高性能計算における Verification-Oblivious プログラミングのためのディレクティブに基づくソースコード検証

安 部 達 也[†1] 佐 藤 三 久[†1,†2]

プログラミング言語 XcalableMP/C ではソースコードにディレクティブを付加することで逐次プログラムの並列化が可能である．本稿では，高性能計算分野におけるプログラマに検証を意識させることなくプログラミングを行わせるために，この並列化のために付加されるディレクティブをプログラム検証にも流用する方法を提案する．この方法における検証は，ソースコードに対するものであるため，静的かつ部分的に行うことができる．また，ディレクティブがそのプログラムのデータの分散と同期に関して抽象したものになっていることを利用することで，いわゆる状態爆発の問題を回避することができる．

Directive-based Source Code Checking for Verification-Oblivious Programming in High Performance Computing

Tatsuya Abe[†1] and Mitsuhisa Sato[†1,†2]

In the XcalableMP/C language programmers can parallelize sequential programs by adding directives to their source codes. In this paper we propose a method of diverting such parallelization directives to program verification in order to provide programmers in high performance computing with verification-oblivious programming. Verification in the method is statically and partly applied to programs since it is source code checking. The method also helps us avoid the so-called state explosion problem by using that for any program the sequence of the its directives can be considered an abstraction of the program with respect to data distributions and synchronizations.

1. はじめに

並列分散計算環境を可能な限り有効に利用するためにはプログラムが並列実行できる箇所を多く持つことが必要である．一方，高性能計算分野はこれまでに利用されてきた逐次プログラムという資産を多く持つ．この資産を今後も有効に利用するためには逐次プログラムの並列化を行うことが必要である．

高性能計算分野においてプログラムを並列化する方法には大きく分けて二つの方向がある．プログラムを並列実行可能にする，いわゆる並列プログラミング言語で一から書き直す方向と，逐次プログラムに変更を加えることなく処理系でプログラムの並列実行を可能にする，いわゆる自動並列化である．前者は人手によるプログラムの変更を必要とする点で逐次プログラムという資産を有効に利用しているとは言い難いという欠点を持つ．一方，後者もプログラムの並列実行可能な部分を何のヒントもなく処理系が読み切ることには限界があるため，事実上プログラマにはそれのヒントを処理系に与えることが求められるという，つまり，これも人手によるプログラムの変更を必要とするという欠点を持つ．しかし，後者におけるヒントの与え方として，逐次プログラムにディレクティブと呼ばれるものを付加するということだけを要求する並列化手法が存在する．この手法で得られる並列プログラムは，ディレクティブを無視すればほぼ元の逐次プログラムであることから，逐次プログラムという資産を比較的有効に利用しているといえる．

本稿では，ディレクティブを付加することにより得られる並列プログラムを検証するための手法を提案する．特に，並列プログラミング言語 XcalableMP/C[6]⋆1 と呼ばれる，大雑把にいうところのプログラミング言語 C に並列実行のためのディレクティブを利用できるように設計された言語で書かれたプログラムの検証を目的とする． 逐次プログラムで検証したい性質は並列プログラムにおいても検証したい．しかし，逐次プログラムでも検証したい性質は本研究の対象から外すことにして，並列プログラムだけで検証したい性質に焦点を当てる．特に，本研究では二つのプログラムに対し

†1 理化学研究所計算科学研究機構
Advanced Institute for Computational Science, RIKEN
†2 筑波大学計算科学研究センター
Center for Computational Sciences, The University of Tsukuba

⋆1 http://www.xcalablemp.org/

て，一方のプログラムが他方のプログラムの並列化であるかという検証，つまり，プログラムの等価性に関する検証を行う．

本研究では自動検証を目指す．一般にプログラム検証を行うにあたりプログラマに何の負担を求めないことは困難と考えられている．それにも関わらず，本研究ではプログラマに検証のために余計な手間をかけさせないことを約束する．これを可能とする方法は，並列実行のためにソースコードに記述されるディレクティブを検証に流用することである．こうすることにより，プログラマにプログラム検証のためだけには余計な手間をかけさせない（プログラム並列化のための手間を既にかけさせていると考える）．また，逐次プログラム自体の検証を本研究で対象としないため，その検証の手間を考慮しないということは前述の通りである．

