Tightly Coupled Accelerators アーキテクチャに向けたXcalableMP拡張

中尾 昌広,[1] 村井 均,[1] 下坂 健則 [1]
田渕 晶大,[2] 塙 敏博,[3] 児玉 祐悦 [2,4]
朴 泰祐,[2,4] 佐藤 三久 [1,2,4]

1. 理化学研究所 計算科学研究機構
2. 筑波大学大学院 システム情報工学研究科
3. 東京大学 情報基盤センター
4. 筑波大学 計算科学研究センター

第143回HPC研究会@和倉温泉，2014年3月

研究背景

- GPUクラスタ
  - 高性能・高いエネルギー効率・低価格
  - 様々なHPCアプリケーションで用いられている
- TCA (Tightly Coupled Accelerators)
  - 密結合並列演算加速機構
  - アクセラレータ（GPU）間の直接通信．低レイテンシ
    - 今後のHPCアプリは強スケーリングも重要
- TCAとアクセラレータを搭載したシステムにおけるプログラミングモデル
  - 例：(CUDA or OpenACC) + MPI + TCAのAPI

筑波大学HA-PACSクラスタ

TCAのインタフェースボード
PEACH2

➡ 他の選択肢として高生産性なプログラミングモデルも用意しよう

研究目的

- 本研究の最終目標
  - TCAとアクセラレータを搭載したクラスタのための高生産プログラミングモデルの提案

- 本発表の研究内容
  - 上記のプロトタイプの提案とその初期評価
    - 並列言語XcalableMPを拡張し，TCAの利用に必要な処理の自動化
    - 拡張したXcalableMPとOpenACCの連携により，TCAとアクセラレータを搭載したクラスタに対するプログラミングの簡易化
    - ベンチマークプログラムを用いた生産性と性能の初期評価

