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.

Cell/B.E. 向けフレームワーク CellSs の タスクの自動切り出しによる改良

今井満寿巳† 　津邑　公暁† 　松尾　啓志†

† 名古屋工業大学

あらまし　マルチコアプロセッサの一つである Cell/B.E. は，高い処理性能を持つが，それを引き出すためにはアーキテクチャ構成を意識したプログラミングが必要であり，プログラマの負担が大きい．その負担を軽減するため，Cell/B.E. 向けのプログラム開発環境が検討されている．Cell/B.E. 向けフレームワーク CellSs は逐次プログラムを複数の SPE で並列実行されるプログラムに変換する．しかし，SPE で実行されるタスクは pragma を用いてプログラマが指定する必要があり，タスクが並列実行されることを意識したプログラミングが必要である．そこで，逐次プログラムから並列性を持つ箇所を抽出し，自動的にタスクに変換する手法を提案する．また，リダクション演算を自動的に生成することで，より多くのプログラムを並列プログラムに変換可能にする手法を提案する．提案手法により，プログラマがタスクを指定すること無しに，逐次プログラムを並列プログラムに変換することを可能にした．また，リダクション演算を自動生成することで，プログラムを書き換えること無く，最大で 5.5 倍の速度向上を達成した．

キーワード　マルチコア，Cell/B.E.，自動並列化

An Aoutomatic Task-Specification Method for CellSs: a Programming Framework for Cell/B.E.

Masumi IMAI†, Tomoaki TSUMURA†, and Hiroshi MATSUO†

† Nagoya Institute of Technology

Abstract　A heterogeneous multicore processor Cell/B.E. can achieve a high performance, but requires programmers to be familiar with its architecture and behavior, and this must make the burden heavy for them. Hence, several development environments for Cell/B.E. have been studied. Cell Superscalar is one of the environments, and it translates sequential programs into parallel programs for Cell/B.E. However, programmers still must designate tasks which should be processed on SPE by using pragmas. To solve this problem, this paper proposes a method for translating sequential programs into parallel completely automatically. This paper also proposes a mathod for generating reduction code automatically. The experiment results with several programs show that the proposed methods improve the performance about 5.5-fold in maximum.

Key words　Multicore，Cell/B.E.，Auto parallelization

1. はじめに

近年，プロセッサの発熱量増加問題により，クロック周波数の向上が頭打ちになりつつあり，プロセッサのスカラ処理性能の向上が難しくなってきている．そのため，今日のプロセッサでは，マルチコア化を進め並列処理性能を向上させる事で，プロセッサ全体としての処理性能の向上を図っている．

Cell Broadband Engine [1](以降 Cell/B.E.) は，SONY，東芝，IBM の 3 社によって共同で開発された，ヘテロジニアスマルチコアプロセッサである．Cell/B.E. は 1 つの汎用コア PPE(PowerPC Processor Element) と，8 つの演算コア SPE(SynergisticProcessor Element) を 1 チップ上に集約している．Cell/B.E. は高い処理性能を持つが，それを引き出すためにはコアの種類に合わせた処理の割り当てや，メインメモリと各 SPE が持つローカルストア (以降 LS) と呼ばれる固有メモリ間での明示的なデータの転送といった，Cell/B.E. のアーキテクチャ構成を意識したプログラミングが必要である．そのため，プログラマの負担は大きく，その負担を軽減する Cell/B.E. 向けのプログラム開発環境が検討されている．

本稿では，Cell/B.E. 向けの開発環境の一つである Cell Superscalar [2](以降 CellSs) に注目する．CellSs は逐次プログラムを複数の SPE で並列実行されるプログラムへと変換する．しかし，そのためにはタスクと呼ばれる SPE で実行される処理単位を，pragma を用いてプログラマが指定する必要があり，

プログラムが並列に実行されることを意識したプログラミングが必要となる．そこで本稿では，逐次プログラムから並列性を持つ箇所を抽出し，その箇所をタスクとして自動的に切り出す手法を提案する．これにより，容易に Cell/B.E. の性能を引き出すことができる．

以降，2. 章では Cell/B.E. 向けの開発環境について述べる．3. 章では本稿で提案するタスクの自動切り出しについて述べ，4. 章でその実装について述べる．5. 章で本手法を評価し，6. 章で結論を述べる．