検証を行うにあたって有界モデル検査と呼ばれる方法を用いる．一般にプログラム検証を行うにあたってソースコードからモデルを構成することは，モデルの状態数が肥大化し，それにより検証が終わらなくなってしまう，いわゆる状態爆発と呼ばれる問題を引き起こす．状態爆発を回避するには検証に必要のないことをプログラムから除去する，いわゆるプログラムの抽象化を行うのが一般的な方法である．本研究では，逐次プログラムの並列化のために付加されたディレクティブを検証にも流用すると前述した．ディレクティブが付加されている箇所は並列実行を行いたい箇所であり，逐次実行箇所よりも検証したい箇所といえる．これはつまり，検証したい箇所に目印がついているということである．この意味で，ディレクティブがそのプログラムのある種の抽象になっていると考えることで状態爆発の問題を回避できるのではないかと期待している．また，有界モデル検査をソースコードに適用するにあたり，本検証を部分検証可能であるように設計している．これは一般に長時間の実行時間を要する高性能計算分野のプログラムを検証するために，その実行時間と同程度の時間を要しては無意味であるからである．

本稿の構成は以下である．2 節で，プログラミング言語 C の拡張で，並列処理に関することをディレクティブにより記述させるプログラミング言語 XcalableMP/C を紹介する．3 節で，今回の検証フローで採用する有界モデル検査器 Bounded Model Checker for ANSI-C (CBMC) とその他のモデル検査器を簡単に紹介する．4 節で，今回の検証フローを説明する．5 節で，今回の検証フローを実行するにあたり開発したツール Verifier for Directive-based Source Codes の実装を概説する．6 節で，実際に本ツールで元プログラムとその並列化との間の等価性検証を行う．7 節で，関連研究を紹介し，将来課題を述べることでまとめる．

2. 並列プログラミング言語 XcalableMP/C

XcalableMP/C はプログラミング言語 C の拡張言語である．XcalableMP/C 独自の構文は原則，ディレクティブと呼ばれる，処理系がそれに何ら処理を施さない場合でもそのプログラムが実行可能である部分に現れる．図 1 における `#pragma xmp` で始まる行がディレクティブである．C の処理系が図 1 のプログラムをディレクティブをすべて無視して処理した場合，図 1 におけるループ（10 反復）は一つの計算機で処理される．その一方，XcalableMP/C の処理系はディレクティブを処理することでプログラム全体を並列実行可能なものに翻訳する．`#pragma xmp nodes p(2)` は計算ノードの名前（ここでは `p`）と数（ここでは 2）の宣言であり，`#pragma xmp template t(0:9)` はテンプレートと呼ばれる仮想的な配列（ここでは名前が `t` でサイズが 10 の配列）の宣言である．テンプレートは`#pragma xmp distribute t(block) onto p` ディレクティブで計算ノードに分散される（ここではブロック分散，つまり，`t[0]`,...,`t[4]` は `p(0)` に `t[5]`,...,`t[9]` は `p(1)` に分散される）．配列 `a` は`#pragma xmp align a[i] with t(i)` ディレクティブでテンプレート `t` に割り当てられる．

ループ文の一行上の `#pragma xmp loop on t(i)` ディレクティブにより各反復は各計算ノードで実行される．

`#pragma xmp reduction(+:asum)` は集計計算のためのディレクティブである．もしこれを明示しておらず，つまり，`asum` が`+` に関する集計計算であることを処理系に教えておかなければ，前述のループ文における各反復を単純に並列実行してしまうと `asum` に格納される値が思っているものと異なるものになってしまう．このディレクティブを記述しておくことで `a[0]` から `a[9]` の総和が計算されて，それが `asum` に格納される．

3. 有界モデル検査器 Bounded Model Checker for ANSI-C

Bounded Model Checker for ANSI-C（CBMC）[⋆1]はプログラミング言語 C [⋆2]のプログラムの有界モデル検査器である．先行するモデル検査器 SMV[?)⋆3]や SPIN[3)⋆4] がそれぞれ専用の記述言語を持つのに対し，CBMC は汎用である C を記述言語とする．CBMC は C を完全に網羅できており，ポインタや浮動小数点を適切に処理できることは特筆すべきことである．

