社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

# OSCARコンパイラのC++プログラム対応の検討

川角　冬馬† TilmanPriesner† 野口　真聖† 韓　吉新† 見神　広紀†

川島　慧大†† 田中啓士郎†† 木村　啓二† 笠原　博徳†

† 早稲田大学
〒162-0042 東京都新宿区早稲田町 27
†† オスカーテクノロジー株式会社
〒162-0042 東京都新宿区早稲田町 27
E-mail: †{tohma,tilman,noguchi,kalfazed,hiroki}@kasahara.cs.waseda.ac.jp,
††{kawashima,tanaka}@oscartech.jp, †††{keiji,kasahara}@waseda.jp

**あらまし**　オブジェクト指向の機能や各種データ構造を提供するクラスライブラリを標準でサポートするコンパイラ言語として C++ が広く用いられており，これまで C により多くのプログラムが記述されてきた分野でも C++ による開発が進められるようになってきている．これら C++ で記述されたプログラムの高速化の手段として，プログラムの並列化は依然として有効である．その一方で，実装の詳細が隠蔽されがちな C++ プログラムでプログラム全域に渡る手動並列化は困難であり，自動並列化に対する期待が高い．しかしながら，仮想関数呼び出しやポインタ変数の過度の使用が C++ プログラム自動並列化に必要なプログラム解析の阻害要因となる．本稿では，OSCAR 自動並列化コンパイラの C++ 対応のための拡張について述べる．C++ 対応の最初のステップとして，C++ フロントエンドの開発及び OSCAR コンパイラの中間表現拡張を行った．さらに，コンパイラの解析器に対して仮想関数呼び出しの解析機能の追加を行った．上記拡張を行った OSCAR コンパイラを，SPEC CPU 2006 に含まれる 447.dealII の CG ソルバを基にしたテストプログラムで評価を行ったところ，g++ を用いた逐次実行と比較して 6 コア並列実行で 3.27 倍の速度向上が得られた．

**キーワード**　コンパイラ, 並列処理, C++, 脱仮想化

## 1. はじめに

大規模アプリケーションの開発では C++ や Java をはじめとするオブジェクト指向言語が多く用いられている．これは，オブジェクト指向言語が提供するクラスの利用によるカプセル化，継承，多態性の各機能が，実装の隠蔽とプログラム部品の高い再利用性をもたらし，プログラム生産性を高めるためである．特に，C++ は各種データ構造を提供する標準ライブラリを備えたコンパイル型のオブジェクト指向言語として，高性能が要求される科学技術計算や画像処理等の，これまで C により開発が進められてきたアプリケーション領域にも広く用いられつつある [1], [2].

これらの C++ で記述されたアプリケーション高速化の手段として並列化は依然として有力な方法であり，手動並列化により大きな性能向上を得られる例も多い [3]．並列性により高い性能向上を得るためには，同期等のオーバーヘッドが性能に与える影響を小さくするために大きな粒度で並列化することが望ましい．しかしながら，C++ プログラムで大きな粒度を確保するための大域的な並列化は，前述のオブジェクト指向の機能により実装がソースコード上は隠蔽されているため困難である．そのため，C++ プログラムの並列化に対しても，コンパイラの持つ高度なプログラム解析機能に基づいた自動並列化に対する期待が高い．

一方，筆者等はこれまでプログラムから階層的に並列性を抽出して並列化する OSCAR 自動並列化コンパイラを開発してきた [4], [5]．OSCAR コンパイラは C 及び Fortran のプログラムを対象とし，高度なデータフロー解析やポインタ解析機能により自動並列化機能に加えて，メモリ最適化と低消費電力最適化を実現した．しかしながら，C++ のようなオブジェクト指向言語には未対応であった．

