, 34 do
  for_slice i = 1, n - 2 do
    for j = 1, m - 2 do
      u(i, j) \
      = (u(i - 1, j) + u(i, j - 1) \
      + u(i, j + 1) + u(i + 1, j) \
      - g(i, j)) * (omega / 4) \
      + u(i, j) * (1 - omega)
    endfor
  endfor
endfor
```

図 20: 収束効率の考察に用いた SOR 法のプログラム

一方、$\omega > 1$ の範囲では`for_slice` で依存が緩和された部分では、半径が確実に大きくなっている分、通常の SOR 法よりも安定なωの条件は厳しくなっていることが予想される。また、プロセッサ数が多くなると`for_slice` で依存が緩和される部分が多くなるので、同半径の円の多重度が上がる。このため、固有値が円周に近付き、安定性が失われる可能性がある。

例として、ヤコビ法ではゲルシュゴリンの円の定理で全て中心 0、半径 1 であり、解析的に固有値が$\lambda = \cos(\frac{m\pi}{M+1})$ である。格子点数が増えていくと最大固有値の値が 1 に近付いていく。すなわち、円の多重度が上がると固有値が円周に近付くことが考えられる。

そこで、安定性に関して緩和係数がどのような役割を示すか調べてみる。例として、図 20に示す二次元の SOR 法のプログラムを使用する。このプログラムを用いて緩和係数とプロセッサ数を変化させた時の様子を図 21に示す。なお、$\omega = 1.5$ の時はプロセッサ数 12、$\omega = 1.7$ の時はプロセッサ数 4 以上で発散した。この図からわかるようにでは不足緩和の時は収束効率は全体的に良いとはいえないが、プロセッサ数の変化による収束効率の悪化は少ない。$\omega = 1$ では安定であるので収束しているが、プロセッサ数が増えるとゲルシュゴリンの円の多重度が上がるため、収束効率が若干悪化する。なっている。$\omega = 1.29$ となり徐々に値が大きくなると収束回数の悪化が激しくなってくるが、この問題は二次元であるので緩和係数が 1 に近い場合なら、ヤコビ法に緩和係数を用いた場合と違い一応収束している。ここでヤコビ法に緩和係数を用いた場合緩和係数は 1 以上で収束しなくなる。そして、緩和係数が 2 に近付くにつれて収束回数はさら

