自動ソースコード変換による PGAS プログラム最適化☆

根岸 康† 村田 浩樹† 森山 孝男†

Guojing Cong‡ I-Hsin Chung‡ Hui-Fang Wen‡ David Klepacki‡

Partitioned Global Address Space (PGAS) 言語が提供する分散共有配列により通信を明示的に扱う必要がなくなり，分散アルゴリズムのプログラミングが容易になった．一方，MPI と異なり通信がプログラム中に明示的に現れないために通信ボトルネックの解析・解消が困難になり，この作業を効率化するツールが求められている．本研究では，PGAS プログラムの通信ボトルネックを自動ソースコード変換により解消する手法を提案する．また，提案手法を最適化フレームワーク High Productivity Computing Systems Toolkit 上に実装したので，このプロトタイプ及び性能測定結果についても説明する． Connected Components (CC) アルゴリズムのプログラムの例では，提案手法に基づく自動最適化により元プログラムの約 5 倍の性能が得られた．これは，プログラマによる既存の最善の手動最適化プログラムと同等(1-8%の相違)の性能であった．

PGAS program optimization through source-to-source translation☆

YASUSHI NEGISHI,† HIROKI MURATA,† TAKAO MORIYAMA,†

GUOJING CONG,‡ I-HSHIN CHUNG,‡ HUI-FANG WEN,‡ and DAVID KLEPACKI‡

The Partitioned Global Address Space (PGAS) language provides the distributed shared array, which makes programming of distributed algorithms easier. On the other hand, unlike MPI, it becomes harder for programmers to identify and solve communication bottlenecks, because communication is not explicitly shown on programs. Our study proposes a method to solve it by automatic source-to-source translation. This paper also explains a prototype implemented on High Productivity Computing Systems Toolkit and the measurement results according to three example programs. In the Connected Components (CC) algorithm program case, the performance is improved by about five times, and achieves the same level (1-8 % difference) with the best prior manually tuned implementation.

1. はじめに

スーパーコンピュータの性能は急速に向上し，数値シミュレーションによって解決可能な問題の領域は大きく拡大している．一方，プロセッサの単体の性能向上は頭打ちとなり，システム性能の向上は

☆ This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

† 日本アイ・ビー・エム（株）東京基礎研究所
IBM Research – Tokyo.
‡ IBM T.J. Watson Research Center.

主にプロセッサ数の拡大や通信アーキテクチャの様々な工夫によって達成されている．このため，スーパーコンピュータのアーキテクチャは複雑になり，その複雑さに対応するためのプログラミングの生産性の向上が大きな課題となりつつある[1].

この生産性の問題のうち，プログラム開発時の効率向上に着目するのが Partitioned Global Address Space (PGAS)言語である．PGAS 言語は，プログラマの生産性向上のために(分散)共有配列の機能を提供する．プログラマはこの共有配列に通常のローカル配列と同様の手順でアクセスできる．共有配列の遠隔参照に必要な通信はコンパイラが自動的に生成する．これにより通信を意識せずに分散プログラミングが可能となり，並列プログラム開発時の生産性は向上する．

一方，HPC プログラムでは，開発時のみならず性能最適化の際の作業効率も大きな問題となっている．PGAS 言語では MPI 通信のように通信がプログラム中に明示されないため，通信に関連したボトルネックを発見・解決することは更に困難になる．現状，設置されたスーパーコンピュータ上で実際実行されている HPC プログラムを観察すると，特にコンパイラでの最適化が困難な通信や入出力性能が十分に最適化されているとは言えず，ソースコード最適化による最適化の余地が大きく残されているプログラムがしばしばみられる．しかし，ソースコード最適化には対象アーキテクチャに関する理解や性能ツールの使いこなし等のスキルが必要で，アプリケーション利用者がこの最適化を行うことは現実的ではない．一方，アプリケーション開発者が開発時に想定しない環境で十分な性能が得られなかったとしても開発者の責任とは言えない．

スーパーコンピュータのアーキテクチャが今後多様化していくことを考えると，今後このようなプログラムの最適化作業を効率化するためのツール・手法はより重要になっていくと考えられる．

本論文ではコンパイラや単純な最適化では解消が困難な PGAS 言語の共有配列への遠隔アクセスのオーバヘッドに着目し，これを自動ソースコード変換によるキャッシュメカニズムの導入によって解決する手法を提案する．また，High Productivity Computing Systems Toolkit[7]と呼ばれる性能最適化用ツール上に提案手法のプロトタイプを実現し，これを用いて Connected Components (以下 CC と呼ぶ)アルゴリズムのプログラムを提案手法に基づいて最適化し性能を測定したので，測定結果について考察する．