2. 研究背景

本章では既存の Cell/B.E. 向けの開発環境とその問題点について述べる．代表的な Cell/B.E. の開発環境に Cell ToolKit [3](以降 CTK) がある．CTK は，Cell/B.E. 上での C/C++言語によるプログラム開発を支援するライブラリである．CTK は SPE プログラムの実行制御や，PPE と複数の SPE を用いた並列処理のための基本的な API を提供する．この API を用いることで，データ転送や SPE プログラムの起動等の記述を簡略化することができる．しかし CTK の提供する機能は，あくまで記述を簡略化するためのものであり，CTK を利用する場合でも明示的なデータ転送や，コアに応じたプログラムの書き分けは依然として必要である．

この問題を解決する開発環境の一つに CVCell [4] がある．CVCell はコンピュータビジョン向けのライブラリ OpenCV を，Cell/B.E. 向けに最適化したものである．OpenCV は物体検出や色変換などの多数の API を提供しており，CVCell はそれらの主要な API を数倍から数十倍高速化している．プログラマは各 API を用いることで，SPE を使用して並列に処理をすることができるため，Cell/B.E. のアーキテクチャ構成を意識する必要が無い．しかし，CVCell ではライブラリによって提供されている処理しか行うことができず，プログラマが任意の並列処理を記述することはできない．

これに対し，任意の並列処理を記述できる開発環境として，CellSs がある．CellSs は逐次プログラムを複数の SPE で並列処理されるプログラムへと変換する．これにより，様々な処理を Cell/B.E. 向けに最適化することができる．CellSs において，SPE で実行される処理の単位はタスクと呼ばれる．タスクの実行に必要なデータの転送は CellSs が請け負うため，プログラマが明示的にデータの転送を記述する必要は無い．このように，CellSs を用いることでプログラマは Cell/B.E. のアーキテクチャ構成を意識すること無く，プログラマ自身が記述した処理を SPE を用いて並列に実行することが可能になる．しかし，CellSs にはプログラマが pragma を用いてタスクを指定する必要があるという問題がある．これは SPE を用いて並列に実行する箇所や実行単位を，プログラマが意識しなければならないということである．また，CellSs にはリダクション演算を自動的に生成する機能を持っていない．そのため，リダクション演算により競合を回避することで並列に実行できる処理であっても逐次実行されてしまい，Cell/B.E. の性能を十分に引き出せないことがある．そこで本稿では，これらの CellSs の問題点を解決するため，逐次プログラムから並列性を持つ箇所を抽出し，その箇所をタスクとして自動的に切り出す手法を提案する．

3. CellSs のタスク自動切り出し

本章では，本稿で提案するタスクの自動切り出し手法について述べる．CellSs の問題点を解決するため，逐次プログラムから並列性を持つ箇所を抽出し，その箇所を自動的にタスクとして切り出し，複数の SPE で並列実行されるプログラムに変換する．また，並列化にリダクション演算が必要であると判定した場合，リダクションコードを自動生成する．

3.1 並列化箇所の抽出

逐次プログラムからタスクを切り出すために，並列化する箇所を抽出する．一般的にループは並列性を持つため，逐次プログラム中のループを並列化の対象とする．

しかし，並列性を持つループでも，並列化すべきで無い場合が存在する．例えば，ループボディの計算量が非常に軽量である場合，並列化により削減される実行時間より，SPE プログラムの起動やデータ転送のオーバヘッドの方が上回ってしまうことがある．これを検出するためには，削減される実行時間とオーバヘッドを比較する必要があるが，実行時間を静的に測定することは難しい．よって，このようなループは，プログラマが pragma を用いてプリプロセッサに指示を出すことで，並列化対象から外すことができるようにする．

また，ループ中に I/O 処理を含む場合は並列化の対象から外す．I/O 処理を含む場合，並列実行するとその結果を一意に決定できなくなってしまうからである．よって，ループ内で printf 等の I/O 処理のある関数を検出した場合，そのループを並列化の対象外とする．