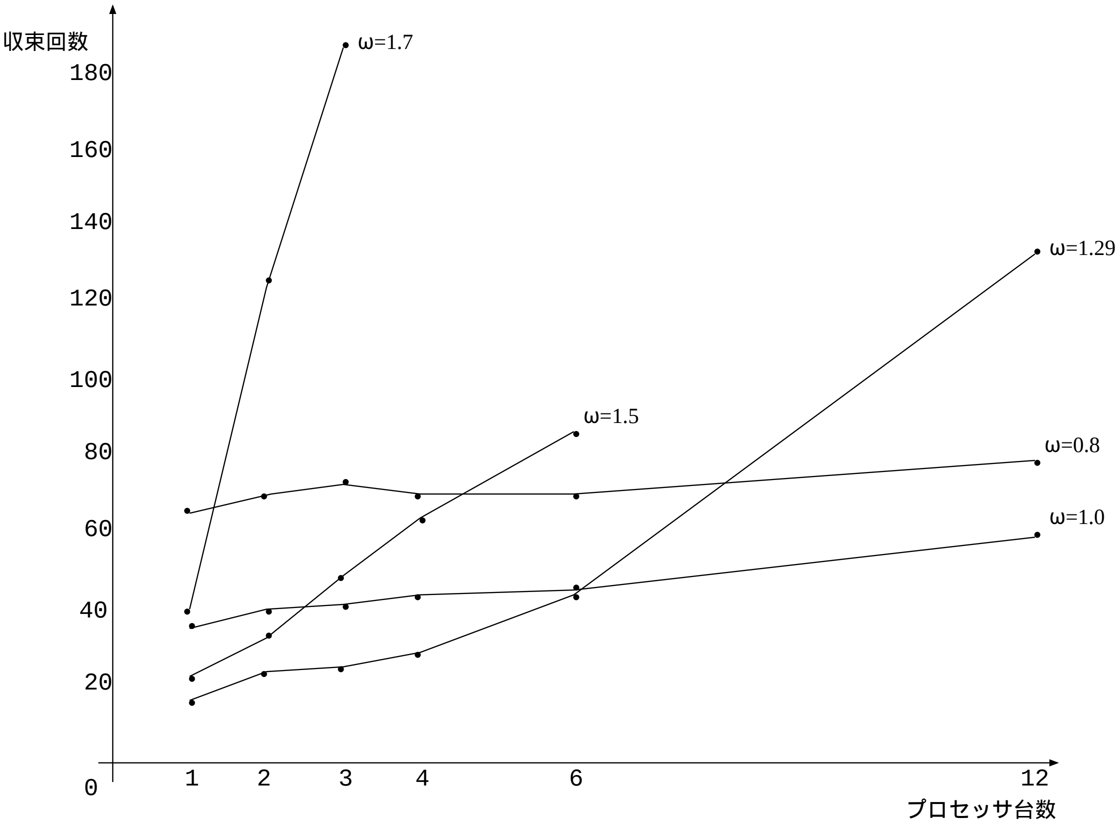


図 21: for_slice と緩和係数との関係

に増加の一途をたどり、またプロセッサ数が増えると発散する場合がある。

結論として、`for_slice` を用いると、緩和係数を大きくしたとき、発散の方向へ向かうことが分かった。そのため、過緩和により計算を行う場合、`for_slice` による処理を行わない方がよい。一方、TMDA の線アルゴリズムのように不足緩和を用いる場合、`for_slice` を用いることにより収束効率は悪化は小さいと考えられるので、`for_slice` を用いた並列化の効果により、全体の計算速度は大幅に向上するものと予想される。

6　性能評価

`for_slice` 構文を使用したプログラムと通常の for 文を並列化したものの比較を AP1000 を用いて評価する。

6.1　TINPAR による評価

TINPAR による実測評価として次のプログラムを用いた。データのサイズを縦横 512×512 とした SOR 法のプログラムである。本体ループのみの実測を行い、収束判定は行わない。本体のループ回数を 200 回とする。`for_slice` を適用するのは i ループとする。

結果を図 23に示す。ここで収束判定もしていない条件でこれだけの差が付くとは予想外であったが、AP1000 のパフォーマンスアナライザを調べてみた結果、TINPAR での通常の for の場合、図 24に示す動作をすることがわかった。TINPAR では一回の改良の終了前には決して次の改良のために必要なデータを送信しない。そのため、

```
const n=512
const m=512

real u(0:n - 1:*,0:m - 1)
real g(0:n - 1:*,0:m - 1)
real omega

for k = 1, 200 do
  for i = 1, n - 2 do
    for j = 1, m - 2 do
      u(i, j) \
      = (u(i + 1, j) + u(i, j - 1) \
