Vol.2022-HPC-185 No.14
2022/7/28

Forループの並列化可能性の判定における穴埋め言語学習の応用

西村 佳[1,a)]　置田 真生[1,b)]　伊野 文彦[1,c)]

概要：
本報告は，深層学習による自動並列化の実現可能性調査を目的として，ループ文の並列化可能性を穴埋め言語学習（MLM）に基づいて用いて判定する手法を提案する．MLM を単純に適用した場合の課題として，プログラミング言語における変数名の多様性が，自然言語処理のために設計された MLM の学習を阻害する．提案手法は，この課題を改善するために，学習前に変数名をその役割に基づいて普遍化する．評価実験では，for ループを含む C プログラムを対象に，OpenMP プログラムを学習した穴埋め言語モデルを用いて各ループを並列化可能または不可能に分類した．実験の結果，提案手法の分類精度は，再現率が 37.5%と低い結果を示したが，適合率が 91.5%と既存の自動並列化コンパイラと比較して同等の高い結果を達成した．変数名の普遍化が適合率を改善する一例を示した．

1. はじめに

並列プログラミングの労力を軽減するために，自動並列化に関する研究が盛んである．自動並列化ツールは，並列実行可能なソースコードの領域（ループや文の集まり）を検出し，その領域に並列化を施す．既存の主なツール [1], [2], [3], [4] は，コード領域内のデータ依存関係を静的に解析し，その結果に基づいて並列化可能性を判定する．

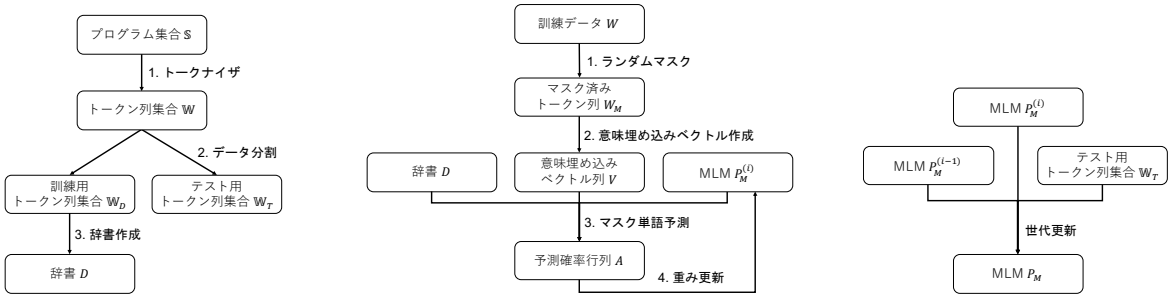
静的依存解析に基づく自動並列化は適用範囲に限界があり，遍く並列化を実現するためには人間による介助が必要である．既存ツールは，領域内に依存関係がないと保証できない場合，すなわち暗黙的な依存関係を含む可能性がある場合に，不正な計算結果を避けるためにその領域を並列化の対象から除外する．そのような領域に対しては，プログラマが指示文でコンパイラにヒントを与えるか，あるいはソースコードを書き換えて，領域内の依存関係を明示することで並列化が可能となる．

そこで本研究は，自動並列化の新たな試みとして，プログラムの自然さ [5] に着目する．近年，深層学習モデル Transformer [6] を用いたプログラム生成の実用化が進んでいる．GitHub Copilot [7] は，プログラマが途中まで記述した内容から続きのプログラムを補完する．また，TransCoder [8] は，異なるプログラミング言語間での変換を Transformer を用いて実現した．これらは，文章の特徴量を抽出するためのエンコーダとして，穴埋め言語学習モデル [9]（Masked language model；以降，MLM）を利用する．MLM は文章の自然さ（言語的正しさ）を学習し，ある単語の前後の単語を高精度で予測する．

Transformer に基づく言語処理モデルを用いることで，自動並列化を実現できる可能性がある．自動並列化は，逐次プログラムから並列プログラムへの変換に帰着する．あるいは，逐次プログラムを未完成の並列プログラムとみなして補完する問題に帰着できる．ただし，言語処理モデルがプログラムに内在する並列性を特徴量として抽出し，学習できる必要がある．

本報告は，言語処理モデルによる自動並列化の実現可能性調査を目的として，ループ文の並列化可能性を MLM に基づいて判定する手法を提案する．提案手法の対象は，C 言語で記述された OpenMP プログラムである．提案手法は，OpenMP プログラムを学習した MLM を用いて，プログラム内の各 for ループを並列化可能か不可能に分類する．すなわち，逐次プログラムの for ループについて，その直前に並列化指示文を挿入した場合の自然さが高ければ並列化可能とみなす．本報告では，同等の条件下で提案手法と静的依存解析に基づく既存手法を比較するために，for ループ内の依存関係が明示的であるプログラムを用いて判定精度を検証する．

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka Uniersity
[a)] ki-nismr@ist.osaka-u.ac.jp
[b)] okita@ist.osaka-u.ac.jp
[c)] ino@ist.osaka-u.ac.jp



