Vol.2019-ARC-235 No.8
Vol.2019-SLDM-187 No.8
Vol.2019-EMB-50 No.8
2019/3/17

レガシコードを含むモデルベース開発における並列化手法

黒柳 彰宏[1,a)]　金森 公洋[1]　枝廣 正人[1]

概要：組込みシステムの大規模・複雑化によりモデルベース開発やマルチコアプロセッサの導入が進んでいる．しかし，C 言語で記述された既存資産をそのままモデルベース開発に利用することは困難である．そこで，本研究ではレガシコードを含むモデルベース開発において並列化を行う手法を提案する．C コードの関数等をモデルのブロックと考えることで並列化を行う．行列計算を行うプログラムを用いた評価の結果，2 コアでおよそ 1.68 倍の高速化を達成した．

Parallelization method in model-based development with legacy codes

1. はじめに

近年，組込み業界では，ソフトウェアの大規模・複雑化に対応するため，モデルベース開発の導入が進められており，開発期間やコストの削減，ソフトウェアの品質向上が図られている．しかし，大規模化したシステムのモデルを一から作成することは困難であるため，既存の資産であるレガシコードが多く使われており，開発においてモデルとレガシコードが混在するという現状もある．

また，システムの大規模・複雑化によって，シングルプロセッサでは組込みシステムのリアルタイム制約が満たせなくなってきているため，低消費電力と高い性能を両立できるマルチコアプロセッサの需要が高まってきている．しかし，マルチコアプロセッサを効率的に利用するために，並列実行可能な部分を手動で抽出し，プログラミング・デバッグを行うことは困難である．

これらの背景から，我々は，モデルベース並列化と称し，MATLAB/Simulink[1] を用いて設計された Simulink モデルから，並列化された C コードへの自動変換フローを提案，開発している [2]．Simulink モデルから構造情報を抽出し，依存関係や処理量を考慮しながらコア割当てをすることで並列化を行っている．

モデルベース並列化では，モデルのブロック構造を解析し並列化を行っている．しかし，レガシコードを含むようなモデルの並列化では，レガシコード自体の解析も行いたいという要求も高まっている．レガシコードは逐次性が高い場合が多いが，マルチタスクや関数タスクとして記述する方法も取られており，その部分は並列実行の可能性がある．しかし，単純に並行動作させると不具合を起こす可能性もあるため，依然，依存性の解析が必要である．

そこで，本論文では，レガシコードを含むモデルを対象として並列化を実現することを目的とし，レガシコード部分を関数レベルで解析することによりモデルベース並列化を適用する手法を提案した．ソースコード内の関数を Simulink モデルのブロックに見立て，関数の中で使われる変数の依存関係を信号線と考えることで，モデルベース並列化で使用するデータ構造 BLXML[3] への変換を行う．そして，その BLXML に対してモデルベース並列化を適用することで並列化を行うフローを提案する．また，本手法の評価として，行列計算のプログラムに対して提案手法を適用し並列度や実行時間の計測を行う．

2. 準備

2.1 MATLAB/Simulink[1]

MATLAB/Simulink とは，MathWorks 社が開発しているモデルベース開発支援ソフトである．MATLAB は数値計算や解析を行う数値解析ソフトであり，Simulink は MAT-

[1] 名古屋大学大学院 情報学研究科
Graduate School of Informatics, Nagoya University
[a)] kuroyanagi@ertl.jp

Vol.2019-ARC-235 No.8
Vol.2019-SLDM-187 No.8
Vol.2019-EMB-50 No.8
2019/3/17

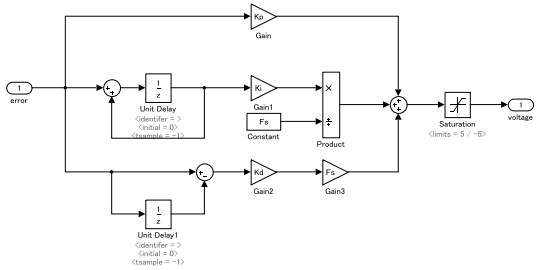

図 1 PID 制御モデル

LAB の拡張パッケージの 1 つである．MATLAB/Simulink では，システムを，処理を表すブロックとブロックの間を信号線でつないだブロック線図（Simulink モデルと呼ばれる）として記述する．この Simulink モデルを用いることで，モデルのシミュレーションによる動作確認や Simulink モデルから C コードや HDL コードの生成が可能である．