      + u(i, j + 1) + u(i - 1, j) - g(i, j)) \
        * (omega / 4) + u(i, j) * (1 - omega)
    endfor
  endfor
endfor
```

図 22: TINPAR の実測評価に用いたプログラム

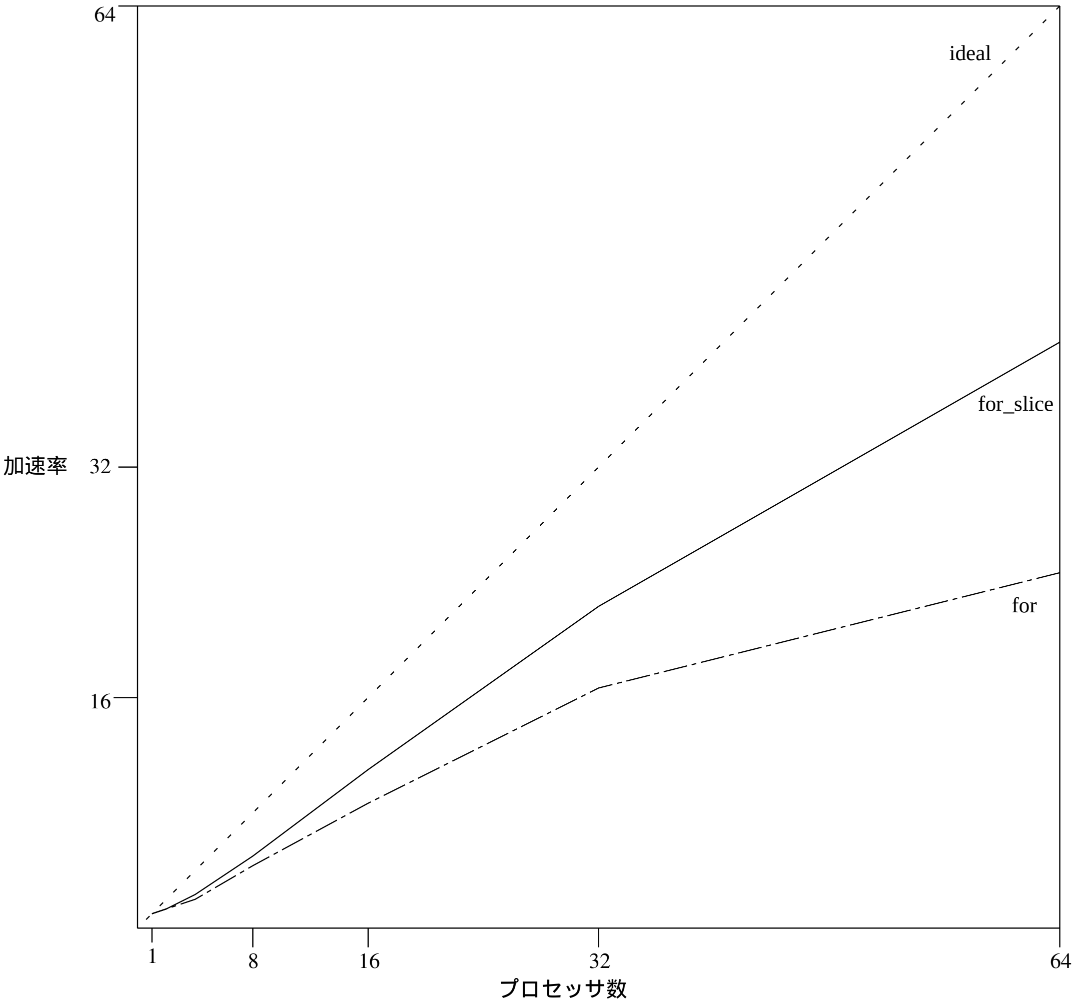


図 23: SOR 法での for と for_slice の実行時間の比較

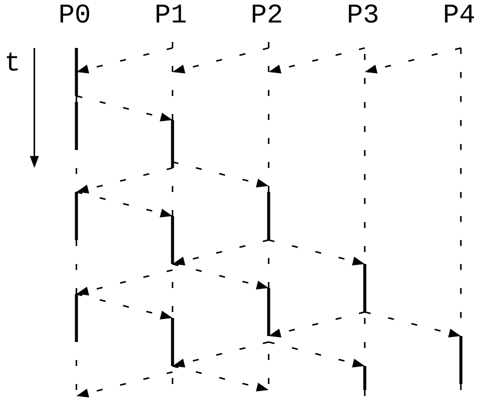


図 24: TINPAR の for の流れ

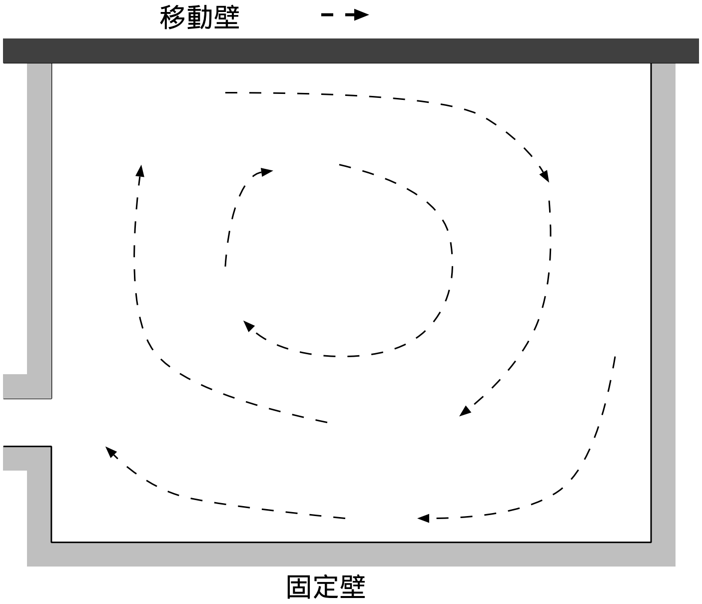


図 25: キャビティ流れの概略図

二回目以降の反復では大きな通信待ち時間が生じ、全体のスピード向上の妨げになったいる。これはTINPARに限らず、多くの並列化コンパイラでも同様であろうが、仮に収束判定のような非決定的にループを飛び出す場合、次の改良のためのデータを先行して送信できない。そのため、計算の開始終了の稼働時間が高く、待ち時間が少ない`for_slice`は単純な計算速度的には有利といえる。

6.2 ナビエストークス方程式による評価

ここでは、例としてナビエストークス方程式を解いた時の評価を行う。

ナビエストークス方程式は物理量ϕに対して次の方程式で表される。

$$\frac{\partial}{\partial t}(\rho\phi)+\frac{\partial}{\partial x}(\rho u\phi)+\frac{\partial}{\partial y}(\rho v\phi)=\frac{\partial}{\partial x}(\mathit{\Gamma}\frac{\partial\phi}{\partial x})+\frac{\partial}{\partial y}(\mathit{\Gamma}\frac{\partial\phi}{\partial y})+S \tag{12}$$

ここで、Γは拡散係数、Sは生成項である。この式は二次元問題の時、次の三つの式により表される。