(a) 訓練データ作成　　**(b)** MLM の訓練　　**(c)** MLM の世代更新

図 1: TransCoder における事前学習の概要

本研究の貢献は，プログラミング言語に対する MLM 学習を促進するために，訓練前に変数の命名規則を統一する点にある．自然言語における単語と比較すると，プログラムにおける変数名は普遍性が低く，同じ変数名（単語）が取りうる意味が無限に存在する．自然言語においては，一般に単語は複数の意味を持つが，意味集合は有限であり，全ての書き手にとって普遍である．しかし，プログラムにおいては，プログラマが変数を自由に命名でき，その意味はスコープごとに特殊である．例えば，変数 “i” がループ制御変数，配列の名前，純変数および関数名など様々な意味を持ちうる．この変数名の過度な多様性が，MLM によるプログラム文の学習を阻害する [10]．クローン検出を目的とした既存のプログラム解析 [11] は，変数名を普遍的な文字列（型名など）に一律に置換して言語処理モデルに与えるが，この手法では並列性解析に必要な変数間の依存関係が失われる．そこで提案手法は，変数の区別を保持しつつ，複数のソースコード間で変数名を普遍化する．

以降では，2 節で穴埋め言語学習に関する前提知識を示し，3 節で問題を定義する．4 節で提案手法について説明し，5 節で提案手法を評価する．6 節で関連研究を紹介し，7 節で本報告のまとめと今後の課題を示す．

2. 準備

穴埋め言語学習モデル（MLM）に基づくプログラミング言語処理の既存手法について，その概要を示し，用語を定義する．

2.1 穴埋め言語学習モデル MLM

MLM は，文，すなわち単語の並び $W = (w_1, w_2, \dots)$ に関する確率分布を評価するための，ニューラルネットワーク（以降，NN）に基づく双方向言語モデルである．学習では，正しい単語の並び W に関して次の式 (1) の目的関数 $P_M(W)$ を最大化するように NN を訓練する．

$$P_M(W) = \prod_{i=1}^{|W|} P_M(w_i | W^{\setminus i}) \qquad (1)$$

ここで，$P_M(w_i|W^{\setminus i})$ は，W 内の i 番目の単語 w_i をマスクで隠した文 $W^{\setminus i}$ に対して，そのマスクを元の単語 w_i で置換した場合の尤度を表す条件付き確率である．

MLM の特長は，以前の言語モデル [12], [13], [14] では不可能だった深い双方向的の学習を可能にする点である．式 (1) で示すように，文頭および文末からの全ての情報（文脈）を利用して訓練することで，高い予測精度を実現する．

MLM を応用したタスクの 1 つに，単語の穴埋め問題がある．1 つの単語をマスクした文 $W^{\setminus i}$ に対して，候補単語集合 D が与えられ，マスクを $w \in D$ で置換した場合に単語の並びが最も自然となる単語の探索問題を $F_b(W^{\setminus i}, D)$ と表す．この問題は，MLM を用いると次の式 (2) のように帰着できる．

$$F_b(W^{\setminus i}, D) = \arg\max_{w \in D} P_M(w | W^{\setminus i}) \qquad (2)$$

2.2 MLM を用いたプログラミング言語の事前学習

TransCoder [8] は，プログラミング言語に特化した MLM に基づくトランスコンパイラである．入力プログラムを，機能的に等価な別言語のプログラムに翻訳する．TransCoder は，入力プログラム S のソースコードを 1 つの文と見なして MLM を適用する．すなわち，ソースコード中のトークンを自然言語における単語と見なし，プログラム S をトークンの並び $W = (w_1, w_2, \dots)$ に変換して MLM に与える．

TransCoder における MLM の事前学習は，訓練データの作成，MLM の訓練，および MLM の世代更新の 3 段階からなる．それぞれの概略を**図 1** に示す．事前に，プログラムをトークン列に変換し，訓練データ集合 $\mathbb{W}_D$ を作成する（**図 1a**）．まず，全ての $W \in \mathbb{W}_D$ に対して MLM の訓練（**図 1b**）を実施し，1 世代の訓練結果 $P_M^{(i)}$ を得る．$P_M^{(i)}$ は第 i 世代の訓練結果を表す．次に，前世代の結果 $P_M^{(i-1)}$ と比較して，世代を更新する（**図 1c**）．世代更新を一定回数繰り返し，事前学習を終了する．

訓練データの作成は，与えられたプログラム集合 $\mathbb{S} = \{S_1, S_2, \dots\}$ から，データセット $(\mathbb{W}_D, \mathbb{W}_T, D)$ を生成する．(1) まず，Clang[15] が提供する構文解析機能を利用して，プログラム $S_i \in S$ をトークン列 W_i に変換する．(2) 次に，得られたトークン列集合 $\mathbb{W} = \{W_1, W_2, \dots\}$ を，

訓練データ集合 $\mathbb{W}_D$ およびテストデータ集合 $\mathbb{W}_T$ に分割する．ここで，$\mathbb{W} = \mathbb{W}_D \cup \mathbb{W}_T$ かつ $\mathbb{W}_D \cap \mathbb{W}_T = \emptyset$ である．(3) 最後に，$\mathbb{W}_D$ に含まれるトークンのうち出現頻度が高い m 個を選択し，辞書（頻出トークン集合）D を得る．m は指定された整数とする．