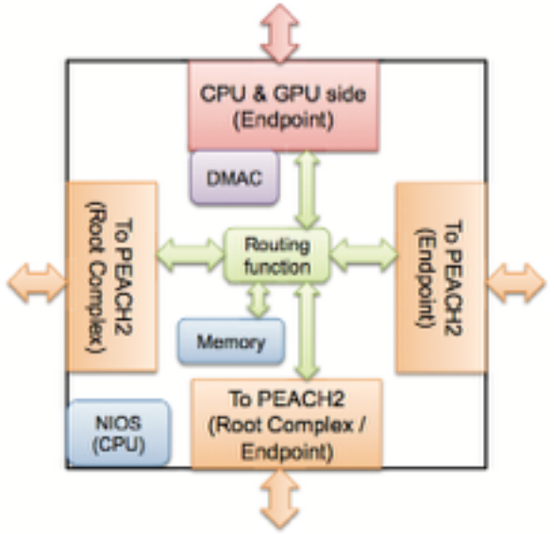
この後の発表の流れ

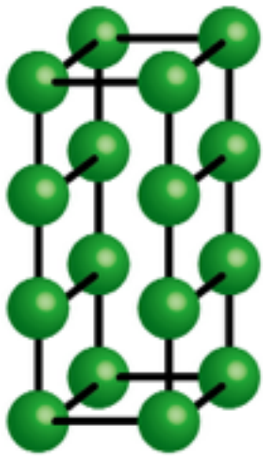
---

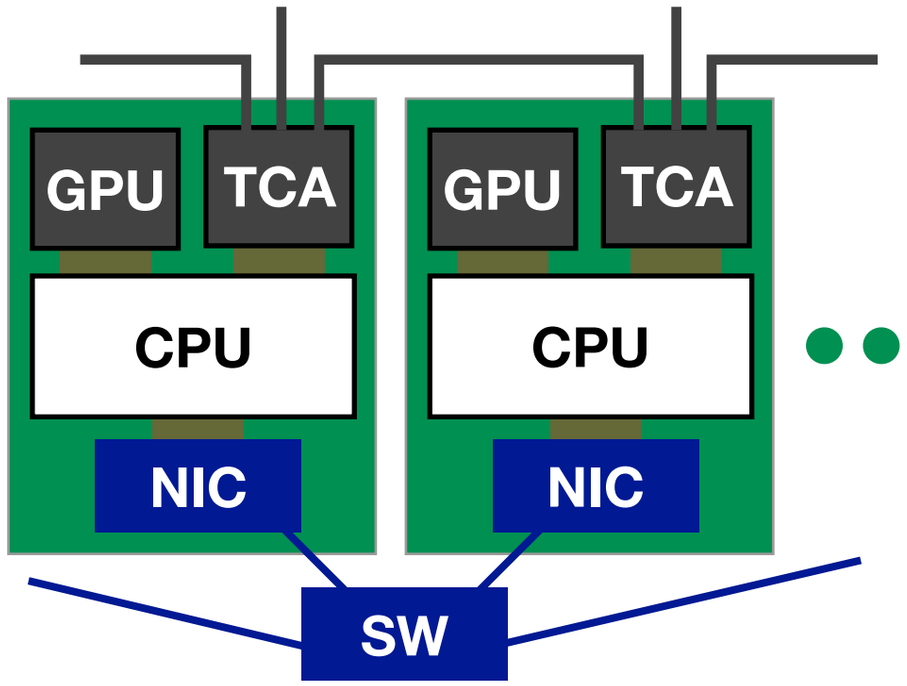
1. TCAを用いたプログラミング
2. プログラミングモデルのプロトタイプの提案と実装
3. ベンチマークプログラムによる生産性と性能の初期評価
    - ラプラス方程式
    - 地震波シミュレーションコード（FDTD法）
4. 考察と関連研究
5. まとめと今後の課題

TCAの詳細

- PCIeの技術を応用
- PEACH2
  - TCA用インタフェースボード
  - HA-PACS/TCA@筑波大で利用
  - Altera社Stratix IV GX（FPGA）
    - PCIe Gen2 x8レーン x4ポート
  - PEACH2同士を
    PCIe外部ケーブルで相互接続
- 16ノードのサブクラスタ
- ステンシル計算に向いている