$$\begin{gathered}\tfrac{\partial}{\partial t}(\rho u)+\tfrac{\partial}{\partial x}(\rho u^2)+\tfrac{\partial}{\partial y}(\rho uv)=\tfrac{\partial}{\partial x}(\mu\tfrac{\partial u}{\partial x})+\tfrac{\partial}{\partial y}(\mu\tfrac{\partial u}{\partial y})-\tfrac{\partial p}{\partial x}\\ \tfrac{\partial}{\partial t}(\rho v)+\tfrac{\partial}{\partial x}(\rho uv)+\tfrac{\partial}{\partial y}(\rho v^2)=\tfrac{\partial}{\partial x}(\mu\tfrac{\partial v}{\partial x})+\tfrac{\partial}{\partial y}(\mu\tfrac{\partial v}{\partial y})-\tfrac{\partial p}{\partial y}\\ \tfrac{\partial}{\partial t}(\rho)+\tfrac{\partial}{\partial x}(\rho u)+\tfrac{\partial}{\partial y}(\rho v)=0\end{gathered} \tag{13}$$

この方程式を離散化すると、次のような式になる。

$$\begin{gathered}\mathbf{A}_{\mathbf{u}}(\mathbf{u},\mathbf{v},\mathbf{p})\mathbf{u}=\mathbf{b}_{\mathbf{u}}\\ \mathbf{A}_{\mathbf{v}}(\mathbf{u},\mathbf{v},\mathbf{p})\mathbf{v}=\mathbf{b}_{\mathbf{v}}\\ \mathbf{A}_{\mathbf{p}}(\mathbf{u},\mathbf{v},\mathbf{p})\mathbf{p}=\mathbf{b}_{\mathbf{p}}\end{gathered} \tag{14}$$