以下，第2章で PGAS 言語の性能上の課題について，第3章で関連する従来技術について，第4章で提案する最適化手法について，第5章で作成したプロトタイプについて，第6章で CC プログラムの最適化事例及びそれに基づく考察について，第7章で結論及び将来の課題について述べる．

2. PGAS 言語の性能上の課題

本章では UPC 言語を例として PGAS 言語の性能上の課題について説明する．

2.1 PGAS 言語とは

Partitioned Global Address Space (PGAS)は並列プログラミングのためのプログラミングモデルであり，このモデルをサポートする言語を PGAS 言語と呼ぶ．PGAS 言語には，既存の言語に PGAS モデルを追加した UPC[2][3], Co-array Fortran[4]や新規言語の Chapel[5]，X10[6]がある．

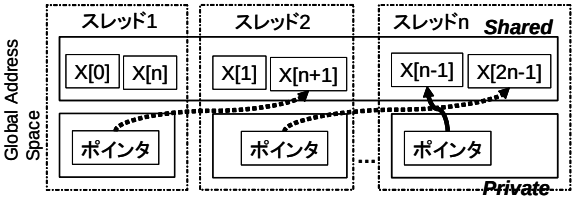

図 1: Partitioned Global Address Space

PGAS モデルは，全ノードで仮想的に共有される共有配列を提供する(図 1 参照)．図 1 は，各 Thread が個別に保持する Private 領域に置かれたポインタが，全てのスレッドが共有する Shared 領域に置かれた配列 X の要素をポイントする様子を示している．共有プログラマは通常のローカル配列と同様な手段でこの配列にアクセスできるので，OpenMP による並列化と同様，比較的少ないソースコードの変更で逐次実行プログラムを分散並列プログラムに変更することが可能となり，プログラミングの生産性を向上させる．

2.2 PGAS 言語の性能上の課題

PGAS 言語固有の性能上の課題として，共有配列への遠隔アクセスが挙げられる．PGAS 言語では，共有配列は各ノードに分散して配置されるが，そのメモリ配置はプログラマに委ねられる．プログラマはプログラムのメモリアクセス特性に合わせて，cyclic，block，block-cyclic 等のメモリ配置やルー

プの各 iteration の実行スレッド割当を指定する．共有配列へアクセスが規則的・固定的で，プログラマがその特性を理解している場合には，頻度の高い共有配列へのアクセスをノード内のメモリへのアクセスとするためのメモリ配置や iteration の実行スレッド割当の工夫により性能を向上できる．

しかし，問題によってはメモリアクセスが不規則で，メモリ配置の工夫だけでは共有配列へのアクセスをローカルアクセスとすることができない場合がある．例えば，後述の CC プログラムと呼ばれるグラフの断片数を数えるプログラムでは，辺と頂点のリストをそれぞれ共有配列として管理するが，辺と頂点は無作為に関連付けられるのでループの実行スレッド割当を片方(例えば辺)にあわせて指定すると，他方(頂点)へのアクセスについては，ローカルアクセスを保証できない．

3. 関連する従来技術

本章では，本研究に従来技術について説明する．

3.1 コンパイラによる最適化

コンパイラによる最適化は，最も重要であり，最適化に当たっての基礎となる．近年は，実行トレースを最適化に活用する研究が進められ，徐々に利用可能になっている[16]．

しかし，HPC 分野ではコンパイラによる最適化に加えて，プログラマが多くの時間をプログラムのソースコード書換えによる最適化に費やしている．その理由は，(1) コンパイラは単一プロセッサの性能最適化にフォーカスしており，通信やファイル入出力等に関する最適化は多くの場合最適化の対象とならないこと，(2)コンパイラがアーキテクチャの進歩をキャッチアップし，最適化が実装され，利用可能になるまでには時間がかかること，(3) HPC プログラムは高価なスーパーコンピュータ上で長時間実行されるため，少し（例えば 1%）の性能向上であっても経済的に大きな意味を持つことなどによる．