3.2 リダクション演算が必要な変数の検出

並列実行される複数の処理がある一つの変数に対してアクセスする場合，競合が起こる可能性がある．そのような場合，単純には並列化することができないため，リダクション演算によって競合を回避する必要がある．リダクション演算とは，並列処理において複数のタスクが 1 変数に対する読み出し及び書き込みをする際の競合を回避するための処理である．各タスクが 1 変数の代替として定義した一時変数に対して読み書きを行い，並列実行終了後に全ての一時変数を逐次的にリダクションする．提案手法ではプログラマに並列性を意識させないために，リダクション演算が必要な変数を自動的に検出する．

リダクション演算が必要か否かの判定には次の条件を用いる．

(1)　対象変数に読み出し及び書き込みを行っている

競合が発生する条件として，1 変数に対する読み書きが挙げられる．そのため，同じ変数に対して読み出し及び書き込みを行っているか否かを判定する．

(2)　処理の実行順序に依存が無い

ある処理の結果により，他の処理の結果が変わるような場合，リダクション演算を用いても競合を回避することはできない．そのため，処理の順序に依存がある (以降，順序依存がある) か否かを判定する．

各変数が上記の条件を満たしているか否かを判定し，満たしていた場合その変数はリダクション演算が必要であるとする．

読み出し及び書き込みを行っているか否かの判定方法を図 1 に示すコードを例に述べる．ここで図中の変数 `var` を判定の対象とする．例えば (左辺) = (右辺); といった代入を行う文において，図 1 中の (1) のように (左辺) が `var` であれば書き込みと判定し，(2) のように (右辺) にも `var` があれば読み出しと書き込みがあると判定する．なお (3) のように`+=`や`-=`といった複合代入演算子を用いた演算も，読み出しと書き込みがあると判定する．また，(4) のように if 文や for 文，while 文の条件文に `var` が使われている場合も読み出しを行っていると判定する．

順序依存の判定では，次の 4 つの条件を全て満たさない場合，順序依存が無いとする．

(A)　対象変数に対して加減算と乗除算を混在して使用している

(B)　if 文の条件文で使われている対象変数に対して，比較した値と異なる値が代入されている

```
if(var < num1){   //(4) 読み出しのみ
  var = num2;     //(1) 書き込みのみ
  var = var + 5;  //(2) 読み出し及び書き込み
  var += 10;      //(3) 読み出し及び書き込み
}
```

図 1　読み出し及び書き込みの判定

```
var = (var + num1) * 2;          //NG1
var = var / num2 - 2;            //NG2
var = var + 2;                   //OK1
var = var / num1 * (num2 + 10); //OK2
```

図 2　加減算と乗除算を混在して使用しているか否かの判定

```
if(var > num1) var = var + num1; //NG1
if(var > num1) var = num2;       //NG2
if(var > num1) var = num1;       //OK
```

図 3　if の条件文で比較した値と異なる値が代入されているか否かの判定

(C)　値が書き換えられた対象変数を他の変数に代入している
(D)　ライブラリレベルで定義されている関数の引数に対象変数を使用している

以降，それぞれの条件について具体的に述べる．説明では変数 var を対象変数とする．また，各処理は並列化対象となるループボディであり，各イタレーションにおいて変数 num1, num2 には異なる値が格納されていると仮定する．そして，各イタレーションが実行される順序を，以降実行順と呼ぶ．

まず図 2 を例に，条件 (A) について説明する．図 2 で NG1 と記述されている行は，var に対して num1 の値を足してから 2 をかけているが，この文を展開すると var に対して+と*の演算を適用している．同様に NG2 と記述されている行は，var に/と-の演算を適用している．このように加減算と乗除算を対象変数に適用する場合は，実行順を入れ替えると最終的な結果が変わってしまう．一方 OK1 と記述されている行は var に対して+のみ適用しており，OK2 と記述されている行は var に対して/と*のみを適用している．このような文は対象変数に対して実行順を入れ替えても，最終的な結果は変わらない．

次に図 3 を例に，条件 (B) について説明する．図 3 で NG1 と記述されている行は，条件文に使われている var に対して加算をしている．また NG2 の行は，条件文で比較した num1 とは異なる値を var に代入している．これらの行は実行順によって条件文の判定が変わるため，最終的な var の値が異なる．一方 OK と記述されている行は，いわゆる num1 の最小値を求めるものであり，num1 がどんな値であっても最終的に var に格納される値は同じである．