すなわち、ある物理量を計算するための方程式の係数行列に他の物理量が入り込んできているため、この問題は非線形問題となる。そのため、直接解を求めることができないため、反復法を用いる。また途中の解の改良において他の物理量はあくまで推定値でしかないため、その時点での各々の方程式を厳密に解く必要はない。一方、方程式の係数行列の生成は時間の掛かる作業である。よって、全体の一回の改良に各物理量は数回計算される。

性能評価は文献 [8] の第三章キャビティ流れの解法プログラムを用いた。キャビティ流れは移動壁により渦の流れをする。それを図 25に示す。解の改良は線領域をTMDA で解き、ガウスザイデル法を組み合わせたものを使用している。また、最終的に緩和計算を行う場合、不足緩和を用いている。

ここで、TINPARの出力そのままでは大規模な反復法の実験をするためには不完全なので、並列化および通信の最適化は自分で行ったものを使用した。なお、プロセッサの台数効果を見るため問題のサイズを100×100 に変えた。実行時間の結果を図 26に示す。比較の対象として、`for_slice` と 3 章で述べた終了連絡方式を収束判定としたものを用いた。まず、`for_slice` を用いた場合の収束までの反復回数を表 1に示す。これらの結果から、プロセッサ台数の増加に伴う収束効率の悪化は微量であることが確かめられた。実際、図 26に示すように, 解の改良のの高速化による全体の計算時間の短縮が、収束効率の悪化による計算時間の増分を上回り、for 文によるものと比べて実行時間が 10%程度速くなっている。

また、`for_slice` で用いている収束判定の総和演算は各プロセッサが所有する部分領域の総和演算を行い、最終的にプロセッサ間総和演算関数を用いて計算しているだけなので非常に簡単である。一方、for で理想状態の速さを保つために、収束判定は 3 章で述べた終了連絡方式で行われねばならない。これは並列化コンパイラでは生成できず、高度なテクニックが必要である。

これらを総合すると`for_slice` は簡単に記述でき、しかも高速化できる利点があるといえる。

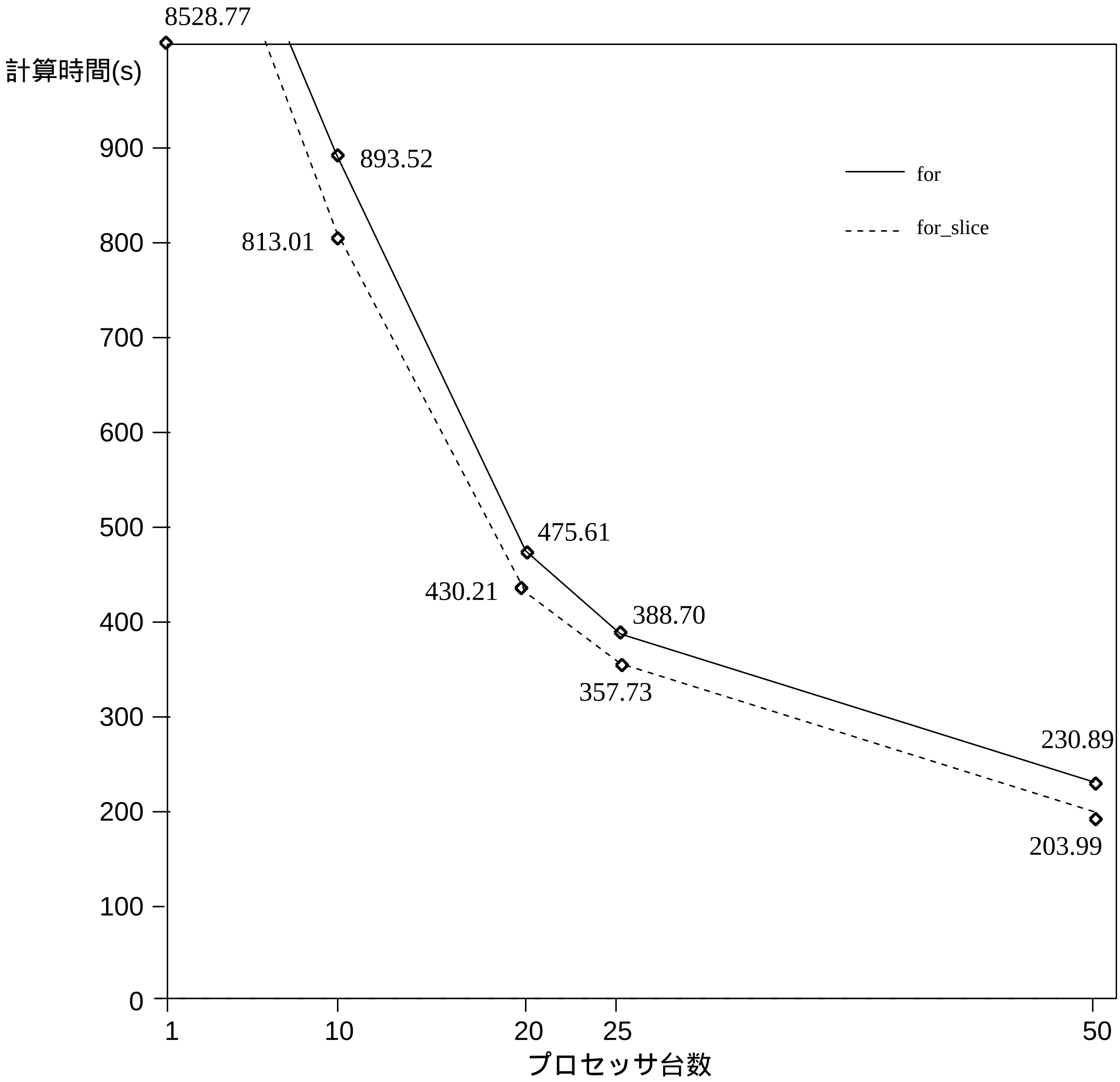


図 26: ナビエストークス方程式での for と for_slice の実行時間の比較

表 1: for_slice 対する収束の変化