MLM の訓練は，1 つの訓練データ $W \in \mathbb{W}_D$ および辞書 D を入力として，MLM $P_M^{(i)}$ の重み（モデルパラメータ）を更新する．(1) まず，トークン列 W の 15%のトークンをランダムに選択し，マスク候補を決定する．候補に含まれるトークンの 80%をランダムに選び，マスクする．すわなち，特殊トークン “`[MASK]`” に置換する．マスクしたトークン w_j の位置 j の集合を I とする．さらに，残りの半数（10%）を辞書 D からランダムに選んだ単語に置換する．最後の 10%は元の単語のまま置換しない．得られたマスク済みトークン列を W_M と表す．(2) 次に，$W_M = (w_1, w_2, \dots)$ の各トークン w_i から，位置情報 i を付与した意味埋め込みベクトル v_i を生成し，ベクトル列 $V = (v_1, v_2, \dots)$ を得る．(3) (V, D) を MLM $P_M^{(i)}$ に与え，予測確率行列 A を得る．A は，$A_{k,j} = P_M^{(i)}(d_j | W_M^{\setminus k})$ （$k \in C$ かつ $d_j \in D$）を要素とする $|C| \times |D|$ 行列である．(4) 最後に，A を真の予測確率行列 A_t と比較した損失に基づいて，誤差逆伝播法を利用して $P_M^{(i)}$ の重みを更新する．

世代更新は，テストデータ集合 $\mathbb{W}_T$ を用いて，訓練した MLM $P_M^{(i)}$ の予測精度を検証する．$P_M^{(i-1)}$ と比較して $P_M^{(i)}$ の予測精度が向上していれば，最終的な予測モデル P_M の重みを，$P_M^{(i)}$ の重みで更新する．

3. 問題の定義

本報告は，C 言語のプログラムに含まれる for ループについて，並列化可能性の判定問題を扱う．具体的には，式 (3) に示す，入力プログラム S および S に含まれる，ある for ループ l に対する 2 値分類問題 $C(l, S)$ を解く分類器を作成する．

$$C(l,S) = \begin{cases} \text{true}, & \text{if 指示文による並列化が可能} \\ \text{false}, & \text{otherwise} \end{cases} \quad (3)$$

ここで，指示文による並列化が可能とは，for ループに OpenMP のループ並列化指示文（`#pragma omp parallel for`）を付与するだけで並列実行が可能であることを意味する．つまり，ループ並列化指示文を付与して並列に実行したとき，正しい計算結果が得られるならば真（true）と分類し，そうでなければ偽（false）と分類する．

問題を簡単化するために，本報告では以下の制約をおく．（1）一般的なライブラリ関数を除いて，プログラムから呼び出す関数の定義は全てソースコードに記述が存在する．（2）指示文以外のソースコードの変形を伴う並列化は対象外とする．（3）ループ並列化指示文に続けて指示節（`private`，`reduction` など）が必要な場合も，並列化可能ループとみなす．

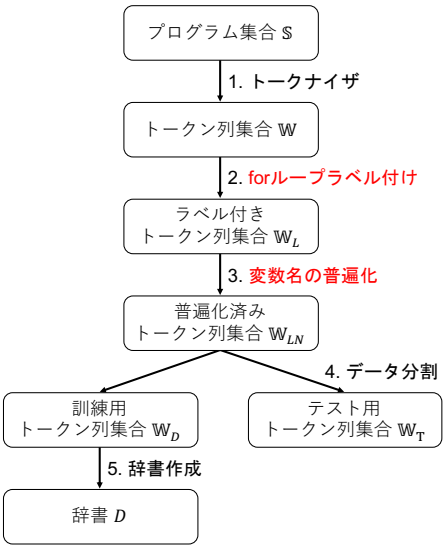


図 2: 提案手法における訓練データの整形

4. 提案手法

提案手法は，OpenMP プログラムを学習した MLM P_M を用いて，ループに対する並列化可能性の分類器 $C(l, S)$（式 (3)）を作成する．具体的には，逐次プログラム S のトークン列 W における for ループ l（予約語トークン `for`）の直前位置 i にマスクを挿入し，P_M を用いて尤度の高いトークンの集合 $F_b(W^{\setminus i}, D)$ を探索する．$F_b(W^{\setminus i}, D)$ に並列化指示文 δ が含まれていれば，並列化可能と判定する．すなわち，次式 (4) の通り，$C(l, S)$ を単語穴埋め問題 $F_b(W^{\setminus i}, D)$ に帰結する．

$$C(l,S) = \begin{cases} \text{true}, & \text{if } \delta \in F_b(W^{\setminus i}, D) \\ \text{false}, & \text{otherwise} \end{cases} \quad (4)$$

4.1 節で後述するように，並列化指示文 δ を 1 語のトークン “`#parallel`” に置換するので，1 単語の穴埋め問題として解くことが可能である．

提案手法は，MLM 学習の促進を目的として，入力トークン列 W に次の 2 つの改変を施す．

- ループのラベル付け
- 変数名の普遍化

ループのラベル付けは，並列化ループと非並列化ループについて記述の対称性を得るための変換である．また，変数の普遍化は，複数のプログラム間で類似する役割（意味）をもつ変数の表現（変数名）を統一するための変換である．**図 2** に示すように，プログラムをトークン列に変換したのち，全てのトークン列 $W \in \mathbb{W}$ に対してそれぞれ，(3) ループのラベル付けおよび (4) 変数名の普遍化を適用する．

