修士論文

Cell/B.E. 向けフレームワーク CellSs の適用可能プログラム拡大による改良

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 22 年度入学　22413513 番

今井 満寿巳

平成 24 年 2 月 3 日

Cell/B.E. 向けフレームワーク CellSs の適用可能プログラム拡大による改良

今井 満寿巳

内容梗概

ゲート遅延に対する配線遅延の相対的な増大から，動作周波数の向上だけではプロセッサの性能向上を見込めなくなってきた．また，集積回路の微細化に伴う消費電力や発熱量の増大から，動作周波数自体の向上も困難になってきている．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，消費電力や発熱量の問題を解決しつつ，プロセッサ全体としての処理性能の向上を図っている．

マルチコアプロセッサの普及に伴い，様々な構造を持つプロセッサが開発されるようになった．その一例として Cell/B.E. が挙げられる．Cell/B.E. は汎用プロセッサと演算プロセッサを，1 チップ上に集積したヘテロジニアスマルチコアプロセッサである．また，Cell/B.E. の演算プロセッサはスクラッチパッドメモリを搭載している．

このような特殊な構造を持つアーキテクチャには，処理性能やエネルギー効率が高い反面，プログラマはアプリケーション開発の際，その特殊なアーキテクチャ構成等を意識してプログラミングしなければならないという問題点がある．そのため，プログラミングの繁雑さを少しでも軽減できるような開発環境が必要とされている．

そこで，本研究ではまず，Cell/B.E. 向けのアプリケーション開発を支援する開発環境の実現を目標とする．マルチコアプロセッサの中でも特に特徴的なアーキテクチャ構成を持つ，Cell/B.E. 向けの開発環境を実現することにより，さまざまな構成のマルチコアプロセッサ向け開発環境に対する知見を得ることを目的とする．Cell/B.E. 向けの既存の開発環境はいくつか存在するが，本研究ではその 1 つである CellSs に注目した．CellSs は，逐次プログラムを，複数の SPE を用いて並列実行されるプログラムへと変換するフレームワークである．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，様々な並列処理を記述することができる．

しかしながら，CellSs にはいくつかの問題点がある．CellSs を用いたプログラミングでは，プラグマを用いてタスクと呼ばれる実行単位を明示的に指定する必要があるため，並列に実行される箇所や単位をプログラマが意識する必要がある．また，プログラムを並列実行する際，競合を回避する手段としてリダクション演算が用いられることがあるが，CellSs にはリダクション演算を自動的に生成する機能が無いため，リダクション演算を用いることにより競合を回避できる処理も逐次実行されてしまう．さ

らに，タスク内で間接参照を用いてデータにアクセスするプログラムの動作が保証されておらず，そのようなプログラムは正しく動作しない可能性がある．そこで本論文では，これらの問題点を解決するため，CellSs の適用可能プログラムを拡大する手法を提案する．これにより，プログラマの負担を軽減しつつ，より多様なプログラムで Cell/B.E. の性能を引き出すことを実現する．

提案した CellSs の改良を実装した結果，並列に実行する箇所やその実行単位をプログラマに意識させることなく，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可能になったことを確認した．さらに，既存の CellSs では動作が保証されていないプログラムも，正しく動作することを確認した．また，既存の CellSs を用いて実行した場合と，改良した CellSs を用いて実行した場合の実行速度を比較した結果，プログラムを書き換えること無く，最大で約 4.1 倍の速度向上が得られることを確認した．

Cell/B.E. 向けフレームワーク CellSs の適用可能プログラム拡大による改良

目次

1 はじめに

ゲート遅延が支配的であった2000年代初頭までは，配線プロセスの微細化による高クロック化により，プロセッサの高速化を実現できた．しかし，数年前からは，ゲート遅延に対する配線遅延の相対的な増大や，集積回路の微細化に伴う消費電力及び発熱量の増大といった問題から，プロセッサの動作周波数の向上は困難になってきている．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，消費電力や発熱量の問題を解決しつつ，プロセッサ全体としての処理性能の向上を図っている．