C++ は従来の OSCAR コンパイラが対応していた C の機能に加え，クラスの継承に伴う仮想関数呼び出し機能を持つ．仮想関数呼び出しのあるプログラムは，これにより呼び出される関数本体が一意に決まらなくなり，結果として並列化に必要なデータフロー解析が困難になる．この問題の解決方法として，C++ で記述されたソースプログラムからクラス継承関係を読み取り，仮想関数呼び出し時に呼び出される可能性のある関数を列挙する必要がある．このため，C++ ソースコードからクラスの継承関係を読み取る新しいフロントエンドとその出力を用いた解析を行うための解析器の拡張が必要となる．以上を

踏まえ，C++ で記述されたプログラムの自動並列化に向けた Clang [13] ベースコンパイラフロントエンドの実装，及びコンパイラ中間表現の拡張を行い，さらに仮想関数呼び出しの解析機能拡張を OSCAR コンパイラに対して行ったので，本稿でその報告をする．

本報告の貢献は以下の 3 点である．

- Clang ベースの OSCAR 自動並列化コンパイラ向けフロントエンドの実装，及び C++ 用中間表現拡張
- C++ の機能が用いられたプログラムの解析手法の提案
- SPEC CPU 2006 に含まれる 447.dealII の CG ソルバを基にしたテストプログラムによる提案手法の評価

以下，第 2. 節では本稿の性能評価で用いた OSCAR 自動並列化コンパイラの概要を述べる．第 3. 節では OSCAR コンパイラの C++ 拡張の概要を説明する．第 4. 節と第 5. 節では C++ 自動並列化に向けた拡張を行ったフロントエンドとミドルパスについて述べる．第 6. 節では本稿での提案手法の評価結果を説明し，第 7. 節で関連研究について述べる．

## 2. OSCAR 自動並列化コンパイラ

本稿で報告する提案手法は OSCAR 自動並列化コンパイラの枠組みの中に実装した．本節では OSCAR 自動並列化コンパイラの概要とコンパイルフローについて説明する．

### 2.1 OSCAR 自動並列化コンパイラの概要

OSCAR 自動並列化コンパイラは C または Fortran で記述されたプログラムを入力とし，並列化済みのソースコードを出力とする Source-to-source コンパイラである [4].

インテルコンパイラ等の自動並列化機能を持つコンパイラが行うループイタレーションレベルでの自動並列化 [8] と異なり，OSCAR コンパイラはプログラム全域のマルチグレイン並列化を行う．マルチグレイン並列化とは以下の 3 種類の並列化を階層的に行うことでプログラム全域から並列性を抽出する手法である．

（1） 粗粒度並列化: プログラムの基本ブロック，ループ，関数呼び出し文同士の並列化

（2） 中粒度並列化: ループイタレーションレベルの並列化

（3） 近細粒度並列化: 基本ブロック内の複数ステートメントの並列化

### 2.2 粗粒度並列化及びタスクスケジューリング手法

本節では粗粒度並列化とそれに関連したタスクスケジューリングについて述べる．

OSCAR コンパイラは並列性解析前にプログラムを基本ブロック (Basic Block, BB), ループ等の繰り返しブロック (Repetition Block, RB), 関数呼び出しを行うブロック (Subroutine Block, SB) に分割する．分割した各ブロックをマクロタスク (MT) と呼ぶ．MT 分割は SB 内部，RB 内部に対しても行う．全 MT の生成後，コントロールフロー解析，データフロー解析を行ってマクロフローグラフ (MFG) を生成する．さらに，MFG の生成後に MFG に対して最早実行可能条件解析を行うことでマクロタスクグラフ (MTG) を生成し，各マクロタスク間の並列性を得る [4]．MT 生成と同様に，MTG 生成は SB,RB 内部に対して階層的に行う [14].