一方，実際には，アプリケーション利用者は対象アプリケーションを再コンパイルのみでプログラム書換えによる最適化をしないままに利用することも多い．コンパイラの最適化は通信や入出力性能については必ずしも十分ではなくプログラム書換えによる最適化が必要な場合が多いが，それには，対象アーキテクチャに関する理解や性能ツールの使いこなし等，多様なスキルが必要となり，アプリケーションの各利用者がプログラム最適化を行うことは現実的ではない．現状実際に実行されているアプリケーションの多くは十分に最適化されておらず，最適化の余地が大きく残されている．

我々の研究の目的は，このギャップを解消し，プログラマのソースコード書換えによる最適化作業に必要なスキルレベルを下げるとともに，その作業を効率化することにある．

3.2 自動最適化ライブラリによる最適化

プログラムの自動最適化のアプローチとして，自動最適化ライブラリがある[14][15]．これは BLAS 等特定のライブラリの性能上重要なパラメータを自動的に最適化するものである．

この手法の目的は，特定ライブラリの性能上のポータビリティーを高めることである．一方，本研究は自動ソースコード変換により，より広範囲のプログラムの性能最適化作業の支援を目的とする．

3.3 自動ソースコード変換技術

HPC プログラムの最適化作業には，ループアンローリング，標準通信 API(MPI)の呼出しから低レベルだが高速な通信 API の呼出しへの書換えのような手間のかかる定型的なソースコード変換が含まれるので，自動ソースコード変換によりプログラミング作業を支援するツールが有効である．このため，Photran[8]や ROSE[10]のようにこれを支援するツールが提案されている．その多くはソースコード可読性向上のためのリファクタリング作業支援[9]を主目的としているが，変換技術自身は HPC プログラムの最適化作業に適用できる．また，[10]や High Productivity Computing Systems Toolkit[7]のように HPC の最適化作業の支援に特化したツールも提案されている．

3.4 通信性能の最適化技術

HPC アプリケーションの通信最適化の研究は多数行われている．適用する複数通信の取りまとめによるパケット数の削減，非同期通信の導入による通信同士及び通信と計算との重ね合わせ[12]は通信性能の最適化の際にしばしば用いられる最適化手法であり，提案手法でも使用する．

4. 自動ソースコード変換を用いた最適化手法

本論文では，自動ソースコード変換を用いた最適

化手法を提案する．

4.1 自動ソースコード変換を用いた最適化

本研究の目的は，プログラマのソースコード書換えによる最適化作業に必要なスキルレベルを下げるとともに，その作業を効率化することにある．効率化に際しては，(1) ホットスポットの検出，(2) ボトルネックの発見，(3) 最適化方針の決定，(4) 最適化の適用の各手順についてサポートすることが必要となる．本論文では，これらの手順のうち，(4) の最適化の適用の手順を自動化することを目標とする．自動最適化の際は，プログラムのソースコードとホットスポット（ソースコード上の範囲）が入力として与えられるものとする．

4.2 適用する最適化手法

提案手法では，キャッシュを用いて共有配列への遠隔アクセスを最適化する．提案手法では，キャッシュを用いて共有配列への遠隔アクセスを最適化する．自動的な最適化は以下のステップで行う．

1. 共有配列への遠隔アクセスの検出
2. ソースコード変換の正当性確認
3. 自動ソースコード変換

以下，最適化方式について説明した後に，これらの各手順について順次説明する．

4.3 最適化の基本方針

基本方針

最適化に際し，共有配列への遠隔アクセスによるオーバヘッドに注目する．このオーバヘッドの原因は共有配列への遠隔アクセスに伴う「多数の小さなデータ」の「同期送受信」である．キャッシュを導入し，この通信を「少数の大きなデータ」の「非同期送受信」に変換する．これにより，データ取得にかかるコストを削減する．

期待される性能向上

変換により期待される性能向上は以下の通り．

1. メモリアクセスの集積による効率化
   多数の小さい遠隔メモリアクセスが少数の大きな通信に変換され，通信効率が改善される．
2. 非同期通信の導入による待ち時間の削減
   同期アクセスから非同期通信に変換され，複数ノードとの通信が並列に実行されるため待ち時間が削減される．
3. キャッシュ再利用による取得回数の削減
   同一データへの複数回アクセスで取得済みデータの再利用により，データ取得回数を削減する．

キャッシュ取得手順

最適化時の対象共有配列からのキャッシュへの値の取得，キャッシュの値の変更の元の共有配列への反映は，以下の手順で行う．以下，説明中で，対象配列の特定の index にアクセスするスレッドを使用スレッド，対象配列の特定 index を保持するスレッドを所有スレッドと呼ぶ．