プロセッサ数	1	5	10	20	25	50
収束回数	3028	3030	3035	3058	3094	3139
収束効率 (%)	100.0	99.9	99.8	99.0	97.9	96.5

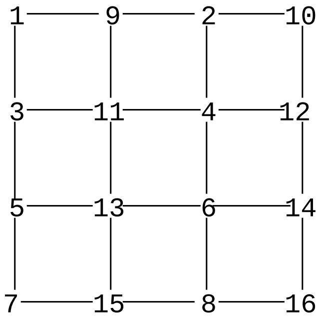


図 27: 並列処理向き順序付けの一例

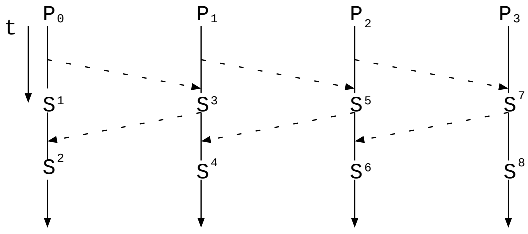


図 28: 並列処理向き順序付けを用いた処理の流れ

7　並列ループ構文の拡張

本章では`for_slice`に類似した拡張並列ループ構文の拡張の一つの方向として、計算順序の変更を指示する機能を提案する。

本来、反復法ではスーパーコンピュータでの高速実行のために、計算する格子点の順序を変更することをしばしば行ってきた [5]。その代表例が Red-Black 法あるいは Odd-Ever 法と呼ばれるものである。この方法ではベクトル処理し易いように計算順序が変えられている。

ここで、スカラプロセッサを用いた並列計算機向きの順序付けを図 27に示す。各格子点は解を改良するために部分領域に分け、違う部分領域に存在する格子点と連続した番号にならないように、なるべく離して順序付けを行っている。これを各プロセッサにうまくマッピングすると高速な処理が行えるはずである。ここで大まかな処理の流れを図 28で示しておく。ここで、S_iは格子列の改良を行っているものとする。S_3、S_5、S_7は計算に更新前の隣接データを使用するため、計算前に通信を行う。また、S_2、S_4、S_6は計算に更新後の隣接データを使用するため、S_3、S_5、S_7実行後通信を行っている。

この順序付けの効果を調べるために、6.2 節で用いたものと同じ問題を解くプログラムを作成した。プログラムの実行時間を図 29に、収束回数の変化を表 2にそれぞれ示している。

この問題に関しては、収束効率はプロセッサ台数が増えるにつれて良くなっている。その理由として、キャビティ流れの移動壁に垂直な方向でデータ分割していることが考えられる。移動壁は生成項であるため、そこからデータが生成されていく。それにより、各プロセッサの計算最後列は通常であると誤差の大きい値を使用せねばならないが、順序を変えると各プロセッサの計算最後列は誤差の小さくなっている値を使用することとなる。そのため、プロセッサ数を多くすると収束効率が良くなることが説明できる。また、一回の改良時間は`for_slice`によるものと同程度であるため、全体の速度として相当な並列化効果が得られている。