⋆1 http://www.cprover.org/cbmc/

⋆2 本稿では対象外であるプログラミング言語 C++ のプログラムの検査も可能である．

⋆3 http://nusmv.fbk.eu/

⋆4 http://spinroot.com/spin/

```
#pragma xmp nodes p(2)
#pragma xmp template t(0:9)
#pragma xmp distribute t(block) onto p
int a[10];
#pragma xmp align a[i] with t(i)
int main() {
 int asum;
 #pragma xmp loop on t(i)
 for(i=0; i<9; i++) {
  a[i]=i;
  asum = asum+a[i];
 }
 #pragma xmp reduction(+:asum)
 return asum;
}
```

図 1 XcalableMP/C プログラムの一例

CBMC はソースコードモデル検査を行う検査器であるが，ソースコード中の特定の関数（つまり `main` でなくてもよい）だけを検査すること，つまり部分的に検査することができる．

CBMC はソースコード中に埋め込まれた表明

```
assert(真偽文);
```

を見つけると，その真偽文を検査する．その真偽文が真であれば真であると，偽であれば反例を返す．

4. 検証の流れ

本検証の流れは以下である（図 **2**）．まず，XcalableMP/C のソースコードを XcalableMP/C 処理系 Omni OpenMP/XcalableMP Compiler Software 0.5.3 のデバッグモードで処理する．これにより，実行コードだけでなく，中間ファイルであるところの XcalableMP/C ディレクティブ展開前と展開後の XcodeML/C[7)⋆1] 文書二つも得られる．ここで XcodeML/C はほぼ C と同じ意味論を持つ XML の一つである．これらの XcodeML/C 文書二つを本検証にあたって自作した翻訳器に与えることで 二つの C コードに翻訳する．ディレクティブ展開前の XcodeML/C コードを本翻訳器に与えて得られる C コードはディレクティブを無視した際に得られる逐次プログラムと同等の動作をするはずのものである．一方，ディレクティブ展開後の XcodeML/C 文書を本翻訳器に与えて得られる C コードは XcalableMP/C 処理系が生成する C コードと同等である．この二つの C コードと検証したい性質を記述したものから CBMC 表明付 C コードを生成する．この最終的に得られた CBMC 表明付 C コードを CBMC に与えることで検証を終える．

```
type CFilter i = Content i -> [Content i]
tag :: String -> CFilter i
children :: CFilter i
o :: CFilter i -> CFilter i -> CFilter i
f `o` g = (concatMap f) . g
```

図 3 `Text.XML.HaXml.Combinators` から抜粋

5. 実　　装

前節で紹介した検証の流れを実現するツール，検証器 Verifier for Directive-based Source Codes を紹介する（図 **4**）．

XcodeML/C はほぼ C と同じ意味論を持つ XML の一つであり，XML であるために C よりも扱いやすくなっている．本研究でもプログラムの抽象化を行うにあたって C コード自体でなくそれを翻訳した XcodeML/C 文書を扱っている．

翻訳器（`veridcode.hs`）自体はプログラミング言語 Haskell で実装した．XML 文書を扱うには Document Object Model や Simple API for XML に準拠している XML パーサがしばしば用いられるが，今回は XML 文書をパース，抽出，翻訳，生成するための Haskell ライブラリである HaXml[⋆2] を利用した．フィルタと呼ばれる XML 文書の要素から要素のリストを返すものを作成し，それらを結合することでさらに大きなフィルタを作成し，それをもって XML 文書を扱うことができるようにする機構を HaXml は提供する．図 **3** を例としてこれを説明する．`Content i` は XML 文書の要素の型であり `CFilter i` はフィルタの型である．`tag` はモジュール `Text.XML.HaXml.Combinators` で定義されている関数で，引数に文字列を受け取りその文字列を要素名に持つ要素を抽出する．`children` もモジュール `Text.XML.HaXml.Combinators` で定義されている関数で，引数の要素の子要素のリストを抽出する．`o` もモジュール `Text.XML.HaXml.Combinators` で定義されている関数であり，その定義は，

```
f `o` g = (concatMap f) . g
```