●：計算ノード

ー：PCIe外部ケーブル

TCAを使ったプログラミング

- TCAが提供するAPIを用いて下記の操作を行う

  1. GPU上にメモリ領域を確保する
     - OpenACCのdata構文 or cudaMalloc() or tcaMalloc()など
  2. 1.の領域のアドレスを用いてTCAのハンドルの生成
  3. 送信ノードと受信ノードとでTCAハンドルの交換
  4. 送信データの登録（送信先，転送データサイズ，ストライドの幅など）
  5. データ送信

これらの操作は低レベルのAPIを用いる

この後の発表の流れ

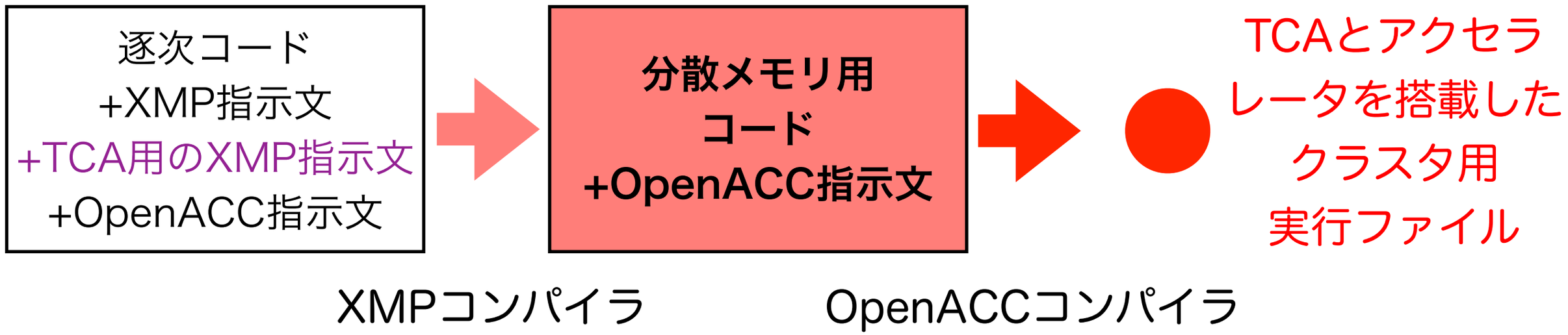
1. TCAを用いたプログラミング
2. プログラミングモデルのプロトタイプの提案と実装
3. ベンチマークプログラムによる生産性と性能の初期評価
    - ラプラス方程式
    - 地震波シミュレーションコード（FDTD法）
4. 考察と関連研究
5. まとめと今後の課題