提案手法の実装は，TransCoder のエンコーダ部を拡張して実装する．ループのラベル付けおよび変数名の普遍化の実装に必要な文法情報は，トークン列の変換と同様に，

```
1  int main(){
2    #pragma omp parallel for
3    for (int i = 0;i < 100;i++){
4      // parallel region
5    }
6
7    for (int i = 0;i < 100;i++){
8      // sequential region
9    }
10 }
```

(a) 元のプログラム

```
1  int main(){
2    #parallel
3    for (int i = 0;i < 100;i++){
4      // parallel region
5    }
6    #empty
7    for (int i = 0;i < 100;i++){
8      // sequential region
9    }
10 }
```

(b) ラベル付け適用後

図 3: for ループラベル付けの適用例

Clang が提供する抽象構文木から得られる情報を利用する．

4.1 for ループラベル付け

for ループにラベル付けするモジュール（以降，ラベル付けモジュール）は，OpenMP プログラム内の各 for ループを対象に，非並列化ループについては直前にラベルトークン “`#empty`” を追加し，並列化ループについては並列化指示文を簡略化したラベルトークン “`#parallel`” に置換する．例として，**図 3a** のプログラムに適用した結果を**図 3b** に示す．実際には，プログラム S を変換したトークン列 W に対してラベル付けを適用する．すなわち，ラベル付けモジュールが，W を入力として受け取り，ラベルを追加および置換したトークン列 W_L を出力する．

まず，非並列化ループを並列化ループと対称的に扱うため，非並列化ループの直前にラベルを追加する．OpenMP プログラムにおいて，並列化ループの記述はその直前に並列化指示文が存在し，明示的である．一方で，非並列化ループの記述は直前に指示文が存在せず，暗黙的である．この明示性の非対称が，MLM の適用を複雑化する．これを回避するため，MLM の学習前に，非並列化ループにも明示的なラベル “`#empty`” を付与する．

次に，並列化指示文のトークン数を一定にするため，並列化指示文を単純なラベルに置換する．“`#pragma omp parallel for`” で始まるループ並列化指示文に対して，それ以降の指示節の内容に関わらず，全体を 1 語のトークン “`#parallel`” に置換する．ただし，この置換が原因で，提案手法は指示文に続く指示節の違いを学習できない．

4.2 変数名の普遍化

変数名を普遍化するモジュール（以降，普遍化モジュール）は，複数のプログラム間で類似する役割を持つ変数の名前を統一するために，トークン列 W_L に含まれる変数トークンを一定の規則に従って置換する．普遍化モジュールは，ラベル付けされたトークン列 W_L を入力として受け取り，普遍化を施したトークン列 W_{LN} を出力する．ただし，記述を簡単にするため，以降の例はソースコード形式で記載する．普遍化を適用する前のトークン列をソースコード形式で記載した例を**図 4** に示す．

アルゴリズム 1 ループ制御変数の単調な普遍化

Input:
$W_{LN} = (w_1, w_2, \ldots, w_m)$
Output:
W_{LN}
$T = \emptyset$
for $i = 0 \cdots m$ **do**
  if w_i が for ループの初期化式で代入される変数である **then**
    $T \leftarrow T + \{w_i\}$
  end if
end for
$n = 0$
for all $t \in T$ **do**
  $n \leftarrow n + 1$
  for $i = 0 \cdots m$ **do**
    if $t = w_i$ **then**
      W_{LN} 内の w_i を “`#F`n” で置換
    end if
  end for
end for

提案手法は，並列性の判定における重要性の観点から，ループ制御変数および配列変数の 2 点に着目して変数の役割を分類する．さらに，ループ制御変数の分類について，多重ループを区別しない単調な分類と，入れ子の深さを基準とした分類に細分化する．なお，ループ制御変数の普遍化と配列変数の普遍化は，組み合わせて同時に適用することが可能である．

4.2.1 ループ制御変数の単調な普遍化

プログラム内の for ループにおけるループの制御変数について，その名前が一定の命名規則に従うよう変換する．一般に，ループ並列化の対象となる for ループは配列に対する処理が中心である．ループ並列化を阻害するループ伝搬依存は，ループ制御変数の関数を添字とした配列アクセスによって生じる．そこで，MLM はループ制御変数を識別し，複数のプログラム間でループ制御変数が共通して持