である．ただし，バッククオートで関数を囲むのは関数を中置とする Haskell の記法である．つまり，フィルタが要素から要素のリストを返す関数であるので，その結合は単に `.` でなく `o` でなければならないということである．

ディレクティブの解釈を含まない，XcodeML/C 文書を C コードに翻訳するだけの Haskell プログラムは HaXml を利用することでおよそ 250 行で記述できる．

検証項目である CBMC 表明付 C コードは，本質的には翻訳器を通して得られた二つの C コードの関数

⋆1 http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/xcodeml/

⋆2 http://projects.haskell.org/HaXml/

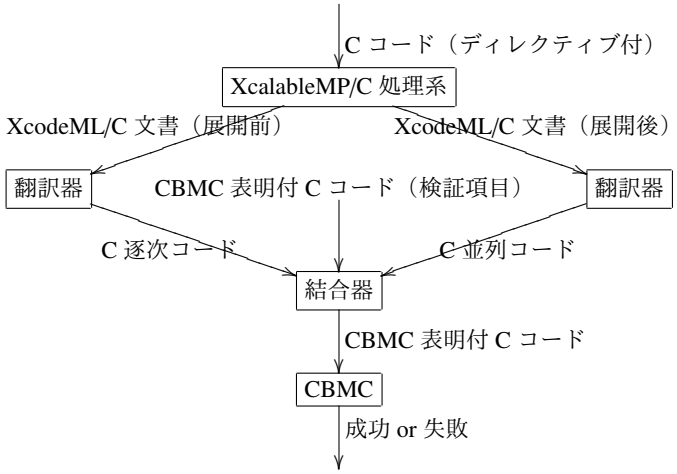

図 2 検証の流れ

（main でなくてもよい）をそれぞれ呼び出す．これは CBMC が部分的に検査ができることを利用している．具体的には，等価性を検証したい関数を main とすると，ディレクティブ展開前の XcodeML/C 文書から翻訳した C コード中の main を_ORG_main と変更し，同様にディレクティブ展開後の XcodeML/C 文書から翻訳した C コード中の main を_XMP_main と変更し，それぞれを CBMC 表明付 C コードの main から呼び出しその返り値を変数 ORG_return と XMP_return に格納する．このようにしたところで，プログラムの等価性を検証するため，今回記述する表明は以下である．

```
assert(ORG_return == XMP_return);
```

翻訳器を通して得られた C コード二つと検証項目である CBMC 表明付 C コードを結合するプログラムは単なるシェルスクリプト（concatenate.sh）である．結合して得られた CBMC 表明付 C コードは CBMC に渡されることで検証を終える．

6. 評　　価

XcalableMP のウェブページに載っているサンプルコード（**図 1**）に対し検証を行ったところ，関数_ORG_main と_XMP_main の両方とも 0 から 9 の総和である 45 を返しており，表明 assert(ORG_return == XMP_return) が成立し，検証が成功した．

XcalableMP/C 処理系に手を入れ，集計計算を各計算ノードで集計してしまうという誤りを意図的に入れたところ，前述の表明は破られ検証は失敗となった．

また，集計計算をできないようにするために，図 1 のプログラムの reduction ディレクティブを意図的に除去したところ，ノード数を 1 にしたプログラムとノード数を 2 以上にしたプログラムで asum の値が相違することを検証により確認した．

7. ま と め

XcalableMP/C におけるプログラミングでソースコードに並列実行のために付加されるディレクティブをモデルの抽象化に流用することと，CBMC が C ソースコードそのままを関数単位という部分的に検査できることを利用して，高性能計算のためのプログラムとその並列化であるプログラムの間の等価性を自動検証する手法を提案した．

関連研究としては，Siegel らの MPI-SPIN[⋆1] と Ebnanesir らの UPC-SPIN[⋆2] を挙げておく．MPI-SPIN は Message Passing Interface（MPI）というメッセージパッシングを基礎とする C の並列実行のためのライブラリを利用した C プログラムをモデル検査器 SPIN で検証するものである．Siegel らは MPI-SPIN を用いて並列数値計算プログラムの検証を行っている[4),5)]．UPC-SPIN も SPIN を利用するところは MPI-SPIN と同様であるが，Unified Parallel C（UPC）と呼ばれるプログラミング言語 C の高性能計算のための拡張言語で書かれたプログラムを対象としているという点に違いを持つ．Ebnanesir は UPC-SPIN を用いて，プログラムのレースフリー性やデッドロックフリー性の検証を行っている[2)]．