OpenACCとXcalableMP

- OpenACC
  - 指示文ベースのアクセラレータプログラミングモデル
  - 逐次コードのイメージを保ったまま，処理をオフロード可能
  - 単体ノード用
- XcalableMP（XMP）
  - 指示文ベースの分散メモリ型システム用並列プログラミングモデル
  - 逐次コードのイメージを保ったまま，並列アプリケーションを作成可能
  - ステンシル計算を簡易に記述するための構文も提供

OpenACCとXMPを組合せ、さらにステンシル計算にTCAを利用するように
XMPを拡張することで，逐次コードのイメージを保ったまま，TCAと
アクセラレータを搭載したクラスタ用のプログラミングが行えるのではないか！！

提案プログラミングモデルのイメージ



- TCA用のXMP指示文
  - XMPのステンシル計算用の機能をTCA用に拡張
  - TCAの利用に必要なハンドルの操作などを自動的に行う

提案モデルのプログラム例

```
double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
#pragma xmp nodes p(x, y)
#pragma xmp template t(0:YSIZE-1, 0:XSIZE-1)
#pragma xmp distribute t(block, block) onto p
#pragma xmp align [j][i] with t(i, j) :: u, uu
#pragma xmp shadow uu[1:1][1:1]
…
#pragma acc data copy(u) copyin(uu)
{
  for(k=0; k<MAX_ITER; k++){
#pragma xmp loop (y,x) on t(y,x)
#pragma acc parallel loop collapse(2)
   for(x=1; x<XSIZE-1; x++)
      for(y=1; y<YSIZE-1; y++)
        uu[x][y] = u[x][y];

#pragma xmp reflect_tca (uu)

#pragma xmp loop (y,x) on t(y,x)
#pragma acc parallel loop collapse(2)
    for(x=1; x<XSIZE-1; x++)
      for(y=1; y<YSIZE-1; y++)
        u[x][y] = (uu[x-1][y]+uu[x+1][y]+
                   uu[x][y-1]+uu[x][y+1])/4.0;
  } // end k
} // end data
```

- 2次元ラプラス方程式

袖を含んだ分散配列の定義

分散配列をアクセラレータのメモリに転送

XMP指示文で分散された配列をOpenACC指示文が分散して処理

TCAを用いた配列uuの袖の同期

実装について（1/2）

```
double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
#pragma xmp nodes p(x, y)
#pragma xmp template t(0:YSIZE-1, 0:XSIZE-1)
#pragma xmp distribute t(block, block) onto p
#pragma xmp align [j][i] with t(i, j) :: u, uu
#pragma xmp shadow uu[1:1][1:1]
…
#pragma acc data copy(u) copyin(uu)
{
  for(k=0; k<MAX_ITER; k++){
#pragma xmp loop (y,x) on t(y,x)
#pragma acc parallel loop collapse(2)
    for(x=1; x<XSIZE-1; x++)
      for(y=1; y<YSIZE-1; y++)
        uu[x][y] = u[x][y];

#pragma xmp reflect_tca (uu)

#pragma xmp loop (y,x) on t(y,x)
#pragma acc parallel loop collapse(2)
    for(x=1; x<XSIZE-1; x++)
      for(y=1; y<YSIZE-1; y++)
        u[x][y] = (uu[x-1][y]+uu[x+1][y]+
                   uu[x][y-1]+uu[x][y+1])/4.0;
  } // end k
} // end data
```

- Omni XMP Compiler
  - AICSと筑波大で開発しているリファレンス実装
  - オープンソース
  - source-to-source
  - このコンパイラをベースに提案モデルを実装中

TCAを用いた配列uuの袖の同期

1. ハンドルの生成と共有
2. 配列の袖領域を登録
   - shadow文から情報入手
3. 1と2の情報をキャッシュ
4. 同期実行

実装について（2/2）

- OpenACCのdata指示文の分散配列の対応
  - OpenACCの仕様上，動的に確保された配列は開始番号と長さを指定する

```
int a[N];
#pragma xmp nodes p(4)
#pragma xmp template t(0:N-1)
#pragma xmp distribute t(block) onto p
#pragma xmp align a[i] with t(i)
 …
#pragma acc data copy (a)
{
 …
}
```