MTG 生成後，各プロセッサコアに MTG 上の MT のスケジューリングを行う．スケジューリング手法はプログラムの形状によって 2 パターンが存在する．プログラムに条件分岐が存在し，実行時不確定性がある場合はプログラム実行中にコア割当を行う動的スケジューリング方式によってスケジューリングをする．プログラムに実行時不確定性が無い場合は実行前にコア割当を行う静的スケジューリング方式を用いてスケジューリングをする．動的スケジューリングと比較して静的スケジューリングでは実行時のコア割当にかかるオーバーヘッドが存在しない分，実行バイナリの性能は高くなる．

### 2.3 OSCAR コンパイラのコンパイルフロー

OSCAR コンパイラのコンパイルフローを図 1 に示す．各種プログラミング言語で記述されたソースコードに対応するフロントエンドが OSCAR 中間表現を生成する．その後，ミドルパスが OSCAR 中間表現を解析し，最適化された中間表現を生成する．最後に，バックエンドが OSCAR 中間表現をソースコードに変換する．OSCAR コンパイラはフロントエンド，ミドルパス，バックエンドの 3 モジュールで構成される．以下で各モジュールについて簡単に説明する．

- フロントエンド: ソースコードを OSCAR 中間表現に変換するモジュールがフロントエンドである．フロントエンドはこれまで Fortran 用と C 言語用が作成された．本研究ではこれら 2 つに加えて C++ 用フロントエンドを作成した．
- ミドルパス: ミドルパスはフロントエンドが出力した中間表現に対してデータフロー解析，コントロールフロー解析，MTG 生成等の解析を行う．その後，キャッシュ・メモリ最適化と複数プロセッサコアに対するコード割当を行い，最適化済み中間表現を生成する．
- バックエンド: バックエンドでは中間表現をソースコードへ変換する処理を行う．本稿での性能評価では OpenMP ディレクティブ付き C ソースコードを出力するバックエンドを使用した．C++ ソースコードから生成した中間表現を C ソースコードへ変換する際の問題点と解決策については 4.4 節で述べる．

## 3. C++ 拡張の概要

本節では本研究で実施した OSCAR コンパイラの C++ 言語対応の概要について述べる．

2. 節で述べた通り，OSCAR コンパイラはプログラムから並列性を最大限抽出するためにインタープロシージャルな解析を行う．このため，プログラムの解析にはコールグラフの生成が必要となる．しかしながら，C++ はクラス継承にともなう仮想関数呼び出しの機能を持っているため，静的なプログラム解析によるコールグラフの生成が困難となっている．本研究は従来の OSCAR コンパイラによってコールグラフ生成ができなかった C++ のソースコードを新規の C++ フロントエンドと OSCAR ミドルパスによって解析し，コールグラフ生成と解析・並列化を行うことを目標とする．