1. 使用スレッドは対象ループ中の対象配列へのアクセス index をその所有スレッド毎に纏めた index 列を生成し，所有スレッドに非同期通信で送信する．
2. 所有スレッドは index 列を受信し，index 列中の各 index の対象配列の値を纏めた value 列を生成し，要求元の所有スレッドに非同期通信で送信する．
3. 使用スレッドは value 列を受信し，value 列中の値を配列の index をキーとするハッシュで実現されるキャッシュの対応する位置に保存する．

4.4 共有配列への遠隔アクセスの検出

提案手法では，最適化の候補となるループを選定するためにコード解析を行い，共有配列への遠隔アクセスを検出する．

UPC の upc_forall ループ文では，C の for ループ文の 3 つの引数に加えて，4 つ目の引数として affinity と呼ばれる各 iteration の実行スレッド決定用の情報を指定する．以下の例では i 番目の iteration は X[i]を保持するスレッドが実行するので，共有配列 X へのアクセスはローカルアクセスが保証される．一方，共有配列 Y へのアクセスはローカルアクセスが保証されない．

```
shared int X[10], Y[100];
upc_forall(i = 0; i < 10; i++; &X[i]) {
  X[i] = i;
  Y[i * 10] = i + 10;
}
```

コード解析では，ホットスポットを含むループ文の affinity とループ中コードの共有配列アクセス部分を解析して，各アクセスがローカルアクセスかどうかを判定し，ホットスポット中で遠隔アクセスが多数行われる共有配列を最適化の候補とする．

4.5 ソースコード変換の正当性確認

提案手法では，ソースコード変換を行う前に変換

の正当性について検証する．最初に UPC 言語のメモリ一貫性モデル([2]Appendix B 参照)について検討する．UPC 言語では，共有配列（変数）のアクセスに当たって以下の 3 つのセマンティクスのいずれかが適用される．

1. local semantics
自スレッド上に配置された共有配列の場合, 常にこのセマンティクスが適用される．UPC 言語は C 言語の通常の配列と同じメモリ一貫性モデルを保証する. すなわち配列への書込みは直後の読込みに反映される．
2. strict semantics
プラグマ等で strict semantics が指定された共有配列への遠隔アクセスに適用される. 共有メモリへの全ての書込みは他 Thread の直後の読込みに反映されることを保証する．
3. relaxed semantics
プラグマ等で relaxed semantics が指定された共有配列への遠隔アクセスに適用される. 配列への書込みは以後の upc_barrier 文の実行まで, 他 Thread の読込みに反映されることを保証しない.

提案手法によるソースコード変換はメモリ一貫性モデル上以下のいずれかの場合に許容される．

1. 共有配列の全値が変換対象ループ中で不変である場合
この場合, 配列への書込みがないので, キャッシュ導入に伴う問題は発生しない.
2. 変換対象コードの semantics が relaxed semantics である場合
ローカルアクセス時の local semantics，遠隔アクセス時の relaxed semantics の保証が必要となる. ローカルアクセス時の local semantics は，キャッシュを index をキーとするハッシュ表で管理することにより保証される. またリモートアクセス時の relax semantics は，upc_barrier 文実行時にローカルなキャッシュへの書込みを他の Thread の読込みに反映させることにより保証される．

4.6 自動ソースコード変換

本手法のソースコード変換の手順は以下の通り．全てのソースコード変換は定型的なものであり, 比較的単純な手法でプログラム可能である．

1. 宣言部
最適化に必要な一時変数の宣言を関数の先頭に挿入する．
2. アクセスヒント部
最適化対象ループの直前にこのループから最適化対象配列へのアクセスのみを取り出したドライループを生成し, そのループ中でアクセス予定の index を指定する．
3. アクセス部
最適化対象ループ中の最適化対象配列への全てのアクセスをキャッシュの参照･更新操作呼び出しに変換する．
4. 同期部
最適化対象ループの前後，及びループ中の upc_barrier 文の直後にキャッシュ同期操作の呼び出し文を挿入する．

4.7 従来のプログラマによる最適化手法との相違点

[13]は UPC で記述されたプログラムをプログラマの手動によるソースコード変更に基づいて最適化する手法について述べている．本論文は4.2節で述べた自動最適化適用時のステップのうち, 自動ソースコード変換ステップで[13]が提案する最適化の基本方針と同様の手法を採用しているが, その具体的な手法は[13]とは異なる．