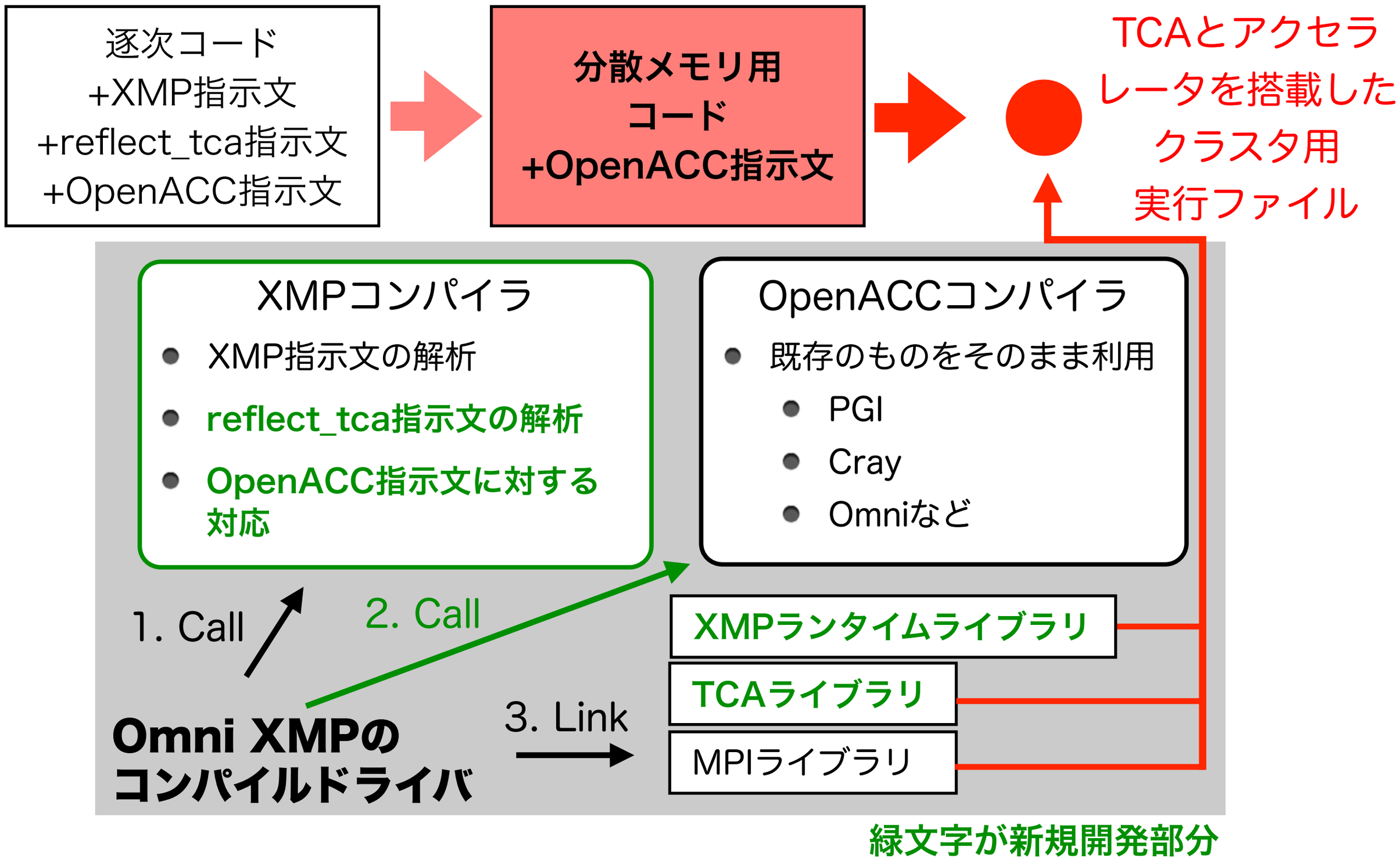
```
//
// int a[]は各ノードで必要な要素のみ
// malloc()で確保される
//
 …
int min = …
int length = …
#pragma acc data copy (a[min:length])
{
```

- OpenACCのloop指示文をfor文の直前に挿入する
  - 各ノードのイテレーションはXMPランタイムライブラリが計算しているから

```
#pragma xmp loop on t(i)
#pragma acc parallel loop
for(int i=0;i<N;i++){
 …
}
```

```
int XMP_init i, _XMP_cond_i, XMP_step_i;
XMP_sched_loop_template_BLOCK(0, N, 1, &(_XMP_init_i),
                      &(_XMP_cond_i), &(_XMP_step_i), …);
#pragma acc parallel loop
for(int i=_XMP_init_i;i<_XMP_cond_i;i+=_XMP_step_i){
```