マルチコアプロセッサの普及に伴い，さらなる高性能化やエネルギー効率の向上を目的として，様々な構造を持つプロセッサが開発されるようになった．その一例として **Cell/B.E. (Cell Broadband Engine)**[1] が挙げられる．Cell/B.E. は，1基の汎用プロセッサ **PPE (PowerPC Processor Element)** と8基の演算プロセッサ **SPE (Synergistic Processor Element)** を，1チップ上に集約したヘテロジニアスマルチコアプロセッサである．また，SPE はメインメモリに直接アクセスすることができない代わりに，小容量だが高速にアクセスできるスクラッチパッドメモリを搭載している．

また，汎用プロセッサのための計算用デバイスとして，描画処理専用プロセッサ GPU (Graphics Processing Unit) を汎用演算に利用する GPGPU (General Purpose computing on GPU) も注目されている．GPU は演算性能が高いだけでなく，電力あたりの性能も高く，既存のアクセラレータと比べて安価なことから，特に HPC (High Performance Computing) 分野で採用が進んでいる．

このような特殊な構造を持つヘテロジニアス環境は，ピーク処理性能やエネルギー効率が高い反面，プログラマはアプリケーション開発の際，その特殊なアーキテクチャ構成等を意識してプログラミングしなければならないという問題点がある．例えば，Cell/B.E. の場合は，2種類のコアに合わせた処理の割り当てや，メインメモリと LS 間で明示的なデータ転送が必要である．このように，特殊なアーキテクチャ構成を持つマルチコアプロセッサ向けのアプリケーション開発は，従来のプロセッサ向けのアプリケーション開発に比べ困難である．そのため，プログラミングの繁雑さを少しでも軽減できるような開発環境が必要とされている．

そこで，本研究ではまず，Cell/B.E. 向けのアプリケーション開発を支援する開発環境の実現を目標とする．マルチコアプロセッサの中でも特に特徴的なアーキテクチャ

構成を持つ，Cell/B.E. 向けの開発環境を実現することにより，さまざまな構成のマルチコアプロセッサ向け開発環境に対する知見を得ることを目的とする．Cell/B.E. 向けの既存の開発環境はいくつか存在するが，本研究ではその 1 つである **CellSs (Cell Superscalar)**[2] に注目した．CellSs は，逐次プログラムを，複数の SPE を用いて並列実行されるプログラムへと変換するフレームワークである．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，様々な並列処理を記述することができる．

しかしながら，CellSs にはいくつかの問題点がある．CellSs を用いたプログラミングでは，プラグマを用いてタスクと呼ばれる実行単位を明示的に指定する必要があるため，並列に実行される箇所や単位をプログラマが意識しなければならない．また，プログラムを並列実行する際，競合を回避する手段としてリダクション演算が必要になることがあるが，CellSs にはリダクション演算を自動的に生成する機能が無いため，リダクション演算を用いることにより並列実行可能な処理であっても逐次実行されてしまう．さらに，タスク内で間接参照を用いてデータにアクセスするプログラムの動作が保証されておらず，そのようなプログラムは CellSs では正しく動作しない可能性がある．そこで本論文では，これらの問題点を解決するため，CellSs の適用可能プログラムを拡大することを提案する．これにより，プログラマの負担を軽減しつつ，より多様なプログラムで Cell/B.E. の性能を引き出すことを実現する．

以降，2 章では Cell/B.E. のアーキテクチャ構成，及び Cell/B.E. 向けの既存の開発環境について述べる．3 章では，CellSs の問題点を解決するために，CellSs の適用可能プログラムの拡大について述べ，4 章では解析部の，5 章ではコード生成部の実装内容について述べる．次に，6 章で提案手法の評価を行い，7 章で関連研究について述べ，最後に 8 章で本論文全体をまとめる．

2　研究背景

本章では，本研究で取り扱う Cell/B.E. のアーキテクチャ構成，及び既存の Cell/B.E. 向けの開発環境について述べる．

2.1　Cell/B.E.

Cell/B.E. は，高い処理性能を目指し，SONY，東芝，IBM の 3 社により共同開発されたプロセッサである．Cell/B.E. は，1 基の汎用プロセッサ PPE と 8 基の演算プロセッサ SPE を 1 チップ上に集約したヘテロジニアスマルチコアプロセッサである．Cell/B.E.