次に図 4 を例に，条件 (C) について説明する．var に対して加算が行われており，その結果を num1 に代入している．このように，値が書き換えられた var を num1 に代入すると，実行順によって最終的な num1 の値が変わるため順序依存がある．

最後に，条件 (D) について説明する．ライブラリレベルで定義された関数は，関数内でどんな処理がされているか分からないため，ライブラリ関数の引数に対象変数を指定した場合は解析不可能であるとする．ただし，CellSs で定義されている関数については，各関数の順序依存の有無に関する情報を持っているため，その情報を用いて判定する．

```
var += 10;
num1 = var; //NG
```

図 4　値が書き換えられた対象変数を他の変数に代入しているか否かの判定

3.3　並列プログラムへの変換

3.1 節で述べた手順で抽出したループを SPE で並列実行されるように変換する．逐次プログラムのループ中の処理を実行するタスクを生成し，元々ループ文があった箇所でそのタスクを呼び出すように変換する．

生成されるタスクは対象ループの複数イタレーションを実行する．これは SPE プログラムの起動コストを削減するためである．例えば，10 イタレーションの処理を，1 イタレーションずつ実行する場合，SPE プログラムを起動するコストは 10 回分かかってしまう．しかし，2 イタレーションずつ実行すれば，起動コストは 5 回分に削減できる．1 タスクで実行するイタレーション数は 1 イタレーションで扱うデータ量と LS の空き容量を考慮して決定する．これは一般的に実行するイタレーション数が増えると，扱うデータ量も増えるためである．ただし，配列以外の変数は実行するイタレーション数によって扱うデータ量が増えないため，配列のデータ量のみを考慮する．$Array$ を 1 イタレーションで扱う配列のデータ量，$Capacity$ を LS の空き容量，N を 1 タスクで実行するイタレーション数とすると，N は次の条件を満たす必要がある．

$$Array \times N \leqq Capacity \tag{1}$$

$Array$ の見積り方を図 5 のプログラムを例に説明する．この例では num は int 型であると仮定する．図 5 では 1 イタレーションでは num[i] の 1 要素しか扱っていないため，$Array$ は 4byte となる．

また，添字等を用いてランダムアクセスする場合，複数イタレーションを 1 つのタスクで実行すると，実行に必要な配列要素のアドレスが連続していない可能性がある．アドレスが連続していないと，タスクの実行に必要なデータを一度に転送することは難しい．そのため N は，タスクの実行に必要な配列要素のアドレスが連続していることを保証できる数である必要がある．N は以上の 2 つ条件を満たす最大の自然数に決定する．

条件文から N を決定した後，逐次プログラムを変換する．図 5 のプログラムを変換した例を，図 6 に示す．この変換例では N を 10 としている．まず，タスクを制御するため，新たに__taskid という変数の宣言を追加する (26 行目)．__taskid はタスク内のループの制御に用いる．__taskid はループの実行前に 0 で初期化され (29 行目)，1 度タスクを呼び出す毎にインクリメントされる (32 行目)．次に，ループの内部をタスクを呼び出す処理に変換し (31 行目)，ループの前後に CellSs の pragma を挿入する (28,34 行目)．さらに，各イタレーションの最後に行われる i のインクリメントを i に 10 加算する処理に変換し，タスクが呼ばれる回数を調節する (30 行目)．この例で len の値が 128 であると仮定したとき，タスクは 13 回呼ばれるが，タスク内のループは元々の条件で終了するため，最後のタスクでは 8 イタレーションしか実行されない．そのため，変換前と同じ 128 回のループが実行される．最後にループの内部の処理を切り出してタスクを生成する (1〜11 行目)．タスクの引数はループで使われる A, B, num, len の他に，__taskid を加えた 5 つになる (2, 3 行目)．タスク内では切り出し元のルー

```
1 int main(void){
2   for(i = 0; i < len; i++){
3     A += num[i];
4     if(B < num[i]){
5       B = num[i];
6     }
7   }
8 }
```