Simulink モデルの一例として，実際に制御システムに用いられる PID 制御を記述したものを**図 1** に示す．

2.2 レガシコード

レガシコードには，「仕様，意図が分からないコード」，「テストのないコード」など，様々な定義や考え方があるが，本論文では，レガシコードとは単に以前に書かれ，使用されてきたプログラムを指すとする．

組込み業界に限らず，様々な分野でシステムが開発され，プログラムとして記述されてきた．従来のコードベースの開発では，既存の仕様書やプログラムなどの資産を元に新たなシステムを作成するということが行われてきており，その中で蓄積されていった既存資産が総じて本論文でのレガシコードにあたると考える．

2.3 モデルベース並列化技術 [2]

組込みシステムの大規模・複雑化に伴うモデルベース開発の導入や，マルチコアプロセッサの需要から，我々は，モデルベース並列化と称して，様々な並列化支援フローの提案やツールの開発を行っている．モデルベース並列化では，Simulink モデルから生成した BLXML と呼ばれるデータ構造と，BLGraph と呼ばれる BLXML から生成できるグラフ構造 [3] を基盤に，コア割当てや自動並列コード生成を可能にしている．

モデルベース並列化における並列化の流れを以下に示す．

(1) ブロック構造情報の抽出
Simulink モデルから構造情報や周期情報を抽出し，BLXML として格納する．

(2) 逐次 C コードの生成
Simulink モデルから，Simulink Coder を用いて逐次 C コードを生成する．

(3) 逐次 C コードの対応付け
逐次 C コードから処理を切り出し，BLXML 上のブロックに対応付ける．

(4) ブロックの性能見積り
BLXML のブロックに対応付けたコード片と SHIM[4] と呼ばれるハードウェアの抽象化記述からおおよその処理量を計算し，性能情報として BLXML に格納する．

(5) コア割当て
BLXML のグラフ構造やブロックの処理量から，並列構造や処理分散を考慮したコア割当てを決定し BLXML に格納する [5]．

(6) 並列コード生成
コア割当て結果に沿って，BLXML 上のコード片を組み合わせることで，マルチコア上で動作可能な並列コードを生成する．現在，並列コード生成は 2 つの OS 向けに行うことができる．eMCOS[6] と呼ばれる，メニーコアプロセッサ対応の組込み向けリアルタイム OS と，車載システム向けのリアルタイム OS である，TOPPERS/ATK-2[7][8] である．

3. レガシコードを並列化する際の課題

モデルベース開発が普及する一方で，大規模なシステムのモデルを一から作成することは多大な時間と労力を要するため，既存資産であるレガシコードをモデルベース開発に統合して使用するという現状もある．しかし，大規模・複雑化したシステムでは，モデル，レガシコードも大規模なものになっている．システムを並列化する際には，システムを適切に分割する必要があるが，特にレガシコードではプログラムがシングルコア向けに記述されており，その中から並列実行可能な部分を手動で抽出し，プログラミング・デバッグを行うことは困難である．

そこで，[9] のように，システムをモデルに変換し，並列化を行う手法も提案されているが，レガシコードでは，仕様に曖昧性があり，機能間の関係がわからないという場合も多い．

4. レガシコードを用いたモデルベース並列化適用手法の概要

本章では，レガシ C コードに対してモデルベース並列化を適用する手法の概要について述べる．

4.1 提案フロー

本研究では，C コードを関数レベルで解析することによって BLXML に変換し，その BLXML に対してモデルベース並列化を適用，並列化を行うフローを提案する．C コードから BLXML を生成することにより，モデルベース並列化が適用できるだけでなく，コア割当てや実行順序の可視化など，モデルベース並列化におけるほかの機能も利用できるという利点もある．これにより C コードとモデルを協調して解析ができるようになると考える．

Vol.2019-ARC-235 No.8
Vol.2019-SLDM-187 No.8
Vol.2019-EMB-50 No.8
2019/3/17

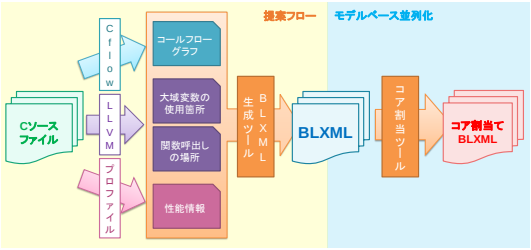

図 2　C コードを用いてモデルベース並列化を行うフロー

実際の並列化フローは図 2 である．黄色で囲われている部分が本研究で提案したフロー，青色で囲われている部分が従来のモデルベース並列化のフローとなっている．これらのフローは具体的には次の手順で行う．