異なる手法を採用する主な原因は[13]の手法が自動ソースコード変換の手法として用いるには以下の点で不都合があることによる．

1. 書換え範囲
[13]の手法は特定のプログラムへの適用を目的としており, プログラムの比較的広範囲に対象プログラムに依存した非定型な書換えが加えられている. 最適化のための書換えは, 最小かつ定型的であることが望ましい. 本論文の提案手法によるプログラム書換えの範囲は小さく, 定型的な変形によって実現される．
2. 最適化の適用条件
[13]はキャッシュの管理に通常の配列を使用し，取得データを iteration 順に保存している．このためデータの書換えはローカルアクセスであっても直後の読込みに反映されず，UPC の local semantics が守られない(4.5節参照)．[13]のプログラムは local semantics が守られなくても正しく動作するが, 他のプログラムへの適用にあたっては制約となる．本論文の提案手法では，index をキーとするハッシュ表によりキャッシュを管理するので local semantics が保証される．
3. 性能向上の条件
2で述べたように[13]では取得データを iteration

順に保存している．[13]では大きな問題とならないが，同一 index への複数回の配列アクセスでキャッシュを再利用しない．これは同一 index の配列アクセス回数が多いアプリケーションの場合問題となる．一方，本論文の提案手法では，同一 index のデータは単一のメモリに保存され，取得も 1 回のみで実現される．

5. プロトタイプ

我々は提案手法を High Productivity Computing Systems Toolkit[7]にソリューションルールとして AIX®オペレーティングシステム上に実現した．本章では，この実装方式について説明する．

5.1 最適化ライブラリ

ソースコード変換の際，最適化のためのソースコードの変更の量を最小化するために，キャッシュ管理ライブラリを導入する．図 2 にこの API を示す．

管理用 API

```
ac_hdl ac_open(shared void *var,size_t size,
                int num);
```
指定された配列用のキャッシュハンドラの作成
(size: 配列の要素サイズ，num: キャッシュ数)

```
int ac_close(ac_hdl hdl);
```
指定されたキャッシュハンドラの解放

```
int ac_accesshint(ac_hdl hdl, int idx);
```
アクセス予定 index の指定

```
int ac_accesshintclear(ac_hdl hdl);
```
全てのアクセス予定 index の設定解除

同期用 API

```
int ac_cachestart(ac_hdl hdl);
int ac_cachestop(ac_hdl hdl);
```
キャッシュ利用の開始・終了

```
int ac_cachesync(ac_hdl hdl, int mode);
```
配列のキャッシュの同期

データアクセス用 API

```
int ac_get(ac_hdl hdl, int idx, void *val);
int ac_get4(ac_hdl hdl, int idx, int *val);
```
キャッシュの指定 index の値の取得(ac_get4: 4-byte 要素用)

```
int ac_set(ac_hdl hdl, int idx, void *val);
int ac_set4(ac_hdl hdl, int idx, int val);
```
キャッシュ中の指定 index の値の変更(ac_set4: 4-byte 要素用)

図 2 最適化ライブラリ API

提案する最適化ライブラリでは，管理用 API の ac_open を用いて対象配列を指定し，ac_accesshint を用いてキャッシュ対象とする index を指定した後，as_cachestart を用いてキャッシュを取得する．取得したキャッシュにアクセスして必要な処理を行った後，必要に応じて ac_cachesync を用いてキャッシュへの書換えを元の配列に反映させる．

キャッシュの管理方式には Write back 方式を主に使用するが，実行時に Write Through 方式等他の方式も選択可能とする．プロトタイプの最適化ライブラリは UPC 言語で記述され 1010 行であった．

5.2 Toolkit との統合

High Productivity Computing Systems Toolkit[7]は，あらかじめ用意されている定式化された専門家の最適化に関する知識に従い，対象プログラムのボトルネックを発見し，最適化の手順を提案し，ソースコードの自動変換を行うツールである．この Toolkit は，ボトルネックを発見する Bottleneck Discovery Engine，最適化の解決策を提案する Solution Determination Engine，最適化を実際に適用する Solution Implementation Engine から構成される（図 3 参照）．