本研究で提案する C++ プログラム自動並列化のためのコン

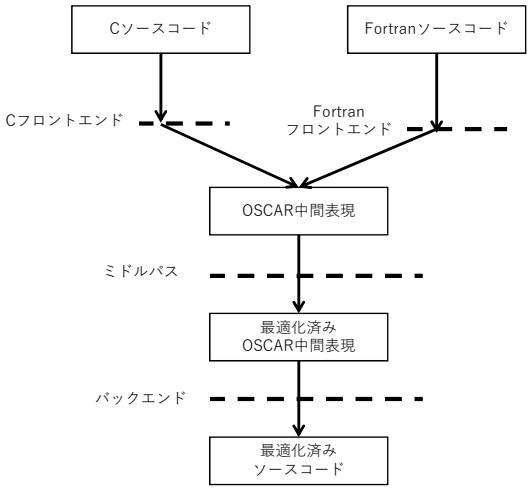


図 1　OSCAR コンパイラのコンパイルフロー

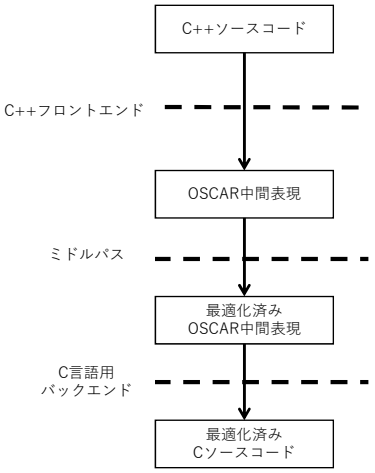


図 2　C++ プログラム自動並列化のためのコンパイルフロー
入力 C++ ソースコードを同等の最適化済み C コードへ変換

パイルフローを図 2 に示す．図 1 と同様にフロントエンド，ミドルパスバックエンドの 3 段構成となっている．本研究では C++ 用の OSCAR フロントエンド及び OSCAR ミドルパスにおける中間表現の解析を目標としており，バックエンドは従来の C 言語用のものを用いた．このため，本研究で提案するコンパイルフローは C++-to-C のコンパイルフローとなっている．

## 4. C++ フロントエンドの実装及びバックエンドにおける C コード生成

本節では C++ で記述されたプログラムを OSCAR 中間表現へ変換する C++ フロントエンドとバックエンドにおける C ソースコード生成について述べる．

### 4.1 実装の概要

本研究で作成した C++ フロントエンドは Clang コンパイラをベースとして実装を行った．Clang とは LLVM ライブラリを用いて設計されたオープンソースのコンパイラである [12], [13]．本研究のための C++ フロントエンドは Clang の字句解析，構文解析，意味解析の部分はそのままに，解析結果である抽象構文木 (Abstract Syntax Tree, AST) を LLVM の中間表現へ変換する部分を OSCAR の中間表現へ変換するものに取り替える形で実装を行った．実装の概略図を図 3 に示す．

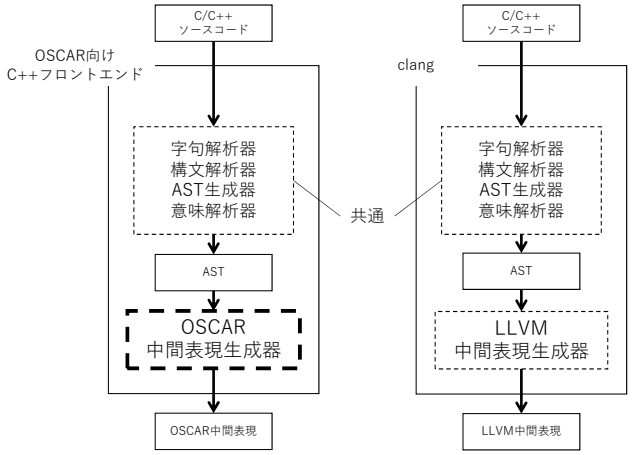


図 3　Clang ベース,OSCAR コンパイラ向け C++ フロントエンドの実装概要

Clang/LLVM の中間表現としては LLVM-IR [12] が広く用いられているが，LLVM-IR はアセンブリに近い低レベルな中間表現であり，自動並列化のための解析や source-to-source コンパイラとして並列化 C/C++ プログラムを生成するために必要なクラス・構造体や配列等のソースコードレベルの多くの情報が失われている．そのため，本研究では AST から OSCAR コンパイラ用の中間表現に変換することをフロントエンドの設計として選択した．

開発したフロントエンドでは AST 生成までの部分は Clang と同じものを用いており，太い点線で示した部分が本研究で新たに実装した箇所となっている．Clang が Clang-AST を LLVM 中間表現へ変換する部分を，Clang-AST から OSCAR 中間表現へ変換するモジュールへ置き換えることで OSCAR 向け C++ フロントエンドを構成している．このような実装を行うことで Clang/LLVM が持つ高度な C++ プログラム解析機能を再利用しながら，実装コストを削減することができる．