MPI-SPIN と UPC-SPIN とはそれぞれ MPI 関数（MPI_ を接頭辞とする C の関数）と upc_ を接頭辞とする UPC の関数とが検証における目印となる．これらは関数が記述できるソースコード中の任意の場所に現れる．また，それらの関数はデータを受け取ったり，そのデータの更新も行い得る．一方，本研究で対象としている XcalableMP/C においては並列処理に関

⋆1 http://vsl.cis.udel.edu/
⋆2 http://asd.cs.mtu.edu/projects/pgasver/

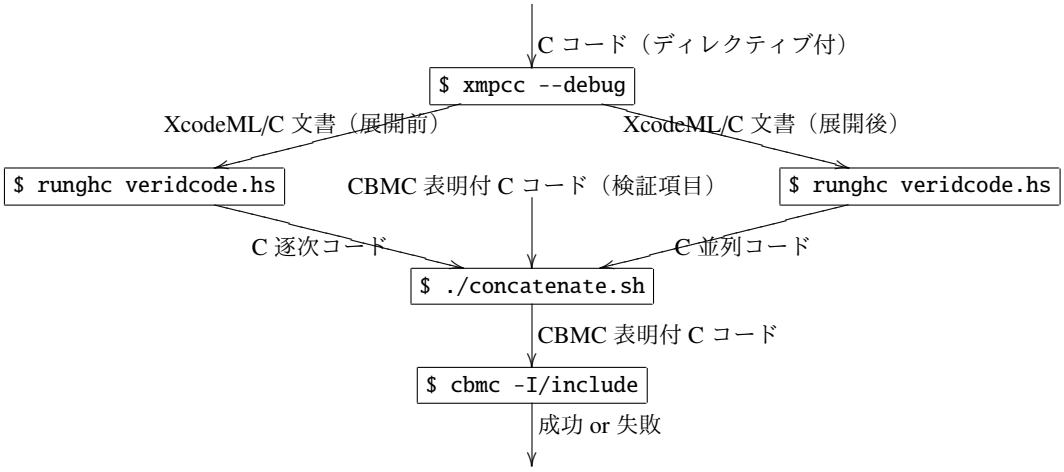

図 4 検証の流れ（実装編）

することはほぼすべてディレクティブで記述されている．そのため，MPI プログラミングや UPC プログラミングよりはプログラミングの自由度が XcalableMP/C より高い．そのかわりに XcalableMP/C コードにおいては，ほぼディレクティブのみを検査することで検証を行うことができると考えられる．

本研究では検証に失敗した際にソースコードの修正を行うための情報を取得する方法を提供していない．CBMC は検証で失敗を検出した際に表明を破る反例を出力をする．しかし，本研究ではこれらの反例からソースコード修正のための有効な情報を得る方法に関する考察を行っていない．これは将来課題である．

これまで XcalableMP/C コードは C コードにディレクティブによりを付加して得られるものであり，ディレクティブを無視すればその XcalableMP/C コードは逐次実行される C コードであるとしてきた．しかし，厳密には XcalableMP 仕様書 0.7a 版の時点でその前提は崩れている．つまり，現在の XcalableMP は並列実行のためにソースコードのディレクティブ以外の箇所に修正を必要とする仕様になっており，これはそのまま 1.0 版においても踏襲されることになっている．これはディレクティブをソースコードの抽象化に流用するという手法の適用を阻害するものではないが，今後はそれだけでは抽象化に不十分であり，ソースコードのディレクティブ以外の箇所を付加情報として利用することで抽象化を可能にすると考えられる．こういった対応も将来課題である．本稿では，本手法による検証をサンプルコードにしか適用していない．使用に耐える高性能計算のプログラムに本手法が適用可能であるかを検討することは将来課題である．

謝辞 モデル検査器全般・Haskell ライブラリ HaXml に関し有益な情報をそれぞれ与えてくれた齋藤正也・北村崇師両博士に感謝の意を表する．また，本稿は本シンポジウム中の議論を反映したものになっている．議論に参加いただいた方々に感謝する．本研究の一部は，文部科学省「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発」における課題「シームレス高生産・高性能プログラミング環境」による．