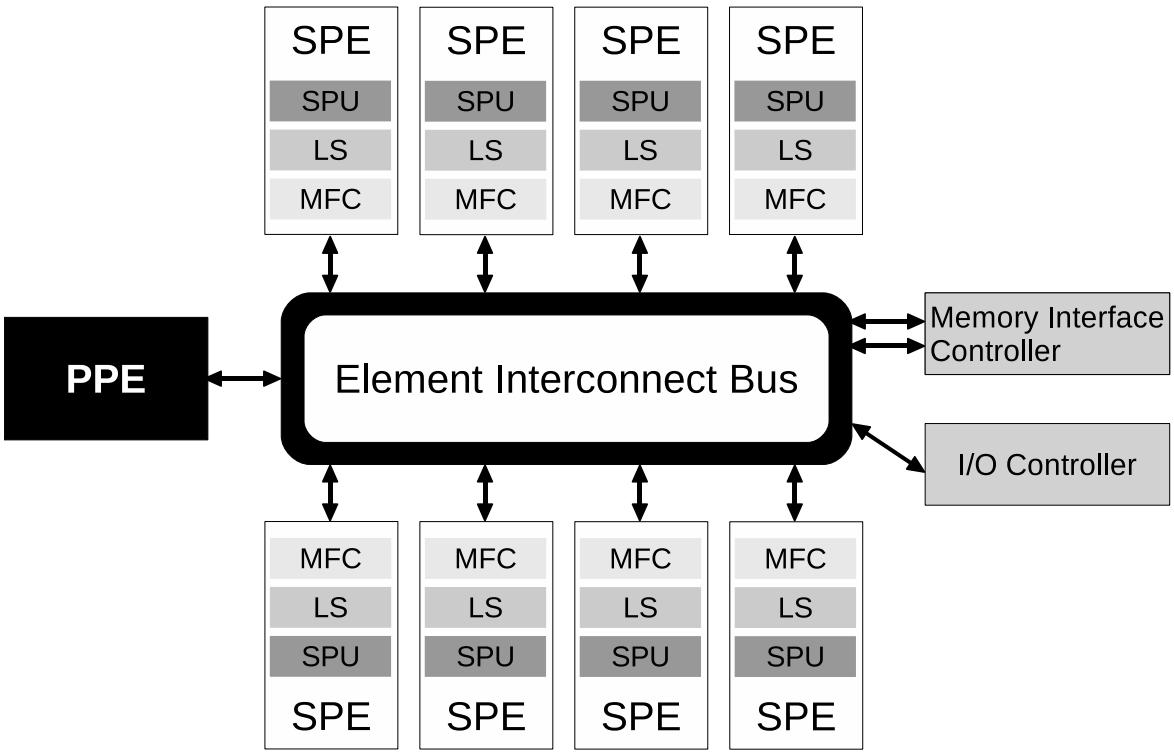

図 1: Cell/B.E. アーキテクチャ

はシングルスレッド時よりもむしろ，マルチスレッド時に高い性能を発揮するプロセッサであり，9 基のコアをあわせた浮動小数点演算能力はピーク時で 200GFLOPS を超える．Cell/B.E. アーキテクチャの概要を**図 1** に示す．

各プロセッサコアは，EIB (Element Interconnect Bus) と呼ばれる高速なバスで接続されており，EIB の転送速度は 204.8GB/秒である．また，EIB はメインメモリや外部入出力デバイスとも接続されており，それらと各コアを接続する役目も果たしている．

PPE は汎用プロセッサであり，そのアーキテクチャ構造は従来のプロセッサと大きな差異は無い．PPE は SPE に比べ，スレッドの切り替えが高速であることから OS の駆動などの処理を受け持ち，通常は SPE へのリソースマネージメントを行う．ただし，PPE のみを用いてプログラムを実行することも可能である．

一方，SPE は演算用のプロセッサであり，高い計算性能を持つ．そのため，Cell/B.E. では SPE を用いて処理を並列に実行することで，その性能を発揮することができる．SPE はそれぞれ 256KB の **LS (Local Store)** と呼ばれるスクラッチパッドメモリと，SPU (Synergistic Processor Unit)，MFC (Memory Flow Controller) と呼ばれるユニットを持つ．SPU とは，SPE の演算処理を行う核となるユニットであり，各 SPU は直接