(1) C コードから入力となるファイル群を生成する
- 関数のコールフローグラフ
- 関数と大域変数の使用箇所
- 関数呼び出しの場所
- 性能情報

(2) 入力ファイルを用いて BLXML 生成ツールを適用し，BLXML を生成する

(3) 生成した BLXML に対し，コア割当てツールを適用する

上記のうち手順 1 と手順 2 は本研究において提案・実装を行った部分であり，5 章から 7 章において詳述する．手順 3 はモデルベース並列化のコア割当てツールを使用している．本来のモデルベース並列化フローでは，C コードと BLXML の対応付けや並列コード生成を行うことができるが，本研究では使用していない．関数のコールフローグラフのみから BLXML を生成し，コア割当てを行うのみである．C コードとの対応付けや並列コードの生成は今後の課題である．

4.2　準備

本項では，提案フローを実装する際に使用したツール等について説明する．

4.2.1　cflow

cflow[10] は，ソースコードから，各関数から呼び出される関数を解析して表示するツールである．また，関数の呼び出し関係の他に，定義ファイルや関数の型の情報も出力でき，オプションによって解析する関数の深さや取り込むデータを指定できる．cflow では，関数の呼び出しをインデントで表しており，このツールの結果を用いて関数のコールフローグラフを生成した．

4.2.2　LLVM

LLVM プロジェクト [11] とは，再利用性の高いコンパイラと周辺ツールの技術の集合であり，イリノイ大学の研究プロジェクトとして，任意の言語に対して SSA ベースのコンパイラのためのコンパイラインフラストラクチャを作成することを目的に開始された．LLVM は多くのサブプロジェクトで構成されており，代表的なものとして LLVMCore ライブラリ，Clang コンパイラなどがある．また，LLVMCore ライブラリは LLVM-IR として知られている．

LLVM はフロントエンド，ミドルエンド，バックエンドで構成されているが，各エンドは互いに独立しているため，それぞれの成果物を単体で利用することが可能である．

LLVM-IR は，LLVM 内部で使用される中間表現であり，C 言語などのモジュールの構造を持った Module というクラスを持つ．Module 内には Global Variable や Function というクラスがあり，関数や変数の情報を保持している．

本研究では，対象のソースコードを LLVM-IR に変換し，変数や関数の情報を取得した．

5.　C コードから BLXML への変換方法

5.1　前提

4.1 項で提案したフローを踏まえ，C コードから BLXML を生成する際の前提は以下の 3 つである．本研究では，下記の前提を満たすプログラムに対して本提案手法を適用している．

- C コードが関数タスクで記述されている
- 関数間のデータのやりとりは主に大域変数を用いて行われる
- C コードの LLVM 化ができる

大規模なシステムにおいて，命令レベルの解析に比べて容易で高速な点や，ある機能やまとまり毎の解析が可能な点から，本手法では関数レベルで解析を行っている．そのため，処理を全て展開したプログラムには本手法を適用できない．また，大域変数を通して関数間の依存関係の解析を行っているため，局所変数や変数・関数ポインタ，関数の引数，返り値のみで構成されているプログラムでは，正しく依存関係の解析ができない．そして，その大域変数の情報は，LLVM を用いて取得するため，C コードの LLVM 化が必要になっている．

5.2　BLXML と C コードの対応関係

本研究では，関数をブロック，関数間の変数の依存関係をブロック間の接続として解析を行った．その際の C コードの情報と BLXML の要素の対応関係を表 1 に示す．C コードから表 1 に書かれている情報を抽出し，BLXML の各要素として対応付けることで BLXML への変換が可能になる．

5.3　接続情報の生成方法

本項では，ブロック間の接続情報を生成する方法について詳しく説明する．4 章で述べたように，C コードにおける変数を Simulink における信号であると考えた場合，あ

Vol.2019-ARC-235 No.8
Vol.2019-SLDM-187 No.8
Vol.2019-EMB-50 No.8
2019/3/17

表 1　BLXML と C コードの対応関係

BLXML			Cコード
タグ	属性	内容	内容
block	blocktype	通常のブロック	内部で関数を呼ばない関数
		サブシステムブロック	内部で関数を呼ぶ関数
	name	ブロック名	関数名
	measuringResult	性能情報	実行時間情報
	sequence	実行順序情報	関数の実行順序
input/output	line	in/outputの信号線	大域変数，引数
	port	入出力信号のポート	関数名+引数/返り値の番号
connect	block	接続先ブロック	前後で使用される関数