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main ( int argc , char * argv [ ] ) {
4    int i , j ;
5    int n = 100 , m = 100 ;
6    double b [ n ] [ m ] ;
7    #empty
8    for ( i = 1 ;
9    i < n ;
10   i ++ ) {
11     #parallel
12     for ( j = 0 ;
13     j < m ;
14     j ++ ) {
15       b [ i ] [ j ] = b [ i - 1 ] [ j ] ;
16     }
17   }
18   #empty
19   for ( j = 1 ;
20   j < n ;
21   j ++ ) {
22     printf ( " b [ % d ] [ % d ] = % f \n " , j , j , b
           [ j ] [ j ] ) ;
23   }
24   return 0 ;
25 }
```

図 4: 変数名の普遍化を適用する前のプログラム例

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main ( int argc , char * argv [ ] ) {
4    int #F1  , #F2 ;
5    int n = 100 , m = 100 ;
6    double b [ n ] [ m ] ;
7    #empty
8    for ( #F1 = 1 ;
9    #F1 < n ;
10   #F1 ++ ) {
11     #parallel
12     for ( #F2 = 0 ;
13     #F2 < m ;
14     #F2 ++ ) {
15       b [ #F1 ] [ #F2 ] = b [ #F1 - 1 ] [ #F2 ] ;
16     }
17   }
18   #empty
19   for ( #F2 = 1 ;
20   #F2 < n ;
21   #F2 ++ ) {
22     printf ( " b [ % d ] [ % d ] = % f \n " , #F2 , #F2
           , b [ #F2 ] [ #F2 ] ) ;
23   }
24   return 0 ;
25 }
```

図 5: ループ制御変数の単調な普遍化の適用例

つ役割を学習できる必要がある．MLM は名前に基づいてトークンを識別し意味を埋め込む．したがって，複数のプログラム間でループ制御変数が同一の名前を持つことが望ましい．ただし，同一プログラム内で異なる制御変数を区別できる必要がある．

この手法は，同一の接頭句 `#F` と可変の接尾句（自然数 n）を連結した名前でループ制御変数名を置換する．接尾句 n は，同一プログラム内の登場順に割り当てる．変換の手順をアルゴリズム 1 に示し，適用例を図 **5** に示す．図の赤字は置換されたループ制御変数を表す．8 行目および 12 行目のループ制御変数 i および j を，それぞれ `#F1` および `#F2` に置換する．

4.2.2 入れ子の深さに基づくループ制御変数の普遍化

プログラム内の for ループにおける制御変数について，ループの入れ子の深さを基準とした一定の規則に従うよう，変数名を変換する．4.2.1 節と同様にループ制御変数に注目し，さらに入れ子の深さを考慮する理由は，最外ループと内側のループでは並列化の対象となる傾向に差があると仮定したためである．MLM が入れ子の深さに強く依存した意味を制御変数に埋め込めるよう，制御変数の名前を深さを基準に複数プログラム間で統一する．

この手法は，同一の接頭句 `#F` と入れ子の深さを表す接尾句（自然数 d）を連結した名前でループ制御変数名を置換する．最外ループに対して $d = 1$ とし，1 つ内側ループに深くなるたびに d を 1 増やす．主な変換手順はアルゴリズム 1 と同様なので，詳細を省略する．異なる点は，3〜5 行目の判定時に入れ子の深さ d を記録し，12 行目の置換時に接尾句として用いる点のみである．適用例を図 5 に示す．図の赤字は置換されたループ制御変数を表す．8〜17 行目の 2 重 for ループ内では，変数 `i` を `#F1` に，変数 `j` を `#F2` に置換する．また，19〜23 行目の 1 重 for ループでは，変数 `j` を `#F1` に置換する．

4.2.3 配列変数と純粋変数の普遍化

ループ制御変数を除き，原則として全ての変数名を普遍化する．ただし，配列変数とその他の変数（純粋変数）を区別し，それぞれ異なる規則に従って変数名を置換する．配列変数を区別する理由は，4.2.1 節で記述したように，配列アクセスに関するループ伝搬依存がループの並列化可能性を決める要因になるためである．MLM が配列変数に他の変数と異なる意味を埋め込めるよう，配列変数に固有の命名規則を与える．

この手法は，配列変数名を接頭句 `#v` と接尾句（自然数 n）を連結した名前に置換し，純粋変数名を接頭句 `#a` と接尾句（自然数 m）を連結した名前に置換する．接尾句 n および m は，同一プログラム内の登場順に割り当てる．配列変数の識別は，配列のポインタ表現に対応するため，宣言ではなく参照を基準に決定する．具体的には，ある変数 `x` が添字演算子（`[ ]`）の被演算子となる場合，`x` を配列変

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main ( int argc , char * argv [ ] ) {
4    int i , j ;
5    int n = 100 , m = 100 ;
6    double b [ n ] [ m ] ;
7    #empty
8    for ( #F1 = 1 ;
9    #F1 < n ;
10   #F1 ++ ) {
11           #parallel
12     for ( #F2 = 0 ;
13     #F2 < m ;
14     #F2 ++ ) {
15       b [ #F1 ] [ #F2 ] = b [ #F1 - 1 ] [ #F2 ] ;
16     }
17   }
18   #empty
19   for ( #F1 = 1 ;
20   #F1 < n ;
21   #F1 ++ ) {
22     printf ( " b [ % d ] [ % d ] = % f \n " , #F1 , #F1
           , b [ #F1 ] [ #F1 ] ) ;
23   }
24   return 0 ;
25 }
```

図 6: 入れ子の深さに基づくループ制御変数の普遍化の適用例

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main ( int #v0  , char * #a0 [ ] ) {
4    int i , j ;
5    int #v1 = 100 , #v2 = 100 ;
6    double #a1 [ #v1 ] [ #v2 ] ;
7    #empty
8    for ( i = 1 ;
9    i < #v1 ;
10   i ++ ) {
11     #parallel
12     for ( j = 0 ;
13     j < #v2 ;
14     j ++ ) {
15       #a1 [ i ] [ j ] = #a1 [ i - 1 ] [ j ] ;
16     }
17   }
18   #empty
19   for ( j = 1 ;
20   j < #v1 ;
21   j ++ ) {
22     printf ( " b [ % d ] [ % d ] = % f \n " , j , j ,
           #a1 [ j ] [ j ] ) ;
23   }
24   return 0 ;
25 }
```