### 4.2 クラス継承関係の出力

C++ フロントエンドは後段の OSCAR ミドルパスに C++ プログラムを解析させるために，プログラム内で定義されたクラスの継承関係を中間表現の一部としてクラス毎に出力する．出力する情報の例を図 4 に示す．図の各小円はクラス，矢印は継承関係をそれぞれ表している．このときに出力する情報には着目するクラスの直下の子クラスに関する情報のみが出力される．2 段以上遠くの継承元に関する情報や兄弟クラスに関する情報は生成しない．図 4 を用いると，例えば B が A を継承するという情報や B が C と D を継承先にしているという情報は出力されるが，C が A の孫クラスであるという情報や C と D が共通のクラスを継承しているといった情報は出力しない．クラス継承木全体は，中間表現を読み取ったミドルパスが内部で再構成する．

### 4.3 テンプレートの解決

C++ は関数テンプレートによりジェネリックプログラミン

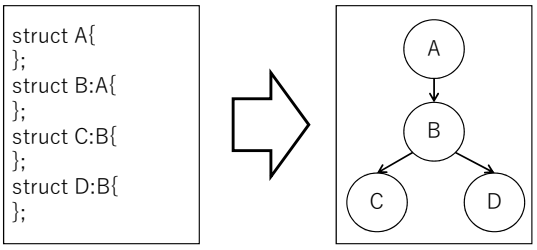


図 4　OSCAR 用 C++ フロントエンドが出力するクラス継承関係

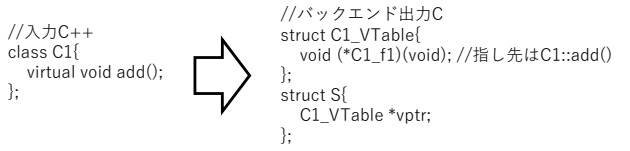


図 5　C++ ソースコードを入力した際の OSCAR C バックエンド出力の擬似コード

グの機能を提供している．関数テンプレートを用いることで 1 つの関数定義を複数の型に対応させることができる．しかし，コンパイラ内部では抽象化された型情報のまま関数の解析を行うことはできない．そのため，本研究で開発した C++ フロントエンドは Clang-AST が解析した情報を用いてテンプレートを解決する．具体的には関数テンプレートで定義された関数について，プログラム中でそのテンプレートのテンプレート引数に与えられる型をすべて集める．その後関数定義を，集めた型の分だけテンプレートを使わない形で列挙する．さらにその後関数テンプレートが使われた関数の呼び出しを新たに定義した関数へ型が合うように置き換える．

### 4.4　バックエンドにおける C ソースコード出力

本節では C++ ソースコードから生成した OSCAR 中間表現を C バックエンドで C ソースコードとして出力する手法について述べる．3. 節で述べた通り，本研究では C++-to-C のコンパイルフローを提案した．元の C++ ソースコードを C ソースコードとして出力する場合，C++ ソースコードに記述されたクラス関数を含んだ構造体定義 (クラス定義) を C 言語の構造体定義の記述に落とし込む必要がある．そこで本研究では Itanium C++ ABI [15] を参考に，その動作が実現できる C コードを生成できるように C バックエンドを拡張した．提案するコンパイルフローに通した際の C++ 入力と C 言語出力の擬似コードを図 5 に示す．クラス内のクラス関数定義を，クラス関数本体を指す関数ポインタが並んでいる構造体に変換することによって C 言語の記述で入力 C++ ソースコードと同等の表現を実現している．

### 4.5　制 約 事 項

本研究で新しく作成したフロントエンドは既存の Clang コンパイラと Clang-AST を用いることで C++ の機能の多くを OSCAR 中間表現へ変換することが可能となっている．しかしながら，例外処理は本研究で作成したフロントエンドではサポートしていない．これは一般に，例外を投げる可能性のあるブロックについては高性能を求められることが少ないためである．

```
struct A{
   virtual void sub(){
      int xa=10;
   };
};
struct B:A{
   virtual void sub(){
      int xb=20;
   };
};
void vcaller(A *x){
   x->sub();
}
```