る変数の，関数間の依存関係がモデルにおける信号線，つまり接続情報に当たる．したがって，各変数が，どの関数で，どういう順序で使われたかという情報が必要になるといえる．

本手法において接続情報を生成する際に必要となる情報は以下の 3 つである．

- 関数の呼び出し関係を表す木構造
- 関数の実行順序
- 各関数内で使用する変数のリスト

関数の呼び出し関係を表す木構造は cflow の結果を用いて生成し，木のノードが関数を，エッジが呼び出し先を表している．同じ関数が別の関数から呼び出された場合は，その関数名の末尾にインデックスをつけてノードを生成することで別々のノードとして生成される．関数の実行順序は，C コード上での実行順序で関数の完了タイミングが基準となる．これは，Simulink モデルではブロックの処理が完了した時点でその出力を送信できるようになることが理由である．C コードにおいては関数が完了したタイミングがそれに当たると考えるため，このように実行順序を決定した．関数内で使用する変数は LLVM を用いて生成した．対象の C コードを LLVM-IR に変換することで，関数とその中で使用する変数の情報を容易に取り出せるようになっている．

5.3.1　接続情報の算出

前述の情報を用いて接続情報を生成する方法を詳しく説明する．ここでは，上に挙げた関数のコールフローグラフを利用した木構造や，関数の実行順序，大域変数の情報はすでに取得できているものとする．

詳しいアルゴリズムを以下に示す．

(1) 各関数に対して，関数内で使用する変数の情報を格納する．
(2) 末端の関数（内部で関数を呼ばない関数）から親の関数へ，自身の使用する変数の情報を伝播する．
(3) 子を持つ関数において，使用する変数の数だけ Inport/Outport ブロックを追加する．
(4) 子を持つ関数において，各変数を使用する関数を子の関数の中から抽出し，それらを実行順序でソートする．
(5) ソートした関数群について隣り合う関数同士を接続する．

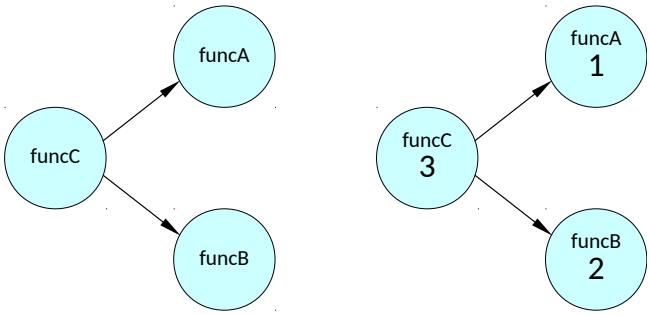

図 3　生成される木構造（左）と実行順序（右）

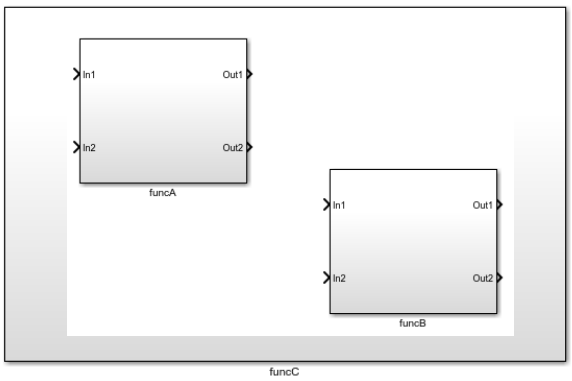

図 4　図 3 の構造を表す Simulink モデル

5.3.2　接続情報の生成例

本節では接続情報を生成する手順を例を挙げて説明する．説明には，内容を見やすくするため Simulink モデルの図を用いる．

まず，cflow によって**図 3** の左図のようなコールフローグラフが生成でき，右図のような実行順序が生成できるとする．これを Simulink モデルの構造で表したものが**図 4** である．

手順 1 として，関数 A と関数 B の内部情報に，使用する変数としてそれぞれ a,b と b,c が格納される（**図 5**）．

手順 2 において，末端の関数 A,B から親の関数 C へ，変数の情報を伝播する．これによって，関数 C では変数 a,b,c を使用するということになり，関数 C で使用する変数に a,b,c が追加される（**図 6**）．