図 5　逐次プログラムの例

```
1  #pragma css task input(num,__taskid,len) inout(A,B)
2  void task(int *A, int *B, int num[10],
3            int __taskid, int len){
4    for(i = 10*__taskid;
5        i < 10*(__taskid+1) && i < len; i++){
6      *A += num[i];
7      if(*B < num[i]){
8        B = num[i];
9      }
10   }
11 }
12
13 void __aggregate_A(int *piece; int *reduction;
14                   int *original){
15   *reduction += *piece - *original;
16 }
17
18 void __aggregate_B(int *piece; int *reduction;
19                   int *original){
20   if(*reduction < *piece){
21     *reduction = *piece;
22   }
23 }
24
25 int main(void){
26   int __taskid;
27
28 #pragma css start
29   __taskid = 0;
30   for(i = 0; i < len; i+=10){
31     task(&A, &B, num[i], __taskid, len);
32     __taskid++;
33   }
34 #pragma css fihish
35 }
```

図 6　変換後のプログラムの例

プを 10 イタレーション実行する．各タスクが処理を分割して実行するために i の初期値を__taskid に 10 を掛けた値にし，ループの終了条件を変更する (4, 5 行目)．

タスクの生成が終了した後，リダクション演算が必要と判定された変数がある場合はリダクションコードを生成する．リダクションコードは__aggregate_変数名という名前の関数として変数毎に生成される (13〜23 行目)．関数は各タスクの結果が格納される piece，各タスクのリダクション結果が格納される reduction，タスク実行前の初期値が格納される original という 3 つの引数を持つ (13, 14, 18, 19 行目)．関数は実行されたタスクの数だけ呼び出され，各タスクの結果を piece として受け取り，リダクションする．なお，reduction の初期値は original と同じ値である．

A はタスク内で値を加算されている．よって，加算を用いてリダクションする (15 行目)．A のリダクションでは，piece から original を減算することで，元の値からの差分を求め，それを reduction に足し込んでいく．

```
1 int main(void){
2   for(i = 0; i < dim; i++){
3     for(j = 0; j < dim; j++){
4       value[i][j] = i + j;
5     }
6     n += value[i][0];
7   }
8 }
```

図 7　1 イタレーションで扱うデータ量を見積もることができない例

```
1  #pragma css task input(i,__taskid) output(value)
2  void task(int value[10], int i, int __taskid){
3    for(j = 10*__taskid;
4        j < 10*(__taskid+1) && j < dim; j++){
5      value[j] = i + j;
6    }
7  }
8
9  int main(void){
10   int __taskid;
11
12 #pragma css start
13   for(i = 0; i < dim; i++){
14     __taskid = 0;
15     for(j = 0; j < dim; j+=10){
16       task(&value[i][0], i, __taskid)
17       __taskid++;
18     }
19 #pragma css barrier
20     n += value[i][0];
21   }
22 #pragma css finish
23 }
```

図 8　内側のループを並列化対象として変換した例

B はタスク内で num の最大値を求めている．B のリダクションでは，piece と reduction を比較し，piece の方が大きければ redcution に piece を代入する (20〜22 行目)．これにより最終的に piece の最大値が B に格納される．

次に，1 イタレーションで扱うデータ量を見積もることができない場合を考える．図 7 に示す例ではループがネストしており，内側のループ回数によって外側のループの 1 イタレーションで扱うデータ量が変化するため，扱うデータ量を静的に見積もることができない．そこで，このような場合は最内ループを並列化の対象とする．図 7 のプログラムを変換した例を図 8 に示す．この例では N を 10 として変換している．最内ループを並列化対象とする場合，__taskid の初期化は，並列化対象のループの直前で行われる (14 行目)．そして，対象ループの直後に CellSs のプラグマを挿入する (19 行目)．CellSs では全てのタスクの終了後にその実行結果を書き戻す．そのため，20 行目のようにタスクの結果が格納される変数を参照する場合は，その直前でタスクの終了を待つようにプラグマを用いて指示する必要がある．

以上の手順を踏んで変換したプログラムを，既存の CellSs と同様に，タスクを SPE プログラムへ，タスクの呼び出しを SPE プログラムを起動する処理へと変換する．これにより，逐次プログラムを複数 SPE で並列実行されるプログラムへと変換できる．