この方式では、`for_slice` によるものと同様に、プロセッサ内部の解の改良順序が逐次的になっており、プロセッサ内部に関しては通常の反復法で記述できる。したがって、プロセッサ間の依存関係による通信のみを変更すればよく、`for_slice` に類似した構文を導入すれば、簡単にこの並列化手法を利用できる。

8　おわりに

並列計算機では依存が存在するとプロセッサ間での通信が必要であるため、並列化効率があまり期待できない場合が考えられる。そこで、我々はプロセッサ間のデータ依存関係を緩和する並列ループ構文`for_slice` を提案した。

`for_slice` を用いた処理では計算結果がプロセッサ台数によって変化するため、直接法では用いることができない。しかし、反復法は途中の計算結果はどうでもよく、収束した時の最終結果のみが重要であるので、収束性が保証される範囲であれば`for_slice` を用いても良い。これは、緩和係数が不足緩和である時、適格であると考えられる。

また、`for_slice` を並列化コンパイラ TINPAR に実装してみた。この実装では、依存の緩和をループの外側前方に通信を移動することと一致していることに基づいている。

性能評価により`for_slice` の処理の速さは収束までの反復回数が少ない場合顕著である。また、ナビエストークス方程式を用いた評価では通常の for 文を並列化した場合の理想状態に対して総計算時間において十分な速さを示した。これは通常の for 文がプロセッサ間の依存によって生じる通信の間に通信と関係のない計算が終ってしまうため、待ち時間が生じているためだと考えられる。

そして、今後の課題で述べた通り、計算の順序付けを変えることにより、問題によっては通常の for 文より速く収束する場合もあり得る。それは生成項とデータ分割面との関係による。そのため、プロセッサごとに割り当てられた部分領域の生成項との関係と各領域のデータ交換タイミングに関して色々調べ、有効なものに対して逐次言語の拡張の形で実現することが重要であろう。