図 6　仮想関数オーバーライドが用いられた C++ コードの例

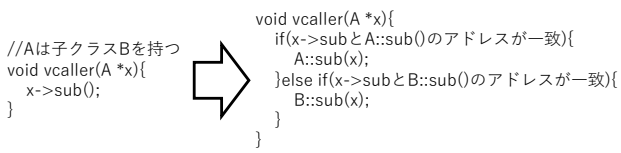


図 7　仮想関数オーバーライドが用いられた C++ コードの変換例
左が入力 C++ ソースコード，右が OSCAR コンパイラの出力 C ソースコードをそれぞれ表している

また，RTTI も自動並列化に必要なプログラム静的解析を妨げるため，現時点ではサポート対象外とした．

## 5.　C++ プログラム解析手法

本節では OSCAR C++ フロントエンドの出力から OSCAR ミドルパスを用いて C++ プログラムを解析する手法について述べる．

### 5.1　従来の問題: 仮想関数オーバーライド

クラスの継承は C++ のプログラムコード再利用性の向上のために重要な機能の 1 つである．しかしながら，クラスの継承とともに利用される仮想関数のオーバーライドはコンパイラによるプログラム解析を困難にする．仮想関数オーバーライドが用いられたコードの具体例を図 6 に示す．図 6 のコードにはクラス関数 sub を持つクラス A とクラス関数 sub をオーバーライドしているクラス B とクラス関数 sub を呼び出す関数 vcaller がそれぞれ定義されている．また，関数 vcaller はクラス A 及び，クラス A の子クラスであるクラス B を引数として受け取ることができる．このとき，関数 vcaller の動作は以下の 2 パターンが存在する．

（1）　クラス A のメンバ関数 sub を呼び出す
（2）　クラス B のメンバ関数 sub を呼び出す

2. 節で述べた通りコンパイラによってコードの静的解析を行うには関数呼び出し文において呼び出される関数が一つに決まっていなければならない．しかし，図 6 の x->sub(); という文は呼び出す関数本体が vcaller の引数によって変わるため従来の方法では解析が難しい．

### 5.2　提 案 手 法

本研究では図 6 に挙げた仮想関数呼び出しをコンパイラによって解析・自動並列化が可能な形に変換する手法を提案する．変換後のコードの形状は Calder らの先行研究 [16] を参考にした．図 7 に擬似コードを示す．図 7 の左側は入力の C++ ソー

スコード，右側は提案手法を実装した OSCAR コンパイラの出力 C ソースコードである．変換までの手順を以下に示す．

（1） C++ フロントエンドで図 4 に示すようなクラス継承関係を解析

（2） OSCAR ミドルパスで仮想関数の呼び出しを関数アドレス比較と通常の関数呼び出しを組み合わせた if-else 文に置換

上記のようにプログラムを変換することで，仮想関数呼び出しにより呼び出される可能性のある関数をプログラム中に明示的に表現する．これにより，仮想関数呼び出しのある C++ プログラムに対しても従来の OSCAR コンパイラの解析器が利用可能になり，OSCAR コンパイラの変更部分を最小化できる．

## 6. 性能評価

本節では 4. 節で述べた OSCAR コンパイラ用 C++ フロントエンドと 5. 節で述べた C++ プログラム解析手法の性能評価方法と性能評価結果について述べる．

### 6.1 評価アプリケーション及び評価環境

SPEC CPU 2006 [2] のベンチマークプログラムの一つである 447.dealII の CG ソルバ部を基にしたテストプログラムを用いて提案手法の評価を行った．テストに用いたプログラムの擬似コード図 8 に示す．テストプログラムは仮想関数 vmult を持った matrix クラスとそれを継承した sparse_matrix クラス及び dense_matrix クラスの 3 つが定義される．2 つの子クラスは継承した仮想関数 vmult をオーバーライドすることで，自身の持つデータ構造に合致した行列ベクトル積計算を行うようになっている．さらに，テストプログラムには仮想関数 vmult を呼ぶ関数 solve が定義される．関数 solve には関数テンプレートが用いられており，引数として matrix, sparse_matrix, dense_matrix を受け取ることができる．これは 447.dealII 内部の CG ソルバに用いられる主要な計算部と同じ構造となっている．