次に手順 3 を行う．関数 C では a,b,c の 3 つの変数を使用することが手順 2 からわかった．したがって，変数 a,b,c それぞれを表す Inport/Outport ブロックを直下の階層に設置する．（**図 7**）このとき，設置した Inport ブロックの実行順序は 0，Outport ブロックの実行順序は ∞ に設定される．手順 3 は関数 C のみで行われ，末端の関数 A,B では行われない．

ここまでの手順で，接続情報の生成に必要なブロックが揃うことになる．

手順 4 において，関数 C の直下にあるブロックから，変数 a,b,c を使用するブロックをそれぞれ抽出する．この例

Vol.2019-ARC-235 No.8
Vol.2019-SLDM-187 No.8
Vol.2019-EMB-50 No.8
2019/3/17

表 2　抽出したブロックとソート後の順序

変数	抽出したブロックと実行順序	ソート後
a	A，Inport，Outport [1，0，∞]	Inport，A，Outport [0 → 1 → ∞]
b	A，B，Inport，Outport [1，2，0，∞]	Inport，A，B，Outport [0 → 1 → 2 → ∞]
c	B，Inport，Outport [2，0，∞]	Inport，B，Outport [0 → 2 → ∞]

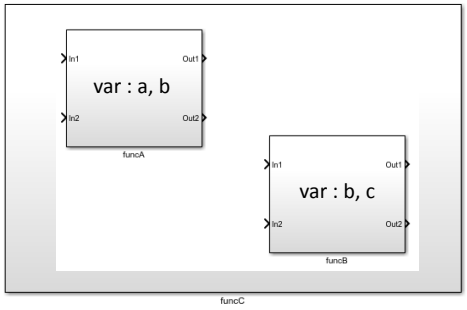

図 5　手順 1 を実行した後の構造

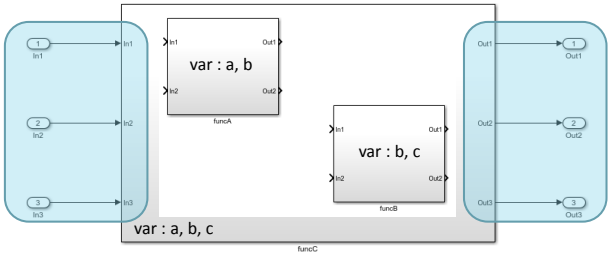

図 6　手順 2 を実行した後の構造

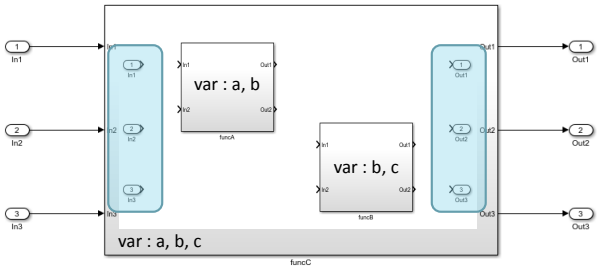

図 7　手順 3 を実行した後の構造

では，変数 a を使用するのは，関数 A，Inport，Outport の 3 つである．同様に変数 b では，関数 A，関数 B，Inport，Outport の 4 つ，変数 c では，関数 B，Inport，Outport の 3 つである．これらを実行順序でソートし，まとめたものが**表 2** になる．

最後に手順 5 において，ソート後のブロックで，隣り合うもの同士を接続する．変数 a についてみると，Inport と関数 A，関数 A と Outport が接続される．変数 b,c についても同様に行うと**図 8** になる．

これらの手順を関数の各階層で行うことで，複数の階層がある場合でも接続情報が生成できる．

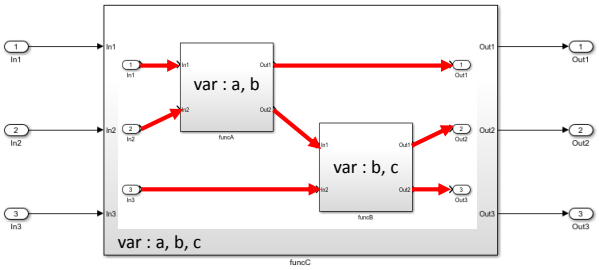

図 8　手順 5 を実行した後の構造

```
1  int a, b, c;
2
3  void funcC(){
4          funcA();
5          funcB();
6  }
7
8  void funcA(){
9          b = a*3;
10 }
11
12 void funcB(){
13         c = a+2;
14 }
```