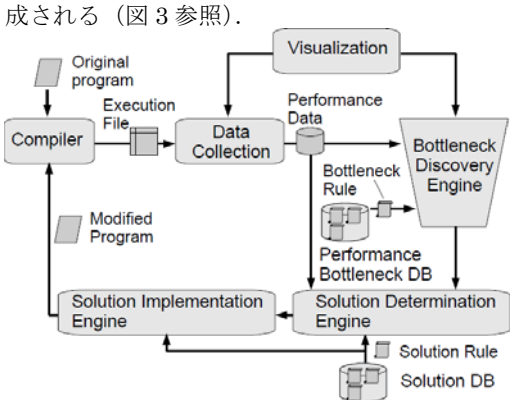

図 3 High Productivity Computing Systems Toolkit

この Toolkit では専門家の知識を定式化したものをルールと呼ぶ．提案する最適化手法は，適用対象ボトルネックを発見するボトルネックルール及び最適化手法を提案・適用するソリューションルールとして実装されることになる．

このうち，ボトルネックルールについては現在検討中であり，現在の実装ではホットスポット全てを今回の想定したボトルネックとして検出するボトルネックルールを使用した．

提案手法の4.5節で述べた正当性確認のロジックや4.6節で述べた自動ソースコード変換のロジックは，

Solution DB 中のソリューションルールとして実現される. 利用者が適用不可能なホットスポットを指定した場合, 正当性確認のロジックが適用可能性をチェックし, 適用が拒絶される. プロトタイプでは, この Toolkit が提供するライブラリ（解析木アクセス, ループ依存性解析, ソースコード変換）を使用した. このロジックは C 言語で記述され 989 行であった.

6. Connected Component アルゴリズムへの適用事例の基づく考察

本論文では UPC 言語の共有配列への不規則アクセスの最適化手法を提案する. 共有配列への不規則アクセスの代表として, twitter や BLOG のアクセス LOG 解析等今後の計算需要の拡大が予測されるグラフ理論から CC アルゴリズムプログラムを例として, 提案する最適化手法を適用する事例について説明する.

6.1 Connected Components (CC)アルゴリズム

CC アルゴリズムは, 辺を両端の頂点の識別子の組として, グラフを構成する辺のリストとして与え, 与えられたグラフがいくつの断片で構成されるかを計算するアルゴリズムである. 以下, このアルゴリズムのコア部分のソースコードを示す.

```
int connected_components(shared E * El,
    int n, int m, shared int **pD,
    int *pncomps)
{
  shared int *D;
  …
  upc_barrier;
  while (1) {
    …
    upc_forall(i = 0; i < m; i++; &El[i]){
      u = El[i].u;
      v = El[i].v;
      if (D[u] < D[v]) {
        D[D[v]] = D[u];
        grafted = 1;
      } else if (D[v] < D[u]) {
        D[D[u]] = D[v];
        grafted = 1;
      }
    }
    upc_barrier;
    grafted = all_reduce_i(grafted,
                           UPC_ADD);
    if (grafted == 0)
      break;
    upc_forall(i = 0; i < n; i++; &D[i]){
      while (D[i] != D[D[i]])
        D[i] = D[D[i]];
    }
    upc_barrier;
  }
  …
}
```

配列 D の各要素 D[i]は, 頂点 i が所属する断片の識別子として, その頂点から到達可能な最小の頂点を示す. プログラムの最初のループ(while (1))がこのプログラムのメインループであり, 実行時間のほぼ全てがこのループ中で費やされる. このメインループは, 以下の 2 つのループを含む.

1. grafting loop
   全ての辺を回るループ(upc_forall (i = 0; i < m; i++; &El[i])). 各辺の両端の頂点で D の値が異なる場合に小さい方にあわせる. このループでは, 各 iteration 中で辺の情報を得るために配列 El に, またその辺の両端の頂点の情報を得るために共有配列 D にアクセスする. ループの affinity は辺に合わせる必要があるので, 両端の頂点の情報を持つ D へのアクセスはローカルアクセスが保証されない不規則なアクセスになる.
2. short-cutting loop
   全ての頂点を回るループ(upc_forall (i = 0; i < n; i++; &D[i])). 各頂点の D の値(到達可能な最小頂点)と D[D[i]]の値(最小頂点から到達可能な最長頂点)が異なる場合は更新する. このループの affinity は&D[i]となっているので, D[i]へのアクセスに関してはローカルアクセスが保証される一方, D[D[i]]へのアクセスについては一般にローカルアクセスは保証されない.