4. リダクションコードの生成の実装

本章では，プログラマの記述無しにリダクションを実現する

```
1 #pragma css task input(a) inout(var1,var2)
2 void Task(int a, int *var1, int *var2){
3   var1 = var1 + a;
4   if(var2 > a * 2) var2 = a * 3;
5   var1 += ++var2;
6   var1 = a + var1 - (a * 10);
7   funcLib(var2, a);
8 }
```

図 9　プログラム例 (リダクション判定)

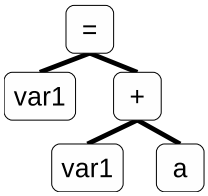

図 10　代　入　文

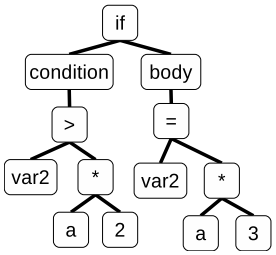

図 11　if　　文

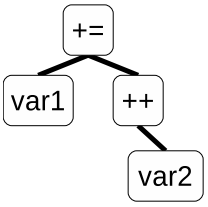

図 12　複合代入文

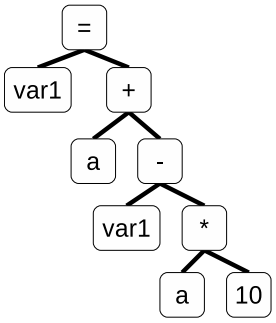

図 13　複数の演算が使用された文

ための具体的な実装方法について述べる．

4.1　リダクション演算が必要な変数の判定

3.2 節で述べたように，ループ内の変数に対して，リダクション演算が必要か否かを判定する必要がある．各条件はループ内を解析して生成した Abstract Syntax Tree (以降 AST) を探索して判定する．AST のノードは子を複数持つことがあり，以降の説明では左側から第 1 子, 第 2 子と呼ぶことにする．図 9 の例を用いて各条件の判定方法について説明する．なお，判定の対象変数は `var1`, `var2` とする．

条件 (1) 読み出し及び書き込みを行っているか

読み出し及び書き込みを行っているか否かの判定では次の条件ごとに判定する．

- 代入文の左辺が対象変数であれば書き込みを行っている
- 代入文の右辺，もしくは if 文や while 文の条件文に対象変数が使われていれば，読み出しを行っている
- 複合代入文の左辺が対象変数である，もしくは対象変数がインクリメントまたはデクリメントされていれば，読み出し及び書き込みを行っている

図 9 の 3 行目から生成される AST を図 10 に，4 行目から生成される AST を図 11 に，5 行目から生成される AST を図 12 に示す．代入文の左辺は図 10 のように，`=` ノードの第 1 子に保持される．そのため，`=` ノードの第 1 子を探索することで，書き込みを行っているか否か判定できる．また，図 11 から代入文の右辺は `=` ノードの第 2 子に，if 文の条件文は `condition` ノードの子に保持されていることが分かる．そのため，`=` ノードの第 2 子と `condition` ノードの子を探索することで読み出しを行っているか否か判定できる．そして，図 12 から左辺は `+=`ノードの第 1 子に，`++`演算子が適用される変数は`++`ノードの子に保持されていることが分かる．そのため，`+=`等の複合代入演算子のノードの第 1 子と`++`ノード，`--`ノードの子を探索することで読み出し及び書き込みを行っているか否か判定できる．

条件 (A) 加減算と乗除算を混在して使用しているか

図 9 の 6 行目から生成される AST を図 13 に示す．図 13 のように，ある変数に適用される演算はその変数のノードよりルートノードに近い位置にある．よって文の先頭ノードまで親ノードを遡っていくことで，その変数に適用される演算を全て探索することができ，対象変数に加減算と乗除算が両方とも適用されているか否か判定できる．この例では，`var1` を遡っていくと `-` と `+` しか存在しないので，加減算しか適用されていないことが分かる．

条件 (B) 比較した値と異なる値が代入されているか

if 文の条件文で使われている対象変数に対して，比較した値と異なる値が代入されているか判定する．この条件の判定は，条件文の中で対象変数が使われているかを判定し，使用されていた場合 if 文の中で対象変数に対して値が代入されているか否かを判定する．代入されていた場合はその値が条件文で比較した値と等しいかを判定する．