図 9　問題の発生するソースコード

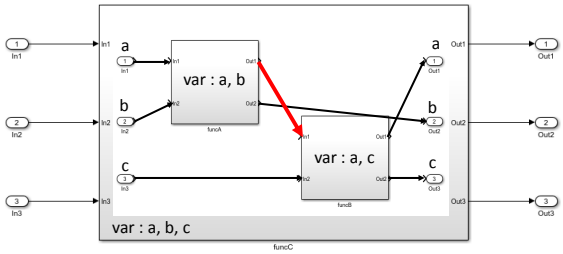

図 10　問題が発生する例

6. 変数の読み書きを考慮した接続情報の生成

生成される接続情報に対し，変数の読み書きを考慮してより正確なものにするため，5.3 項の生成方法の改善を行った．本章では，その方法について説明する．

5.3 項の方法では，関数内で使用する変数は全て，関数の入力と出力の両方に割当てている．しかし，この方法では，関数間に余分な接続関係がついてしまう場合がある．その例が**図 9** であり，5.3 項の手法を適用して生成されるモデル図が**図 10** である．変数 a に着目して接続情報を生成すると，ソート後には [Inport，funcA，funcB，Outport] となり，隣同士が接続される．したがって，funcA と funcB の間に依存関係があるということになる．しかし，もとの C コードにはその 2 つの関数間に依存関係がないことが分かるだろう．

変数の依存関係は，その変数に何かしらの更新があった場合にそれ以降の処理との間に発生する．逆に言えば，変数の更新がなければ以降の処理との間に依存関係は発生しない．つまり，ある関数内において読込みでしか使われない変数では，変数の値の更新は発生しないため，その関数から以降の関数に依存関係は発生しないといえる．先の例でも，変数 a は funcA，funcB において読みでしか使われていない．従って，その関数では出力の接続情報を生成する必要がない．

そこで，本項では変数の読み書きを考慮した接続情報の生成手法を提案する．変数の入出力への割当てや接続関係の生成を変数の読み書きによって場合分けすることで，前述の余分な依存関係が生成されることを防ぐことができる．さらに，BLXML 内のブロックを減らすことができ，BLXML の簡素化にもつながると考えられる．

上記の手法を BLXML 生成ツールに実装した．また，変数の読み書き情報は LLVM を用いて取得した．以下に実際のアルゴリズムについて示す．5.3 項同様，コールフローグラフ，実行順序の情報は取得できているとする．

(1) 生成された木構造の各関数に使用する変数の情報を格納する．
(2) 変数の読み書きに応じて，関数の入出力に変数を割り当てる．ただし，読み書き両方で使われていた場合は書きを優先する．
- 書き込みの場合
  関数の入力と出力両方にその変数を割り当てる．
- 読み込みの場合
  関数の入力のみにその変数を割り当てる．

(3) 末端の関数から親の関数へと変数を伝播する．この際，入力の変数と出力の変数は別々に伝播させる．
(4) 子を持つ関数において，入力の数だけ Inport を，出力の数だけ Outport を追加する．
(5) 子を持つ関数において，各変数を使用する子ノードを抽出し，実行順序でソートする．
(6) ソート後の先頭のノードを最新の関数として保持する．
(7) 2 番目のノードから順に以下の処理を行う．
手順 5 で注目した変数を，
- 入出力両方に持っている（書き）
  最新の関数と現在の関数を接続し，最新の関数を現在の関数で更新．
- 入力のみに持っている（読み）
  最新の関数と現在の関数を接続．

この手法では，読みしか行わない変数は関数の入力のみに，書きを行う変数は入出力両方に割り当てている．その後，その変数の出力があるかどうかを判定することで，読みしか行わない関数から以降の関数へと接続関係がつかないようになっている．また，入出力で分けることによって，必要のない出力変数は省略されるため，Outport ブロックの削減にもつながる．

7. 関数呼び出し以外の処理を表現

通常のソースコードは，関数と関数呼び出しのみで構成されているわけではなく，関数呼び出しの前後にその他の処理を行っている．しかし，5.3 項の手法ではそれらの処理を BLXML 上で表現できてない．なぜなら，コールフローグラフのみからブロックを生成しているため，関数呼び出し以外の処理を取得できないからである．一方，それらの処理で使われる変数は LLVM から取得できている．つまり，使用する変数が存在するのに実際にその処理を行うブロックが存在しないという状態になっている．このために，本来必要な接続関係を生成できないという課題があった．そこで，本項では関数呼び出し以外の処理を BLXML で表現するために，それらの処理を新たなブロックとして追加する手法を提案する．