メインメモリや他のSPE上のLSにアクセスすることはできない．そのため，処理に必要なデータはメインメモリからLSへ転送する必要がある．このデータ転送は，MFCによって制御される．このLSとメインメモリ間でのデータ転送に要する時間は非常に長いため，メモリレイテンシを隠蔽する方法として，ダブルバッファリングという手法がよく使用される．これは，LS上にバッファを2つ用意しておき，片方のバッファに対してデータを転送している裏で，もう片方のバッファのデータに対して計算を行うという方法である．またSPEは，128bitのSIMD演算命令用のパイプラインを持ち，LSへのアクセス命令用のパイプラインと合わせて，2-Wayのスーパースカラパイプライン構造となっている．計算のレイテンシは大きいが，128本の豊富なレジスタを利用して，複数のデータに対して同時に処理を適用することで，計算のレイテンシを隠蔽することが可能である．

このように，Cell/B.E.は特殊なアーキテクチャ構成を持つ．それ故に，Cell/B.E.向けのアプリケーションを開発する際に注意を払う必要がある．注意点は以下の3点である．

(1) Cell/B.E.の特徴を活かした，PPEとSPEを共に利用するアプリケーションを開発するためには，性質の異なる2種類のコアで動作するプログラムを，それぞれに対して用意する必要がある．また，それらのプログラムは，協調して動作するように設計する必要があるため，プログラマには並列分散プログラミングの技術が要求される．

(2) PPEと複数のSPEを協調動作させるようなプログラムを記述する場合，Cell/B.E.で用いられている，DMA (Direct Memory Access) 転送と呼ばれるデータ転送方式や，プロセッサ間で同期をとる場合のメモリシステムの機構など，アーキテクチャの詳細を理解する必要がある．また，Cell/B.E.の種類によって搭載されているSPEの数が異なることや，Cell/B.E.のハードウェア提供ベンダやCell/B.E.上に載せるOSの違い等により，SPEを制御する方法が異なることがある．そのため，開発されるアプリケーションが，アーキテクチャや下位システムに依存した移植性の低いものになりやすい．

(3) SPEは，SPU SIMD命令を用いることで，高速な演算を実現している．しかし，コンパイラや開発ツールを用いて自動的に最適化を行い，プログラムの高速化に繋がる箇所を抽出することは未だ困難である．そのため，SPEの性能を引き出すためには，プログラマがSPEを効率的に使用する技術を学習する必要があり，Cell/B.E.を用いたプログラミングの技術障壁は高い．

```
1  int main(void){
2    spe_program_handle_t *prog;
3    spe_context_ptr_t spe;
4    unsigned int entry;
5    spe_stop_info_t stop_info;
6
7     // SPE プログラムイメージのオープン
8    prog = spe_image_open("spe_program.elf");
9     // SPE コンテキストの生成
10   spe = spe_context_create(0, NULL);
11    // SPE プログラムのロード
12   spe_program_load(spe, prog);
13
14   entry = SPE_DEFAULT_ENTRY;
15    // SPE プログラムの実行
16   spe_context_run(spe, &entry, 0, NULL, NULL, &stop_info);
17
18    // SPE コンテキストの破棄
19   spe_context_destroy(spe);
20    // SPE プログラムイメージのクローズ
21   spe_image_close(prog);
22 }
```

図 2: 一般的な PPE プログラムにおける SPE プログラムの実行

これらを意識することはプログラマにとって負担となるため，その負担を軽減する開発環境が必要とされている．

2.2 Cell/B.E. 向けの既存の開発環境

Cell/B.E. 向けのアプリケーション開発の際，プログラマは Cell/B.E. のアーキテクチャ構成を意識してプログラミングしなければならない．一般的な PPE プログラムにおける，SPE プログラムを起動してから終了するまでの流れの例を**図 2** に示す．この例で