本テストプログラムの解析には以下の 2 点に困難さが存在する．

（1） 関数 solve の引数の型が 3 通り存在する点

（2） 同名の関数 vmult が複数存在するため，関数呼び出し文 m->vmult(); で呼ばれる関数が複数通り存在する点

問題 1 については C++ フロントエンドにおける関数テンプレートの解析によって解決する．また，問題 2 については C++ フロントエンドが出力するクラス継承関係とそれを用いたミドルパスによる解析で解決する．

評価環境は以下の通りである．

- OS: Ubuntu 16.04.3 LTS
- CPU: Intel Xeon CPU X5670 @ 2.93GHz 6 コア 12 スレッド
- メモリ: 49.4GB

本評価では物理 6 コアを搭載した Intel サーバーマシン上で評価を行った．評価は最大で 6 コアを用いて行った．

### 6.2 評価結果

評価結果を図 9 に示す．提案手法を適用した OSCAR 自動並列化コンパイラを用いて，g++ 7.4.0（-O2 により最適化）を用いた逐次実行と比較して 2 コア並列実行で 1.62 倍，4 コア並列実行によって 2.61 倍，6 コア並列実行によって 3.27 倍の速度向上が確認できた．また，sparse_matrix::vmult の計算速度は 6 コアで 5.83 倍，dense_matrix::vmult の計算速度は 6 コアで 4.70 倍の向上を確認できた．

以上の結果から本稿で述べてきた，C++ で記述されたプログラムに特徴的な仮想関数と関数テンプレートを用いたプログラムに対しても並列性抽出が可能であることが確認できた．

```
template <class MATRIX>
void solve(MATRIX *m){
        m->vmult();
        return;
}
```

```
struct matrix{
  virtual void vmult(){
    何もしない
  };
};
struct sparse_matrix:matrix{
  virtual void vmult(){
    疎行列 × ベクトル
  };
};
struct dense_matrix:matrix{
  virtual void vmult(){
    密行列 × ベクトル
  };
};
```

図 8 評価に用いたプログラムの擬似コード
仮想関数 vmult は疎行列用と密行列用がそれぞれ定義される
vmult の呼び出し部では関数テンプレートが用いられる

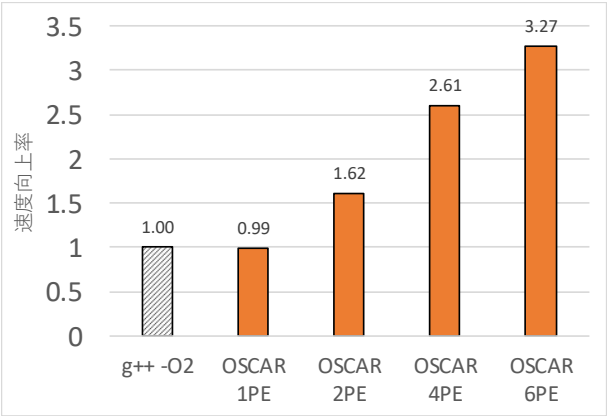


図 9 SPEC CPU 2006 dealII の主要計算部を元にしたテストプログラムによる評価結果

## 7. 関連研究

Clang ベースのフロントエンドに関連した研究としては Frama-C [9] の Frama-Clang [10] が挙げられる．Frama-Clang は本研究と同様に C++ ソースコードから生成された Clang-AST から Frama-C 独自の中間表現への変換を行っている．

5. 節で述べた仮想関数解析に関連した研究としては Java でのメソッド呼び出しの解析が挙げられる [11]．この研究は Java ソースコードのクラス階層を解析してメソッド呼び出し文に対してインライン展開を行い，動的なメソッド呼び出しによるオーバーヘッドの削減を目的としている．