関数呼び出し以外の処理を追加するには，その処理がどの関数呼び出しの前後にあるかという情報が必要になる．そこで，本手法では，ソースコード上での行番号に着目して位置関係を解析した．本研究では，変数がどの関数内で使用されているかを取得し，関数間でその順序を解析することによって関数間の接続情報を生成していた．従って，コード上の処理は変数への操作とみなすことができる．そこで，変数の情報として，変数名と使用する関数，その読み書きに加え，その処理が C コード上で何行目にあるかを取得した．さらに，関数呼び出しの位置も必要になるため，各関数において，呼び出す関数とその行番号も取得した．これらの情報も LLVM を用いることで取得できる．

これらの情報から，ブロックを生成するアルゴリズムが以下である．

(1) 子を持つノードにおいて，直下で使用する変数とその行番号を抽出する．
(2) 関数呼び出しの行番号を抽出し，昇順にソートする．
(3) 各変数について，行数をもとに関数呼び出し間の位置を探索し，その位置を表すリストに追加する．
(4) リストに格納された変数を使用する変数として，新しいノードを生成する
(5) 生成したノードを手順 1 のノードの子ノードとしてリストに追加する．
(6) 子ノードのリストに適切に実行順序を振る．

これによって，変数が使用される位置を考慮して，その処理を新たなブロックとして追加することができる．また，実行順序についても求めた位置関係から決定することができる．

この手法で生成したブロックは，従来の手法で生成したブロックと同様に扱うことができるので，両方を含んだブロック群に対して 5.3 項で説明した手法を適用することで，接続関係が生成できるようになっている．

Vol.2019-ARC-235 No.8
Vol.2019-SLDM-187 No.8
Vol.2019-EMB-50 No.8
2019/3/17

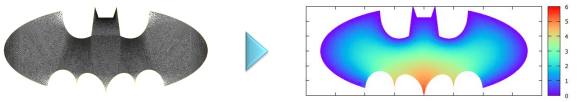

図 11　プログラムの入出力画像

表 3　各関数の実行時間

関数名	実行時間 (ms)	回数	1 回の実行時間 (us)
dmatvec	50385	392	128533.163
set_dbc	80	1	80000
solve_cg	35	1	35000
assem_ecm_full	18	1	18000
read_msh	1	1	1000
ddot	10	1171	8.540
daxpy	2	780	2.564
mk_ecm_impl2	1	21094	0.047
mk_ecm_impl3	1	21094	0.047

8. 評価

8.1 評価プログラムと評価環境

実際に汎用 PC 上で動作するプログラムに対して本手法を適用し評価を行った．対象とするプログラムは，行列計算を行うプログラムである．メッシュ状に分割された領域を入力することで，2 次元ポアソン方程式を解き，結果となる分布を出力する．プログラムの入力と出力画像が**図 11** になる．

また，本評価は以下の環境で行った．

- プロセッサ：Intel core-i7 2600（4 コア 8 スレッド）
- 実行環境：Visual Studio 2017

8.2 逐次性能計測

ツールの入力となる性能情報を取得するために，プログラムの逐次実行時間を計測した．計測には Visual Studio のプロファイリング機能を利用し，セルフの実行時間を取得した．計測結果が**表 3** である．

この結果から，実行時間の大部分を dmatvec という関数が占めていることが分かる．従って，dmatvec 関数を並列に実行することができれば，性能が大きく向上すると考えられる．そこで，本評価ではこの dmatvec 関数を新しい関数に 2 分割した．dmatvec 内の for 文を前半と後半に分け，dmatvec_impl1 と dmatvec_impl2 として定義した．

8.2.1 提案フローの適用と性能評価

前述のように修正したプログラムから入力となるファイルを生成し，BLXML を生成した．その後，モデルベース並列化のコア割当てツールを適用することでコア割当てを行った．その結果，dmatvec_impl1 はコア 0 に dmatvec_impl2 はコア 1 に割り当てられた．