は，spe_program.elf という名前の SPE プログラムを実行する．まず，spe_image_open 関数を用いて，実行ファイルに格納された SPE プログラムのイメージをメインメモリ上にオープンする (図 2, 8 行目)．そして，spe_context_create 関数を用いて SPE コンテキストを生成し (図 2, 10 行目)，spe_program_load 関数を用いてオープンされた SPE プログラムを LS へロードする (図 2, 12 行目)．ロードされた SPE プログラムは spe_context_run 関数により実行される (図 2, 16 行目)．その後，SPE プログラムが終了し，不要になった SPE コンテキストは spe_context_destroy 関数を用いて破棄される (図 2, 19 行目)．最後に，メインメモリにオープンされた SPE プログラムのイメージを，spe_image_open 関数を用いてクローズする (図 2, 21 行目)．以上の処理により SPE プログラムを実行することができるが，これらの処理は全てプログラマが記述しなければならない．また，SPE を使用する場合，メインメモリと LS 間でのデータ転送や，PPE・SPE 間での同期処理が必要になることが多く，これらの処理もプログラマが記述する必要がある．このように，Cell/B.E. 向けのアプリケーション開発の際，プログラマは従来のプロセッサ向けのアプリケーション開発には必要無かった，Cell/B.E. 特有の処理を記述しなければならず．これはプログラマにとって負担となる．そのため，その負担を軽減する Cell/B.E. 向けの開発環境が必要とされている．

代表的な Cell/B.E. 向けの開発環境 1 つに CTK (Cell ToolKit)[3] 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 API を提供する．この API を用いることで，SPE プログラムの起動から終了までの一連の処理や，データ転送，同期処理を簡単に記述することができる．さらに，CTK は C++向けの API として，いくつかのテンプレートを定義している．このテンプレートを用いると，SPE プログラムを関数オブジェクト化し，SPE プログラムの実行を関数呼び出しのように記述することや，PPE プログラムで定義されたメソッドをコールバックとして定義し，SPE プログラムから呼び出すことが可能になる．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る．そのため，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 1 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[5, 6] を，Cell/B.E. 向けに最適化したものである．OpenCV は画像処理，構造解析，モーション解析，物体追跡，パターン認識など，多数のコンピュータビジョン向け処理の API を提供している．CVCell はそれらの API の内，

主要なものを Cell/B.E. 向けに最適化している．CVCell で行われている Cell/B.E. 向けの最適化は主に 3 種類に分けられる．1 つ目は SPE を用いた並列処理である．OpenCV で実装されている処理は多くの場合，並列性を有している．そのため，複数の SPE を用いて並列に処理することで高速化を図っている．2 つ目は，SIMD 演算を用いて処理することである．Cell/B.E. は SIMD 演算を用いる事で高い性能を発揮できるプロセッサである．そして，OpenCV で実装されている処理は高いデータ並列性を持つため SIMD 演算との相性が良い．そのため，処理に SIMD 演算を用いることにより高速化が期待できる．3 つ目はダブルバッファリングを用いた転送時間の隠蔽である．Cell/B.E. では，SPE で処理をするために処理に必要なデータを LS に転送する必要があるが，その転送に要する時間は長い．CVCell ではその転送オーバヘッドをダブルバッファリングにより隠蔽することで高速化を図っている．以上の最適化の結果，OpenCV の API を用いた場合に比べ，CVCell の API を用いた場合は，数倍から数十倍の高速化に成功しているプログラマは CVCell が提供する API を利用することで，SPE を利用するための処理を記述すること無く，複数の SPE を用いて並列に実行するプログラムを記述することができる．しかし，CVCell を用いたアプリケーション開発では，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．さらに，OpenCV で提供されている API の全てが CVCell で提供されているわけではない．そのため，CVCell を用いて開発可能なアプリケーションは限られている．

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 を使用して並列実行されるプログラムに変換する．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．また，プログラマは逐次プログラムを記述するだけで良いため，PPE プログラムと SPE プログラムへ処理を書き分ける必要は無い．CellSs が行う変換のフロー図を**図 3** に示す．CellSs はプリプロセッサによって，逐次プログラムを PPE プログラムと SPE プログラムへと変換した後，それをコンパイルし CellSs のランタイムライブラリをリンクさせることで，Cell/B.E. 向けの実行ファイルを生成する．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．タスクは関数として記述され，プログラマがプラグマを用いて明示的に指定する．CellSs はタスク間のデータの依存を動的に分析し，プログラムの結果が一意に決定するように，タスクをスケジューリングする．一方，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.