Omni XMP Compilerの変更



緑文字が新規開発部分

この後の発表の流れ

1. TCAを用いたプログラミング
2. プログラミングモデルのプロトタイプの提案と実装
3. ベンチマークプログラムによる生産性と性能の初期評価
   - ラプラス方程式
   - 地震波シミュレーションコード（FDTD法）
4. 考察と関連研究
5. まとめと今後の課題

生産性と性能の初期評価

- ベンチマーク
  - 2次元ラプラス方程式
  - 地震波シミュレーションコード
    - 時間領域差分法（Finite-difference time-domain Method）
    - 逐次版は岡元太郎 先生（東工大）が作成
- 生産性と性能の評価方法
  - 提案モデルである「XMP+OpenACC」で記述したコードと「MPI+TCAのAPI+OpenACC」で記述したコードとの比較
  - 地震波シミュレーションコードについては，性能測定は行えなかったので，生産性のみの比較

2次元ラプラス方程式の生産性の比較

XMP+OpenACC（提案モデル）

```
double u[XSIZE][YSIZE]
double uu[XSIZE][YSIZE];
…

#pragma acc data copy(u) copyin(uu)
{
for(k=0; k<MAX_ITER; k++){
  …
#pragma xmp reflect_tca (uu)

#pragma xmp loop (y,x) on t(y,x)
#pragma acc parallel loop collapse(2)
  for(x=1; x<XSIZE-1; x++)
    for(y=1; y<YSIZE-1; y++)
      u[x][y] = (uu[x-1][y]+uu[x+1][y]+
            uu[x][y-1]+uu[x][y+1])/4.0;
   …
```

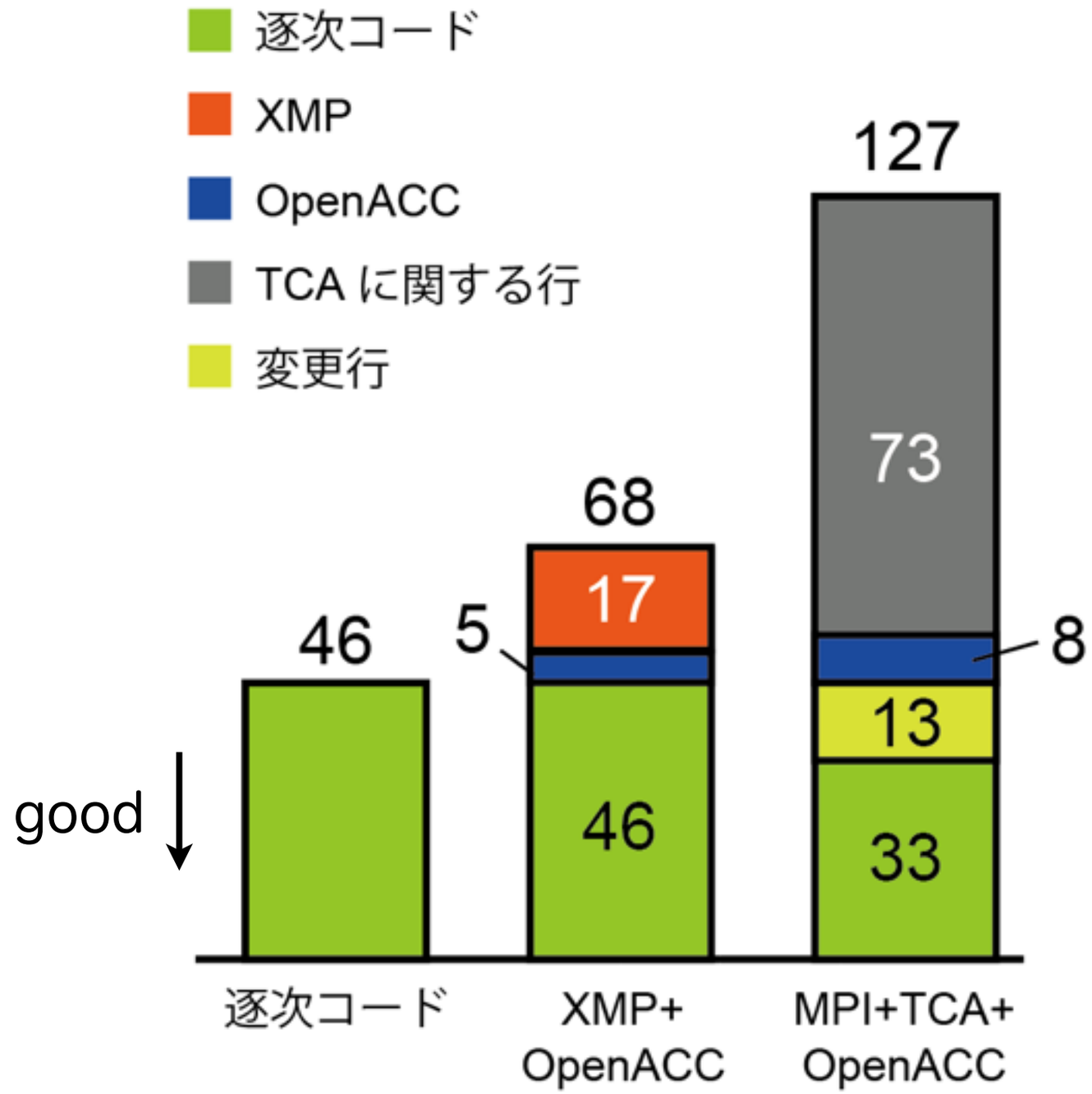
MPI+TCA+OpenACC

```
#define Local_XSIZE (XSIZE/X_PROCS+2)
double u[Local_XSIZE][Local_YSIZE]
double uu[Local_XSIZE][Local_YSIZE];
…
#pragma acc data copy(u) copyin(uu)
{
  // TCAのハンドルの登録など
  // 45行

for(k=0; k<MAX_ITER; k++){
  …
  // TCAを用いた袖送信とデータ受信完了待ち
  // 5行

#pragma acc parallel loop collapse(2)
  for(x=1; x<Local_XSIZE-1; x++)
    for(y=1; y<Local_YSIZE-1; y++)
      u[x][y] = (uu[x-1][y]+uu[x+1][y]+
            uu[x][y-1]+uu[x][y+1])/4.0;
   …
```