本論文では, この 2 つのループのうち全てのアクセスのローカルアクセスが保証されない grafting loop における共有配列 D へのアクセスに着目して最適化を行う.

6.2 最適化のための変換後のソースコード

以下, grafting loop の最適化のために変換されたソースコードを示す.

```
int connected_components(shared E * El,
   int n, int m, shared int **pD,
   int *pncomps)
{
  shared int *D;
  …
  upc_barrier;
  while (1) {
    …
    ac_cachesync(hdl, 0);
    upc_forall(i = 0; i < m; i++; &El[i]){
      …
      u = El[i].u;
      v = El[i].v;
      ac_get4(hdl, u, &du); /* du=D[u] */
      ac_get4(hdl, v, &dv); /* dv=D[v] */
      if (du < dv) {
        ac_set4(hdl, dv, du);
          /* D[D[v]]=D[u]; */
        grafted = 1;
      } else if (dv < du) {
        ac_set4(hdl, du, dv);
          /* D[D[u]]=D[v]; */
        grafted = 1;
      }
    }
    ac_cachesync(hdl, 0);
    upc_barrier;
    grafted =
      all_reduce_i(grafted, UPC_ADD);
    if (grafted == 0)
      break;
    upc_forall(i = 0; i < n; i++; &D[i]){
      while (D[i] != D[D[i]])
        D[i] = D[D[i]];
    }
    upc_barrier
    ac_cachesync(hdl, 0);
  }
  …
}
```

[13]では最適化対象関数コード中に配列の index の変換等様々なコード生成する必要があり，書換えが複雑であったが，本手法によれば，ソースコード変換は比較的単純な文の挿入及び変更で構成され，元々のプログラム構造も保存されている．

6.3 性能測定

CC プログラムを[13]の手動による最適化手法及び提案手法に基づいた自動最適化を行い，性能を測定した．ただし，プログラマの手動による最適化ではアルゴリズム自身の変更を含まないものとした．性能測定には，AIX®オペレーティングシステムが動作する Power5+™プロセッサのクラスタを，コンパイラには論文執筆時点で最新の Berkeley UPC コンパイラの version 2.11.4 を用いた．性能測定環境は以下の通り．

ハードウェア
プロセッサ: 1.9 GHz, 16 proc. (IBM® P575+)
メモリ: 64GB, DDR2
ソフトウェア
OS: AIX 5300-09-03-0918
コンパイラ: upcc, v. 2.11.4 (STABLE)
最適化オプション: -O
ネットワーク: lapi
測定条件
スレッド/ノード: 16 (1 スレッド/プロセッサ)
入力データ: 頂点数 1 億 - 辺数 2 億

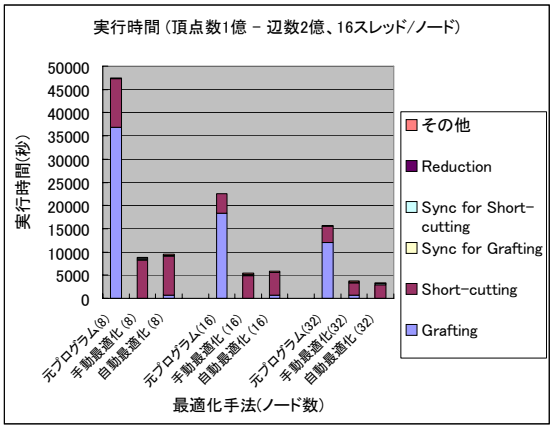

図 4: 実行時間の内訳

6.4 性能測定結果についての考察

図 4 は，頂点数 1 億及び辺数 2 億を入力とし，ノード数を 8,16,32 に変化させた場合の実行時間を示す．プログラマの手動最適化プログラム及び提案手法による自動最適化プログラムの性能はほぼ同一で元プログラムの約 5 倍である．手動最適化プログラムと提案手法による自動最適化プログラムの性能の違いは 1-8%で，違いは提案手法の自動最適化の際に導入したハッシュ表の管理オーバヘッドに

よるものと考えられる．この差は HPC プログラムでは無視し得ないものではあるが，アプリケーションの利用者のプログラム最適化のスキル不十分，もしくは，作業時間不足等により，プログラム最適化を全く行わない場合も多くみられる現状では，提案手法による自動最適化は有用であるといえる．