図 7: 配列変数と純粋変数の普遍化の適用例

数と識別する．適用例を図 **7** に示す．

5. 評価

次の 2 つの観点から，提案手法を評価する．

- 並列化可能性の判定精度
- 変数名の普遍化の有用性

提案手法を用いて MLM を訓練し，その MLM を用いて逐次 C プログラムに含まれる for ループの並列化可能性を判定する．

まず，判定精度を評価するため，提案手法の判定結果を既存の自動並列化コンパイラの判定結果と比較する．比較対象として，AutoPar [1] および Cetus [2] を用いた．これらのツールは，プログラムの静的依存解析に基づいて for ループの並列化可能性を判定する．

次に，変数名の普遍化の有用性を評価するため，複数の普遍化を段階的に組み合わせて判定精度を比較する．組み合わせを**表 1** に示す．ループ制御変数に関する 2 種類の普遍化は排他的であり，同時に適用できない．

実験で用いた MLM のアーキテクチャは，Vaswani ら [6] と同じ構成である．6 層の 8 つの Attention ヘッドを使用し，モデルの次元数を 1024 とする．また，学習率を 10^{-5} とし，学習のエポック数を 20 とする．実装は TransCoder を拡張した．

実験環境は，2 台のホスト PC に接続された計 4 台の GPU を用いた．GPU は NVIDIA GeForce RTX 2080（VRAM 容量 8 GB）である．利用した CUDA および cuDNN のバージョンはそれぞれ 11.0 および 8.0.5 である．また，MLM の実行に必要な Python および PyTorch は，それぞれバージョン 3.6.9 および 1.7.1 を用いた．

表 1: 変数名の普遍化の組み合わせ

	ループ制御変数	配列変数と純粋変数
普遍化なし	-	-
手法 1	単調	-
手法 2	単調	適用
手法 3	入れ子の深さ	-
手法 4	入れ子の深さ	適用

表内の - は普遍化しないことを表す．

5.1 データセット

MLM の学習に用いる訓練データには，大量のプログラムが必要となるため，AutoPar によって自動並列化された OpenMP プログラムを用いた．AutoPar に与える逐次プログラムは，AtCoder [16] に提出された C99 準拠のプログラムを使用した．ABC，ARC，および AGC（それぞれ，初心者，中級者，および上級者向け）コンテストの問題に正解したプログラム群から，ランダムに 8,254 個のプログラムを選択した．行数の総計は 738,400 行である．AtCoder に提出されたプログラムの特徴として，標準ライブラリ以

外のユーザ定義ライブラリの呼び出しは存在しない．

判定精度の評価に用いる評価データには，自動並列化ツールのためのベンチマーク DataRaceBench [17], [18] を使用した．DataRaceBench に含まれる 181 個のプログラムのうち，暗黙的な依存関係を含まない for ループで構成されたプログラム 98 個（総行数 8,319）を利用した．評価データに含まれる for ループの集合 L の各要素に対して，並列プログラマが判断した並列化可能（`true`）あるいは不可能（`false`）のラベルが付与されている．この情報 $\mathbb{L}_t : L \to \{\texttt{true}, \texttt{false}\}$ を正解ラベルと呼ぶ．

評価データは，訓練データと同様にトークン列に整形する．具体的には，図 2 に示す (1) トークナイザ，(2)for ループラベル付け，および (3) 変数名の普遍化を各プログラムに施す．ただし，(2) において，並列化指示文ではなく正解ラベル $\mathbb{L}_t$ に基づいて for ループにラベルを付与する．

5.2 実験結果

5.1 節の訓練データを用いて訓練した MLM P_M および訓練データの辞書 D を用いて，評価データに対する並列化可能性を判定した．具体的には，評価データに含まれる各 for ループ l について，その直前のラベルトークンをマスクし，式 (4) の通り $C(l, S)$ を判定した．なお，20 エポックの訓練に要した時間は約 16 時間であった．

提案手法の判定結果と正解ラベル $\mathbb{L}_t$ を比較し，適合率，再現率，正解率および F 値を算出した．各手法を用いた結果の比較を**図 8** に，具体的な評価指標の値を**表 2** に示す．T_P，T_N，F_P，および F_N はそれぞれ真陽性，真陰性，偽陽性，および偽陰性を表す．なお，それぞれの手法において，F 値が最も高いエポックの結果を記載した．

普遍化なしを除く全ての提案手法において，適合率が約 90%を達成し，既存手法と比較して同等程度の結果を示した．普遍化した手法 1〜4 は，普遍化なしと比較して，適合率が約 20 ポイント高い結果を示した．

全ての提案手法において，適合率が 40%未満しか達成できず，既存手法と比較して約 50 ポイント低い結果を示した．特に手法 1 および 2 は，他の提案手法と比較して，約 17 ポイント低い結果を示した．