- 提案モデルは、配列の宣言やイテレーションなどを逐次のまま記述可能

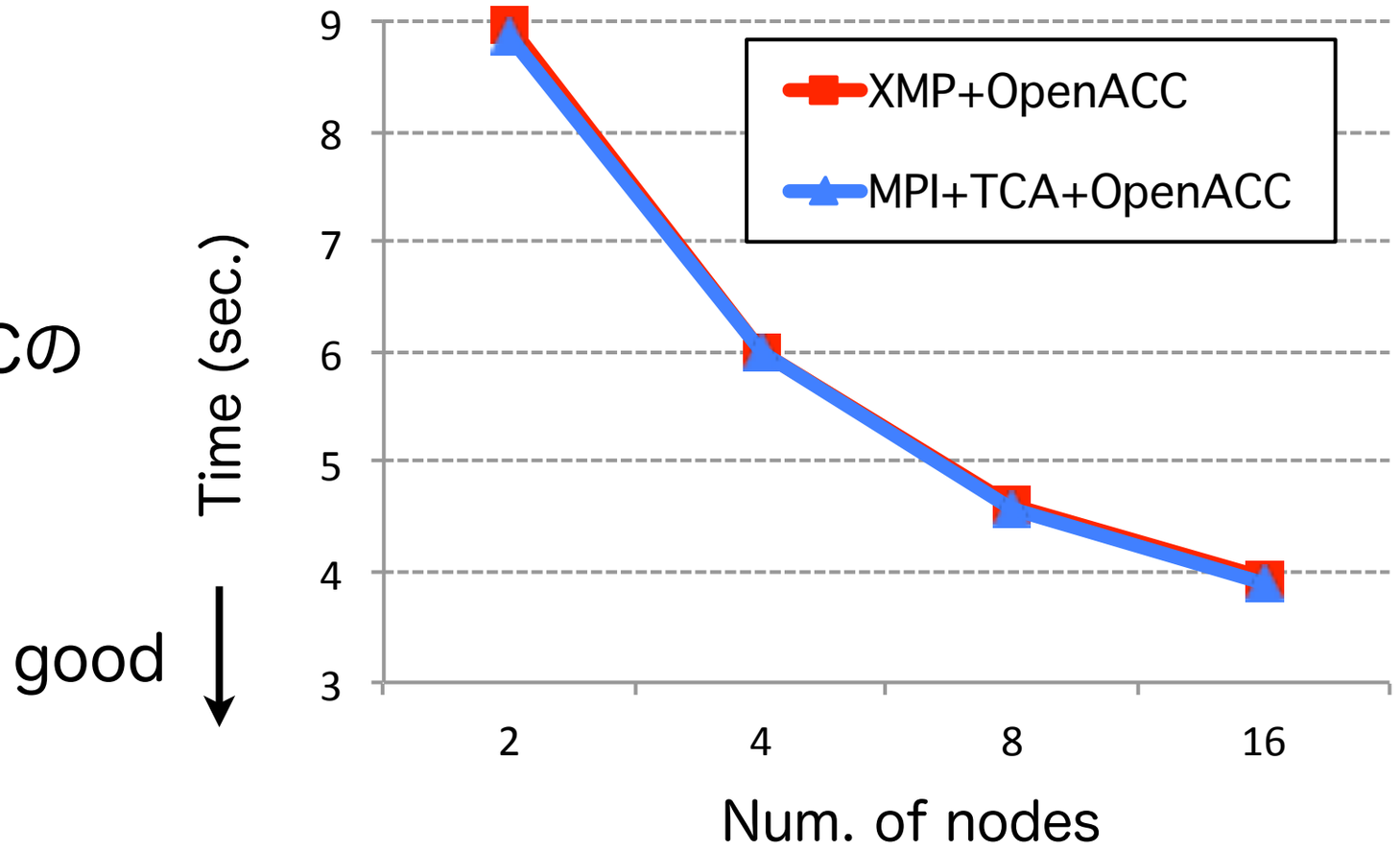
2次元ラプラス方程式に対する生産性

- 提案モデルはコードの追加のみで
  ハイブリッド化可能
  - 逐次コードはそのまま
  - 17+5 = 22行の追加のみ
  - 総行数は68行
- MPI+TCA+OpenACC
  - 73+8=81行の追加
  - 13行の既存コードの変更
  - 総行数は127行



2次元ラプラス方程式に対する性能

- 性能
  - HA-PACS/TCA
    - NVIDIA K20X
    - 1process/1node, 1GPU/1process
    - 問題サイズ：
      16384 x 16384
    - 強スケーリング
- 提案モデルは
  MPI+TCA+OpenACCの
  98.5%以上の性能
- 1.5%の差の原因は
  イテレーションの
  計算を時間発展内で
  毎回行っているから