図 11 に示すように，if 文の条件文は `condition` の子に保持されているため，そこを探索することで条件文で対象変数が使用されているか判定できる．対象変数が使用されていた場合，`body` ノードの子を探索し，対象変数が `=` ノードの第 1 子にある場合，`=` ノードの第 2 子が条件文で比較した値と同じであるか否か判定する．この判定では対象変数以外の比較演算子ノードの子と，`=` ノードの第 2 子のそれぞれをルートとする部分木を比較する．部分木が完全に一致しなかった場合，比較した値と異なる値が代入されていると判定できる．図 11 では `var2` が `condition` の子に存在するので，`>` ノードの子である`*`ノードを保持しておく．そして，`body` ノードの子を探索すると `=` ノードの第 1 子に `var2` が存在することが分かる．ここで，保持しておいた `>` ノードの子の `*` ノードと，`=` ノードの第 2 子の `*` ノードをそれぞれルートとする部分木を比較する．すると，部分木が完全に一致しないため，条件文で比較した値と異なる値を代入していることが分かる．

条件 (C) 書き換えられた対象変数を他の変数に代入しているか

この条件では値が書き換えられたか否か判定し，書き換えられていたら他の変数に代入しているか否か判定する．対象変数の値が書き換えられる処理かは，先述した通り，`=` ノードの第 1 子や`++`ノードの子を探索することで判定できる．値が書き換えられたと判定されたら，フラグを立て，他の変数に代入しているかの判定に移る．他の変数に対象変数を代入する場合，代入文の左辺に他の変数が，右辺に対象変数が存在する．そのため，第 1 子が対象変数でない代入演算子ノードの第 2 子以降に対象変数が存在する場合，対象変数を他の変数に代入していると判定できる．フラグが立っている状態で，他の変数に代入していた場合，値が書き換えられた対象変数を他の変数に代入していると判定できる．CellSs で生成される AST は深さ優先で探索することで，各文が実行される順番で探索できるため，この判定では深さ優先探索を用いる．

図 11, 図 12 を例に判定の様子を説明する．探索は各文が実行される順番に行うため，図 11, 図 12 の順で探索する．図 11 の AST を探索すると `=` ノードの第 1 子に `var2` が存在するためフラグを立てる．ここで，この代入文は if 文の中にあり実行されるか分からないが，本手法では実行されるものとして判定

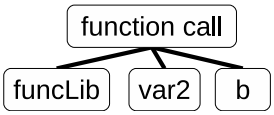

図 14　関数呼び出し

表 1　評 価 環 境

プラットフォーム	PLAYSTATION3
OS	Fedora 10
CPU	Cell/B.E.
動作周波数	3.2GHz
使用 SPE 数	6

する．そして図 12 の AST の検索に移る．ここで，`var1` を第 1 子に持つ`+=`ノードの第 2 子以降に `var2` が存在するため，値が書き換えられた `var2` を他の変数に代入していると判定できる．

条件 (D) ライブラリレベルで定義されている関数の引数に対象変数を使用しているか

図 9 の 7 行目から生成される AST を図 14 に示す．`function call` ノードの第 1 子に呼び出す関数名が保持されているので，それを基にプログラマが記述したコードの中に関数が定義されているか否かを探索する．関数の定義が見つからない場合，その関数はライブラリレベルで定義されていることが分かる．ライブラリ関数を呼び出していた場合，第 2 子以降を探索することで，ライブラリ関数の引数に対象変数を使用しているか判定できる．この例で，`funcLib` 関数がライブラリ関数であると仮定すると，`var2` をライブラリ関数の引数に使用していることが分かる．

4.2　リダクションコードの実行

3.3 節で述べたように，リダクション演算が必要であると判定された場合は，リダクションコードを生成する．そして，全てのタスク終了時に，このコードを用いて各タスクの結果をリダクションする．

リダクションの流れを順に説明する．まずタスクの呼び出し時に，CellSs に存在する，引数の情報を保持するための構造体を用いて，引数毎に対応するリダクションコードの関数ポインタを保持しておく．そして，各タスクの終了時その結果を直接引数のアドレスに書き込まず，タスク毎に用意した一時領域に書き込む．その後，全てのタスクが終了したら，保持しておいた関数ポインタを用いてリダクションコードを実行する．各タスクの結果を引数としてリダクションコード実行することで，各タスクの結果を一つの値にリダクションできる．全てのタスクの結果をリダクションし終えたら，その結果を引数へ書き込む．以上の処理を CellSs のランタイムライブラリで実行することで，プログラマがリダクションコードを実行する処理を記述することなく，リダクションを実現する．