## 8. まとめ

本稿では C++ で記述されたプログラムを自動並列化コンパ

イラで解析するための Clang ベースフロントエンドと C++ の重要な機能であるクラス継承，関数オーバーライド，関数テンプレートを解析する手法を提案した．本手法によって自動並列化に必要なプログラムのコールグラフ生成が可能となった．本稿での提案手法を SPEC CPU 2006 に含まれる 447.dealII の CG ソルバ部を基にしたテストプログラムを用いて 6 コアの Intel サーバー上で評価を行った．その結果，g++ を用いた逐次実行と比較して，6 コアで 3.27 倍の速度向上を確認した．

## 文　献


[1] D. Hardy, "NAMD - Scalable Molecular Dynamics," University of Illinois Urbana-Champaign, https://www.ks.uiuc.edu/Research/namd/ 参照 Jan. 17. 2020.

[2] D. Rice, "SPEC CPU® 2006," Standard Performance Evaluation Corporation, https://www.spec.org/cpu2006/ 参照 Jan. 17. 2020.

[3] S. Kumar, Chao Huang, G. Almasi and L. V. Kale, "Achieving strong scaling with NAMD on Blue Gene/L," Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, Rhodes Island, 2006.

[4] H. Kasahara, H. Honda, K. Aida, M. Okamoto, A. Yoshida and W. Ogata. "OSCAR Fortran Multigrain Compiler," Parallel Language and Compiler Research in Japan. Springer, Boston, MA, 1995.

[5] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako and H. Kasahara, "OSCAR API for Real-time Low-Power Multicores and Its Performance on Multicores and SMP Servers", Lecture Notes in Computer Science, Springer, Vol. 5898, pp. 188-202, 2010.

[6] K. Yamamoto, T. Shirakawa, Y. Oki, A. Yoshida, K. Kimura, H. Kasahara, "Automatic Local Memory Management for Multicores Having Global Address Space", 29th International Workshop on Languages and Compilers for Parallel Computing(LCPC), Sep. 2016.

[7] T. Hirano, H. Yamamoto, S. Iizuka, K. Muto, T. Goto, T. Wake, H. Mikami, M. Takamura, K. Kimura, H. Kasahara, "Evaluation of Automatic Power Reduction with OSCAR Compiler on Intel Haswell and ARM Cortex-A9 Multicores", The 27th International Workshop on Languages and Compilers for Parallel Computing(LCPC), Sep. 2014.

[8] Intel, "Intel® Fortran Compiler | Intel® Software," Intel, https://software.intel.com/en-us/fortran-compilers, 参照 Jan. 17. 2020.

[9] K. Florent, K. Nikolai, P. Virgile, S. Julien and Y. Boris. "Frama-C: A software analysis perspective," Formal Aspects of Computing. 2015, vol. 27, no. 3, pp. 573–609, 2015.

[10] P. Baudin, "The Frama-Clang plugin", https://frama-c.com/frama-clang.html 参照 Jan. 17. 2020.

[11] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu and T. Nakatani. "A study of devirtualization techniques for a Java Just-In-Time compiler," ACM SIGPLAN Notices. vol. 35, no. 10, pp. 294–310, 2000.

[12] "The LLVM Compiler Infrastructure Project," https://llvm.org/, 参照 Jan. 17. 2020.

[13] "Clang C Language Family Frontend for LLVM," http://clang.llvm.org/, 参照 Jan. 17. 2020.

[14] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka and H. Kasahara (2005) "Hierarchical Parallelism Control for Multigrain Parallel Processing," Languages and Compilers for Parallel Computing. LCPC 2002. Lecture Notes in Computer Science, vol 2481. Springer, Berlin, Heidelberg

[15] "Itanium C++ ABI," https://itanium-cxx-abi.github.io/cxx-abi/abi.html.

[16] B. Calder and D. Grunwald. "Reducing Indirect Function Call Overhead In C++ Programs," In Proceedings of the ACM SIGPLAN ' 94 Symposium on Principles of Programming Languages, pp. 397-408, 1994.