全ての提案手法において，F 値が約 40%しか達成できず，既存手法と比較して約 50 ポイント低い結果を示した．提案手法の中では，普遍化なしと手法 4 が高い結果を示した．

5.3 考察

既存の依存解析に基づく自動並列化ツールは，全ての評価指標において提案手法よりも高い結果を示した．この理由は，今回の評価に用いたベンチマークは for ループの依存性が明示的なものであり，ループにおける各イテレーション間の依存関係を正確に解析することができたためである．

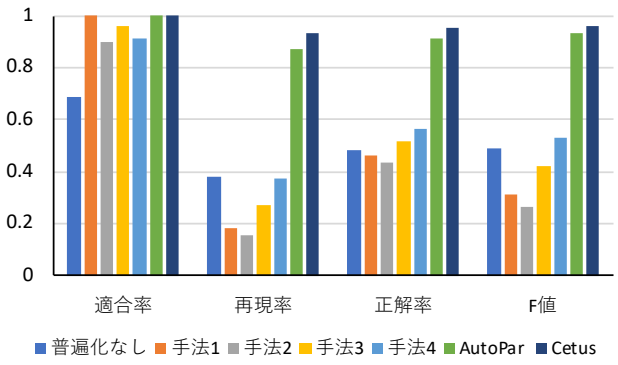


図 8: 既存の自動並列化ツールとの判定精度の比較

for ループ制御変数を普遍化したプログラムで訓練した MLM は，一定の類型を持つ並列化可能 for ループに特化した判定ができる．for ループ制御変数を普遍化した 4 つの手法全てにおいて，普遍化なしと比較して適合率が向上したことから，for ループ制御変数を普遍化することが適合率の向上に重要である．提案手法が普遍化なしと比較して再現率が低いことについて，普遍化なしは，適合率が約 70%と低い結果を示すことから，普遍化なしのプログラムで訓練した MLM は，ある程度大雑把に並列化可能を判定する．一方で，for ループ制御変数を普遍化する手法は，繊細に一定の類型を持つ並列化可能 for ループを判定する．

さらに，入れ子に基づく for ループ制御変数の普遍化により，for ループ制御変数の単調な普遍化と比べて，より多くの類型に特化することができる．これは，手法 1 および 3 と手法 2 および 4 を比較し，どちらも再現率が向上していることから判断できる．

提案手法の中では，F 値の高さおよび自動並列化の観点から手法 4 が最も優れた手法である．F 値において，手法 4 および普遍化なしが高い結果を示した．自動並列化の観点では，依存関係があるループを並列化するとプログラムの破損に繋がるため，適合率の高さが重要となる．普遍化なしは，手法 4 と比べて，適合率が低く再現率が高い．したがって，自動並列化の観点から普遍化なしより手法 4 の方が優れている．

6. 関連研究

Beau ら [19] は，BERT の MLM に基づくエンコーダと LSTM に基づくデコーダの seq2seq モデルを用いて自然言語からソースコードへの翻訳する手法を提案した．BERT のエンコーダにプログラムを入力する際，プログラミング言語における変数名の多様性によって，辞書集合の要素が無限になるという問題が生じる．Beau らは，この問題を低減するために，配列変数および純粋変数を “lst1”，“lst2” および “var1”，“var2” のように普遍化する．本報告では，単語 w_i の意味が普遍的ではなく，さまざまな意味を持ちうることを問題とする．そしてこの問題を低減するために，

表 2: 判定精度に関する実験結果の詳細

手法名	適合率	再現率	正解率	F 値	T_P	T_N	F_P	F_N	Total
普遍化なし	0.6875	0.381502	0.481060	0.490706	66	61	30	107	264
手法 1	1.0	0.184971	0.465909	0.312195	32	91	0	141	264
手法 2	0.9	0.156069	0.435606	0.266009	27	88	3	146	264
手法 3	0.959183	0.271676	0.515151	0.423423	47	89	2	126	264
手法 4	0.915492	0.375722	0.568181	0.532786	65	85	6	108	264
AutoPar	1.0	0.872832	0.916667	0.932099	151	91	0	22	264
Cetus	1.0	0.930636	0.954545	0.964072	161	91	0	12	264

役割に基づいて，新たに for ループ制御変数を他の変数と区別して普遍化する．

Roy ら [11] は，ソースコード置換とテキスト比較によるコードクローン検出手法を提案した．クローンされたコード片の多くは，変数名に軽微な修正が加えられることが多い．したがって既存のコードクローンは，識別子を “id” に変換する．この手法では全ての識別子を “id” に変換すると偽陽性が高まる問題を解決するために，特に変数名に変更が多いコード片に含まれる識別子を “id” に変換してコードクローンを検出する．本研究では，変数同士の関係性を訓練した MLM を作成するために，コードクローン検出のように識別子を一律に普遍化せず，for ループ制御変数，配列変数，純粋変数を区別して普遍化する．