5.　評価・考察

既存の CellSs と提案手法を実装した CellSs を比較し，評価を行った．評価環境を表 1 に示す．評価にはモンテカルロ法を用いて円周率を求めるプログラム (以降 montecarlo)，積分を行うプログラム (以降 integrarl) と，CellSs のサンプルプログラムである行列乗算を行うプログラム (以降 matmul)，疎行列 LU 分解を行うプログラム (以降 lu) を用いた．CellSs では並列化の効果を十分に得るためにはある程度の数のタスクが必要である．よって，montecarlo と integral で生成されるタスクの数は，SPE の数も考慮して 600 程度とした．その結果を図 15 に示す．縦軸は実行時間を表しており，既存の CellSs による実行

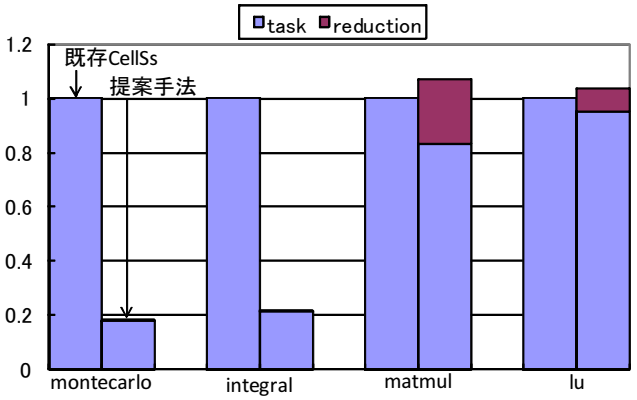

図 15　評 価 結 果

時間をを 1 として正規化した．グラフは左から既存の CellSs，提案手法を実装した CellSs となっている．また，図 15 中の凡例は実行時間の内訳を表しており，task はタスクの実行にかかった時間，reduction はリダクションにかかった時間である．

図 15 より，montecarlo, integral において，提案手法を用いることで速度向上できていることが分かる．これらは，既存の CellSs を使用した場合でもタスクは複数生成されるが，互いに依存関係があるため結果的に 1 つの SPE しか同時に動作しないようなプログラムである．これに対し提案手法を用いた場合，リダクション演算の追加により 6 つの SPE を用いて並列実行可能となったため，このような性能向上が得られた．

一方で，matmul, lu においては実行時間が増加してしまった．これらのプログラムには多くのタスク中一部のタスク間にのみ依存関係が存在しているため，既存の CellSs でも十分な並列性を使用できることと，多くのデータを扱うためにリダクションのオーバヘッドが大きくなったことから，このような結果となった．

6.　まとめと今後の課題

本稿では，Cell/B.E. 向けフレームワーク CellSs に対して，タスクの自動切り出しによる改良を施すことで，プログラマが並列に実行する箇所やその実行単位を意識することなく，複数 SPE で並列実行されるプログラムを作成することが可能な環境を提供することを提案した．

提案手法により，pragma を用いることなく，逐次プログラムを複数 SPE で並列実行されるプログラムに変換することを可能にした．また，リダクション演算を自動生成することで，プログラムを書き換えること無く，最大で約 5.5 倍の速度向上を実現した．

今後の課題として，より広い範囲のプログラムへの対応が挙げられる．今回の実装では，逐次プログラムのループのみを並列化の対象としているが，他にも並列性を持つ箇所は存在する．そういった箇所に対して並列化を行うことで，さらに Cell/B.E. の性能を引き出すことができると考える．

文　献

[1] Sony Computer Entertainment: “Cell Broadband Engine Architecture”, 1.01 edition (2006).
[2] P. Bellens, J. M.Perez, R. M.Badia and J. Labarta: “Cellss: a programing model for the cell be architecture” (2006).
[3] “Ctk: Cell toolkit library”, http://cell.fixstars.com/ctk/.
[4] “Opencv on the cell”, http://cell.fixstars.com/opencv/index.php/OpenCV_on_the_Cell.