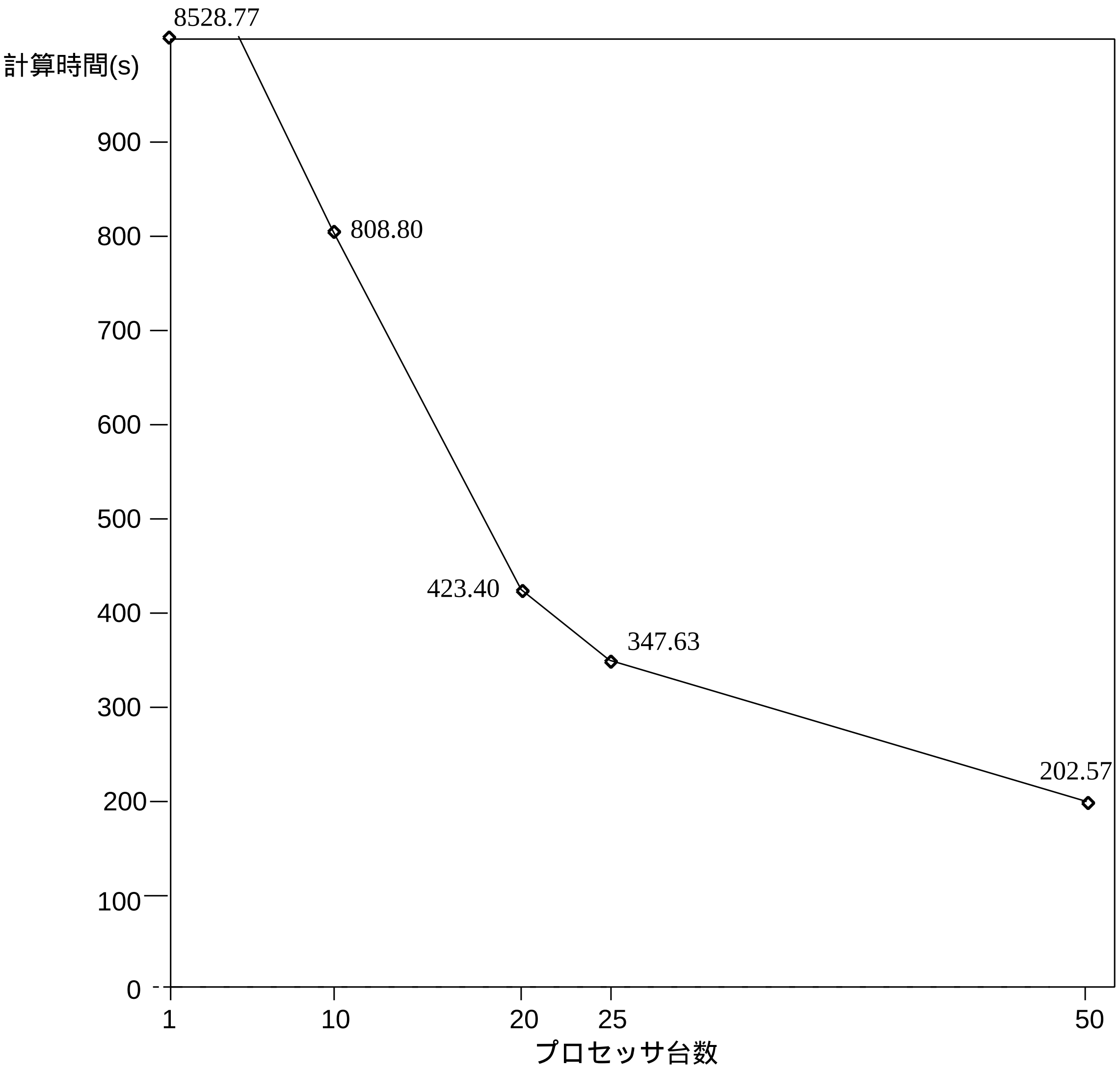


図 29: 並列用順序付けの実行時間

表 2: 並列順序付けに対する収束の変化

プロセッサ数	1	5	10	20	25	50
収束回数	3028	3019	3010	2996	2990	2968
収束効率 (%)	100.0	100.3	100.6	101.1	101.3	102.0

謝辞

本研究の機会を与えてくださった富田眞治教授に，深甚な謝意を表します。

また、貴重なご助言を頂いた中島浩助教授、森眞一郎助教授に深く感謝致します。

また, 並列計算機 AP1000 の実行環境を御提供頂きました (株) 富士通研究所に感謝致します。さらに，共同研究者である窪田 昌史氏をはじめとして、日頃御討論頂いた富田研究室の諸氏に感謝致します。

参考文献

[1] M. Wolfe: The Tiny Loop Restructuring Research Tool, Proc. of 1991 Int'l Conf. on Parallel Processing vol.II, pp.46–53, 1991.

[2] High Performance Fortran Forum: High Performance Fortran Language Specification Version 1.0, 1993.

[3] 三吉郁夫ほか:メッセージ交換型並列計算機のための並列化コンパイラ TINPAR , 並列処理シンポジウム JSPP'95,pp.51-58,1995.

[4] Hans Zima, Barbara Chapman:COMPILING FOR DISTRIBUTED-MEMORY SYSTEM,University of Vienna, 1994.

[5] I.S.Duff and G.A.Meurant:The effect of ordering on preconditioned conjugate gradient, BIT,29:635-657,1989.

[6] C.A.J. フレッチャー著, 澤見英男訳: コンピュータ流体力学, シュプリンガーフェアラーク東京株式会社,1993.

[7] 藤井孝藏:流体力学の数値計算法, 東大出版会,1994.

[8] 荒川忠一:数値流体力学, 東大出版会,1994.