7. まとめと今後の課題

本報告では，言語処理モデルによる自動並列化の実現可能性調査を目的として，for ループの並列化可能性を MLM に基づいて判定する手法を提案した．提案手法は，MLM の訓練を促進するために，事前に変数名を普遍化することでプログラミング言語における変数名の多様性を軽減する．普遍化は，（1）ループ制御変数の単調な普遍化，（2）入れ子の深さに基づくループ制御変数の普遍化，（3）配列変数と純粋変数の普遍化の 3 種類を組み合わせて適用した．実験の結果，提案手法は，既存の依存解析に基づく自動並列化ツールと比較して，適合率が同等程度，再現率が約 50 ポイント低く，正解率が約 35 ポイント低く，F1 値が約 40 ポイント低い結果を示した．また，for ループ制御変数を普遍化することで普遍化しない場合と比較して，適合率が約 20 ポイント高い結果を示した．for ループ制御変数の入れ子の深さに基づく普遍化により，再現率が約 17 ポイント高い結果を示した．F 値から，提案手法の中では，普遍化（2）と普遍化（3）を組み合わせた手法が最も有効であった．

提案手法は，高い適合率で並列化可能 for ループを検出できる一例を示した．したがって，穴埋め言語学習は，一定の類型を持つ並列化可能箇所を発見する能力を持ち，自動並列化を実現できる可能性がある．

今後の課題として，プログラムに対する穴埋め言語モデルを説明可能にすること，および 3 節に示す制約お緩和が挙げられる．

謝辞 本研究の一部は，JSPS 科研費 JP20K21794 および JP22K11972 の補助による．

参考文献

[1] Liao, C., Quinlan, D. J., Willcock, J. and Panas, T.: Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions, *Journal of Parallel Programming*, Vol. 38, No. 5-6, pp. 361–378 (2010).

[2] Bae, H., Mustafa, D., Lee, J., Aurangzeb, Lin, H., Dave, C., Eigenmann, R. and Midkiff, S. P.: The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation, *Journal of Parallel Programming*, Vol. 41, No. 6, pp. 753–767 (2013).

[3] Mendonca, G. S. D., Guimarães, B. C. F., Alves, P., Pereira, M. M., Araujo, G. and Pereira, F. M. Q.: DawnCC: Automatic Annotation for Data Parallelism and Offloading, *Journal of ACM Transactions on Architecture and Code Optimization*, Vol. 14, No. 2, pp. 13:1–13:25 (2017).

[4] Bondhugula, U., Hartono, A., Ramanujam, J. and Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer, *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, Tucson, AZ, USA, pp. 101–113 (2008).

[5] Hindle, A., Barr, E. T., Gabel, M., Su, Z. and Devanbu, P.: On the Naturalness of Software, *Communications of the ACM*, Vol. 59, No. 5, pp. 122–131 (2016).

[6] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. and Polosukhin, I.: Attention Is All You Need, *In Proceedings of the International Conference on Advances in Neural Information Processing Systems (NeurIPS 2017)*, Red Hook, NY, USA, pp. 6000–6010 (2017).

[7] GitHub, Inc.: GitHub Copilot, `https://github.com/features/copilot/` (accessed June 27, 2022).

[8] Roziere, B., Lachaux, M.-A., Chanussot, L. and Lample, G.: Unsupervised Translation of Programming Languages, *In Proceedings of the International Conference on Advances in Neural Information Processing Systems (NeurIPS 2020)*, Vol. 33, pp. 20601–20611 (2020).

[9] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *In Proceedings of the Annual Conference on the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL 2019)*, Minneapolis, Minnesota, pp. 4171–4186 (2019).

[10] Wang, R., Zhang, H., Lu, G., Lyu, L. and Lyu, C.: Fret: Functional Reinforced Transformer With BERT for Code Summarization, *Journal of IEEE Access*, Vol. 8, pp. 135591–135604 (2020).
[11] Roy, C. K. and Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, *In Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC 2008)*, NW Washington, DCUnited States, pp. 172–181 (2008).
[12] Radford, A. and Sutskever, I.: Improving Language Understanding by Generative Pre-Training, Technical report, OpenAI (2018).
[13] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K. and Zettlemoyer, L.: Deep Contextualized Word Representations, *In Proceedings of the Annual Conference on the North American Chapter of the Association for Computational Linguistics (NAACL 2018)*, New Orleans, Louisiana, pp. 2227–2237 (2018).
[14] Melamud, O., Goldberger, J. and Dagan, I.: context2vec: Learning Generic Context Embedding with Bidirectional LSTM, *In Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning (CoNLL 2016)*, Berlin, Germany, pp. 51–61 (2016).
[15] LLVM Developer Group: Clang: a C language family frontend for LLVM, `https://clang.llvm.org` (accessed June 27, 2022).
[16] AtCoder Inc.: AtCoder, `https://atcoder.jp` (accessed June 27, 2022).
[17] Liao, C., Lin, P., Asplund, J., Schordan, M. and Karlin, I.: DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools, *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC17)*, New York, NY, USA, pp. 11:1–11:14 (2017).
[18] Lin, P. and Liao, C.: High-Precision Evaluation of Both Static and Dynamic Tools using DataRaceBench, *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC21)*, New York, NY, USA, pp. 1–8 (2021).
[19] Beau, N. and Crabbé, B.: The impact of lexical and grammatical processing on generating code from natural language, ArXiv (2022). abs/2202.13972.