次に生成されたコア割当てに従って，並列コードを作成した．並列コードの作成には OpenMP を用いた．OpenMP

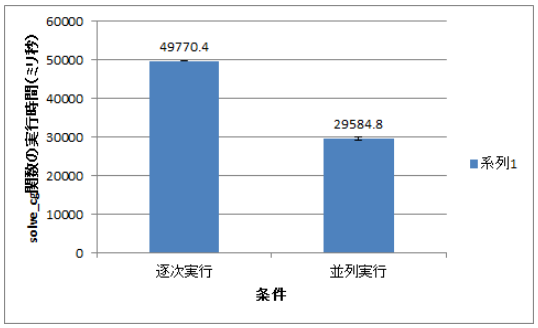

図 12　逐次実行と並列実行の solve_cg の実行時間
（エラーバーは 99.7 信頼区間）

の section によって，dmatvec_impl1 と dmatvec_impl2 を別々のコアで実行するようにしている．

作成した並列コードを VisualStudio 上で実行し，逐次実行と 2 コアでの並列実行のそれぞれにおいて実行時間を計測した．その計測結果が**図 12** である．逐次実行ではおよそ 50 秒であった実行時間が，並列実行時は 30 秒になっている．従って，2 コアでの並列実行によっておよそ 1.68 倍の性能向上を確認した．

8.3 考察

並列実行可能な部分として関数を分割したソースコードにおいてツールを適用し並列化を行ったところ，分割した 2 つの関数が別々のコアに割り当てられた．これは，ソースコードから生成した BLXML において，分割した関数が並列化可能な部分として表現できていたため，コア割当て時に別々のコアに割り当てることができたと考えられる．従って，ソースコードにおける関数間の関係を抽出できたといえる．

並列性能評価では，並列実行では，逐次実行に比べて 1.68 倍の性能向上に成功した．実行時間の大部分を占める関数を半分に分割しているため，理想的には性能が 2 倍になるはずであるが，実際の性能向上は 1.68 倍にとどまっている．これは，分割した for 文を 1 つに結合するなどの処理のために，並列化のオーバヘッドが発生したためであると考えられる．本評価では，分割した dmatvec 関数のみを並列実行したが，その他の関数の割当ても並列コードに適用し，性能を計測する必要もあると考える．

9. おわりに

本論文では，レガシコードを BLXML に変換することで，モデルベース並列化を適用し並列化を行う手法を提案し，行列計算プログラムでの適用評価を行った．ソースコードにおける関数を Simulink モデルにおけるブロックに見立て，変数の依存関係をブロック間の接続と考えることで変換が可能になった．また，変数の読み書きや，関数呼び出し以外の処理を考慮することで，より高精度な構造解析が可能になった．また，行列計算のプログラムを用い

た評価では，生成される BLXML の構造の正確性を確認し，コア割当てから並列コードを作成し実行することで，性能向上を実現した．

今後の課題としては，関数の引数や返り値の解析，依然としてある余分なブロックを削減することである．これらを考慮することでより正確に構造を抽出した BLXML が生成できるようになると考える．また，並列コードの自動生成方法も今後の課題としてあげられるだろう．

謝辞 本研究を進めるにあたり，数々のご助言・ご協力を頂きましたトヨタ自動車株式会社の高辻義人様，竹澤友紀男様，鈴木芳行様に深く感謝致します．

参考文献

[1] Mathworks:matlab/simulink. http://www.mathworks.co.jp.
[2] 山口滉平, 竹松慎弥, 池田良裕, 李瑞徳, 鍾兆前, 近藤真己, 枝廣正人. Simulink モデルからのブロックレベル並列化. 組込みシステムシンポジウム 2015 論文集, 第 2015 巻, pp. 123–124, oct 2015.
[3] 山口滉平. モデルベース開発におけるブロックレベル並列化のためのデータ形式. Master's thesis, 名古屋大学, 2016.
[4] M. Gondo and F. Arakawa and M. Edahiro. Establishing a standard interface between multi-manycore and software tools - SHIM. In *2014 IEEE COOL Chips XVII*, pp. 1–3, 2014.
[5] 鍾兆前, 枝廣正人. 組込み制御システムに対するマルチコア向けモデルレベル自動並列化手法. 情報処理学会論文誌, Vol. 59, No. 2, pp. 735–747, 2 2018.
[6] eMCOS. https://www.esol.co.jp/embedded/emcos.html.
[7] TOPPERS/ATK-2. https://www.toppers.jp/atk2.html.
[8] 中野友貴. モデルベース並列化 (MBP) におけるマルチレートモデルの車載 RTOS 向けランタイムとコード生成. Master's thesis, 名古屋大学, 2017.
[9] Juraj Feljan, Jan Carlson, and Tiberiu Seceleanu. Towards a Model-Based Approach for Allocating Tasks to Multicore Processors. In *38th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 117–124, 2012.
[10] GNU cflow. https://www.gnu.org/software/cflow/.
[11] The LLVM Compiler Infrasrtucture. http://llvm.org/.