参 考 文 献

1) Cavada, R., Cimatti, A., Jochim, C. A., Olivetti, G. K.E., Pistore, M., Roveri, M. and Tchaltsev, A.: *NuSMV User Manual*, 2.5 edition (2010).
2) Ebnenasir, A.: UPC-SPIN: A Framework for A Framework for the Model Checking of UPC Programs, *Proceedings of the 5th Partitioned Global Address Space Conference*, ACM (2011).
3) Holzmann, G. J.: *The Spin Model Checker*, Addison-Wesley (2003).
4) Siegel, S. F., Mironova, A., Avrunin, G. S. and Clarke, L. A.: Using model checking with symbolic execution to verify parallel numerical programs, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ACM, pp.157–168 (2006).
5) Siegel, S. F., Mironova, A., Avrunin, G. S. and Clarke, L. A.: Combining symbolic execution with model checking to verify parallel numerical programs, *ACM Transactions on Software Engineering and Methodology*, Vol.17, No.2, pp.1–34 (2008).
6) XcalableMP Specification Working Group: *XcalableMP Application Program Interface Version 1*, 0.7a edition (2010).
7) XcalableMP/Omni Compiler Project: XcodeML/C 仕様書，0.9J edition (2009).

質 疑 応 答

問．これを使って見つけることができた例はあるか．それでどう直せばよかったか．(田中哲朗)

答．現在はない．ディレクティブ以外にユーザが何も書かないとすると、ユーザが書き間違えることは非常に少ない．処理系を作った人の間違いを見つけることはできる．この意味で，仕様と実装のずれを見つけることはできるはずである．

問．予めバグを入れたもので検証したか．(田中哲朗)

答．(ユーザが書く部分でという意味では）していない．(処理系にバグを入れるという意味では行っている．シンポジウム終了から本稿執筆までの間に検査器を改良し，ユーザが `reduction` ディレクティブを書き落としたことを検知できるようにした．6 章を参照のこと．)

問．レースコンディションは対象になっているか．(荒川淳平)

答．元々は対象にする予定だった．今回対象にした XcalableMP ではレースコンディションが起こるようなことを（ユーザには）書けない．(PGAS 言語におけるプログラミングには）グローバルビューとローカルビュー（というデータの見方）があるが，今回はグローバルビューの対応しかしていない上に，これに関するレースの有無は現在は見ればわかる程度のものしか扱えない．ローカルビューの対応ができたときには意味があるものになるはずと考えている．

問．これはプログラマ，処理系作成者等，誰のためのものであるか．(岩崎英哉)

答．(元々の）目的はユーザーのためのものだった．レースコンディションが書けるようになるとユーザのためのものになると思っているが，現在はできない．現在は，処理系を作る人向けの検査器になってしまっている．翻訳器は自作したが，(本ツールは検証項目を扱うところだけに関わるべきで）翻訳器はできるだけ処理系のものを使うべきだと思っている．

問．(本研究が扱った）バグよりはパフォーマンスバグが起こりやすいと思う．これを検査する需要はあるか．(田中哲朗)

答．私の知る限りない．調査不足かもしれない．

問．XcalableMP だとデータの分散をどうするかはユーザーが指定する．逐次コードに（ディレクティブを）追加していくことになるが，そのあたりに関する書き間違いも入ってくるのではないか．(松崎公紀)

答．そうだと思う．現在の XcalableMP 処理系のパーサも（ディレクティブに関しては）雑に扱っているようである．

問．(本稿では扱っていない XcalableMP/C のディレクティブ）shadow をどうするかとかも書けるのではないか．(松崎公紀)

答．そういう意味では（本ツールが）ユーザーの誤りを見つけてくれるものになるかもしれない．

問．事例はあるか．(小出洋)

答．自分で作ったサンプルしか回してない．ソースコード検証なので MPI はブラックボックスになっている部分を書かないといけないが，まだ書ききれてない．簡単なものに関しては検証できている．検証で MPI の関数を書き間違えたことも検出できることもあるかもしれない．

問．ソースが公開されている MPI を使うこともできるのではないか．(小出洋)

答．そうだと思う．