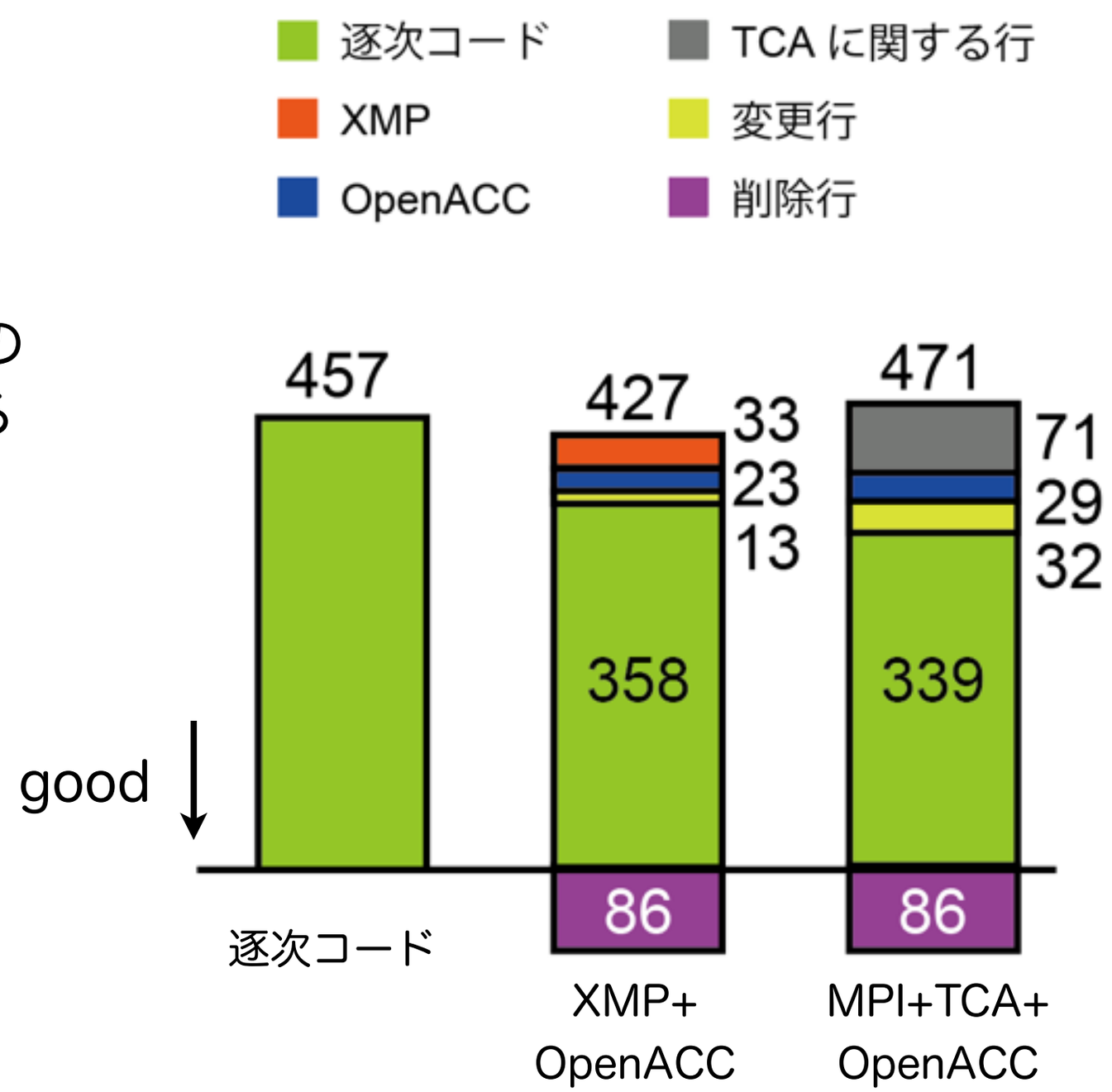


地震波シミュレーション

- オリジナルはFortran言語の逐次プログラム（東工大・岡元先生）
- C言語に変更
  - 行数：457行
- 同期配列数：9
- 問題空間は3次元で周期境界

地震波シミュレーションに対する生産性

- 提案モデル
  - 追加：33+23 = 56行
  - 変更：13行
  - 削除：86行
    - 周期境界のコードがTCAの通信で置き換えているから
  - 総行数は427行
- MPI+TCA+OpenACC
  - 追加：71+29 = 100行
  - 変更：32行
  - 削除：86行（理由は同上）
  - 総行数は471行



この後の発表の流れ

1. TCAを用いたプログラミング
2. プログラミングモデルのプロトタイプの提案
3. 提案モデルを実現するコンパイラの実装
4. ベンチマークプログラムによる生産性と性能の初期評価
   - ラプラス方程式
   - 地震波シミュレーションコード（FDTD法）
5. 考察と関連研究
6. まとめと今後の課題

考察

- 生産性について
  - 既存のコードの変更量は少ない
    - 逐次コードのイメージを保持
    - 端数処理などを含めた
      インデックス計算を自動化

    ➡ プログラミングコストの削減

  - TCAの処理の自動化
    - 同期対象の登録などのハンドルの
      操作を行う必要がない

    ➡ プログラミングコストと
    学習コストの削減

- 性能について
  - 追加コストはインデックス計算のみ
    - ループ文の内容が多かったり，ループ長が長ければ，相対的に
      追加コストは小さくなると考えられる

関連研究

- X10やChapelなどのGPU対応
  - ベース言語の文法の拡張
- XMP-dev
  - 同上
  - “PGAS並列プログラミング言語XcalableMPにおける演算加速装置を持つクラスタ向け拡張仕様の提案と試作”．ACS論文誌, 2010
- 上記と比較した今回の提案の利点
  - アクセラレータ用コードの最適化はOpenACCに任せられる
    - コンパイラの開発コストの削減
    - OpenACCのコードがあれば，TCAを用いた分散アプリケーションを少ない工数で開発可能

まとめと今後の課題

- 本研究の目的
  - TCAとアクセラレータを搭載したクラスタのためのプログラミング
    モデルのプロトタイプの提案と初期評価
- 明らかにしたこと
  - ラプラス方程式とFDTD法といったステンシル計算において，提案モデル
    は変更コードの量は少なく，性能はMPIを用いたモデルの98.5%以上を
    発揮した
- 今後の課題
  - TCAに関する高速化
    - 通信のpack/unpack
  - 他のXMP指示文（broadcastなど）のアクセラレータ対応の検討