図 4 の実行時間の内訳を見ると，元プログラムではほとんどのオーバヘッドは grafting loop から生じている．一方，2 つの最適化プログラムでは，grafting loop の実行時間はほぼ全て削減され，ほとんどの時間が short cutting loop で費やされている．

6.5 使用メモリ量についての考察

キャッシュ管理のために，手動最適化プログラムでは通常の配列，提案手法による自動最適化プログラムではオープンアドレスハッシュ表(open addressed hash table)を使用する．これによるメモリ使用量の違いについて考察する．

手動最適化プログラムでは，<自スレッドがアクセスする辺数>*2 個，1 個当り 4 バイト(配列要素のデータ長が 4 バイトの場合)のメモリを使用する．一方提案手法では，<自スレッドがアクセスする頂点数>個，1 個当り 8 バイト(データ 4 バイト:データ + index 及び status 4 バイト)のメモリを使用する．

提案方式では複数の辺から共有される頂点は 1 つに縮合されるので要素数に関しては提案手法が有利となる一方，要素当りのメモリ量については手動による最適化の方式が有利となる．

7. 結論及び将来の課題

本論文では，PGAS プログラムの通信ボトルネックを自動ソースコード変換に基づくキャッシュメカニズムの導入により解消する手法を提案し，最適化フレームワーク High Productivity Computing Systems Toolkit 上に実装したプロトタイプ及び性能測定結果について説明した．CC プログラムの例では，提案手法に基づく自動最適化により元プログラムの約 5 倍の性能改善が得られた．この性能は既存の最善の手動最適化プログラムの性能に匹敵(1 億頂点- 2 億辺の場 1-8%の相違)するものであった．

今後は，提案手法に基づいてより多くのプログラムの最適化を行いより定量的に本手法の効果について評価したい．

謝辞 本研究は，米国国防高等研究計画局(US DARPA) 契約番号 HR0011-07-9-0002 による．

参 考 文 献

[1] High Productivity Computer Systems Project : http://www.highproductivity.org/

[2] The UPC Consortium. “UPC Language Specification (V1.2), ” http://upc.gwu.edu/docs/upc_specs_1.2.pdf (2005).

[3] Primary webpage of the Unified Parallel C: http://upc.gwu.edu/

[4] Co-Array Fortran homepage: http://www.co-array.org/

[5] The Chapel Project homepage: http://chapel.cray.com/

[6] The X10 Programming Language homepage: http://x10-lang.org/

[7] Guojing Cong, I-Hsin Chung, Hui-Fang Wen, David J. Klepacki, Hiroki Murata, Yasushi Negishi, Takao Moriyama, “A Holistic Approach towards Automated Performance Analysis and Tuning,” Proceedings or Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Aug. 25-28, 2009.

[8] Photran web site. http://www.eclipse.org/photran/

[9] D. Roberts, J. Brant, and R. Johnson. “A refactoring tool for Smalltalk.” Theory and Practice of Object Systems, 3(4):253–263, 1997.

[10] ROSE project web site. http://rosecompiler.org/

[11] Yasushi Negishi, Hiroki Murata, Takao Moriyama, “A Proposal of Operation History Management System for Source-to-Source Optimization of HPC Programs,” Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD2009), July 19-20, 2009.

[12] Jun Doi, Yasushi Negishi, “Overlapping Methods of All-to-All Communication and FFT Algorithms for Torus-Connected Massively Parallel Supercomputers,” Proceedings of Supercomputing (SC10), Nov. 13-19, 2010.

[13] Guojing Cong, Gheorghe Almasi and Vijay Saraswat, “Fast PGAS connected components algorithms,” Proceedings of the Third Conference on Partitioned Global Address Space Programming Models (PGAS 2009)

[14] R. Whaley and J. Dongarra. “Automatically tuned linear algebra software (ATLAS),” Supercomputing 98, Orlando, FL, November 1998. www.netlib.org/utk/people/JackDongarra/PAPERS/atlas-sc98.ps.

[15] R. Vuduc, J. Demmel, and K Yelick. “OSKI: A library of automatically tuned sparse matrix kernels,” SciDAC 2005, Journal of Physics: Conference Series (2005).

[16] W. Chen, R. Bringmann, S. Mahlke, and et al. “Using profile information and scheduling.” Lecture Notes in Computer Science, 757:31-48, January 1993.

[17] Tarek El-Ghazawi, Phillip Merkey, Steve Seidel. “High Performance Parallel Programming with Unified Parallel C (UPC),” SC05 Tutorial (2005).