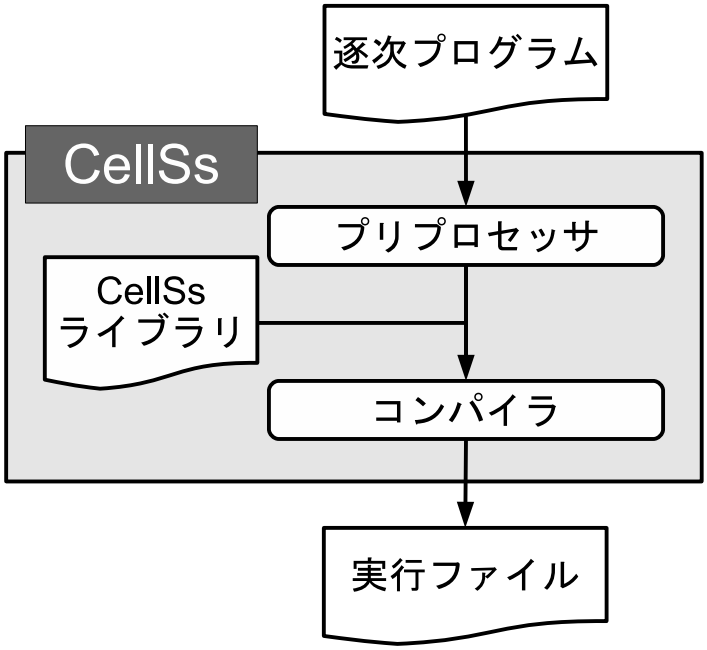

図 3: CellSs の変換フロー

```
1  typedef struct {
2    int id;
3    int *nums;
4  } Param_t;
5
6  #pragma css task input(array, param) inout(sum)
7  void SumArrayParam(int *sum, int array[64], Param_t param){
8    for(i = 0; i < 64; i++){
9      *sum += array[i] + param->nums[i];
10   }
11 }
```

図 4: タスクの記述例

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 3 つの問題点がある．その問題点を**図 4** に示す例を用いて説明する．1 つ目の問題点は，プログラマがプログラムの並列性を意識しなければならない

ことである．CellSs を用いたアプリケーション開発では，図 4 中の 6 行目のようにプラグマを用いてタスクを指定する．このとき，プログラマはタスクとして指定する箇所，この例では関数 SumArrayParam が，並列性を持つか否かを判断しなければならない．また，この例では SumArrayParam 1 回の呼び出しで，配列の 64 要素を処理するように記述されている．SPE にはタスク単位で処理が割り当てられるため，この場合，1 SPE は 1 度に 64 要素を処理することになる．このように CellSs を用いたアプリケーション開発では，タスクの粒度はプログラマによる記述によって決定するため，プログラマは LS の容量や利用する SPE の数等を考慮して，効率的に処理できる粒度を考える必要がある．以上のように，CellSs を用いてプログラムを記述する場合，PPE プログラムと SPE プログラムの書き分けや，明示的なデータ転送は必要は無いが，SPE を用いて並列に実行する箇所や実行単位を意識しなければならず，未だプログラマの負担は大きい．

2 つ目の問題点は，リダクション演算を用いて競合を自動的に回避する機能が無いことである．図 4 の例の場合，複数のタスクを並列に実行すると，変数 sum へのアクセスで競合が発生してしまう．しかし，CellSs 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ため，競合を回避することができない．そのため，この例では，タスクを並列に実行することができず，全てのタスクが逐次的に実行され，結果として 1 SPE しか使用されない．Cell/B.E. の高い処理性能は複数の SPE を用いて処理を並列に実行することで発揮できるため，このような場合，CellSs では Cell/B.E. の性能を十分に引き出せない．

3 つ目の問題点は，タスク内で間接参照を用いてデータにアクセスするプログラムの動作が保証されないことである．図 4 の例では 9 行目で構造体 param のメンバ nums を間接参照しているため，CellSs ではこのプログラムの動作は保証されていない．このように，CellSs では間接参照の使用に制限があり，プログラマはそれを考慮してプログラムを記述しなければならない．

3 CellSs の適用可能プログラムの拡大

前章で述べたように，CellSs には 3 つのの問題点がある．そこで，それらの問題点を解決し，より多くのプログラムを自動並列化できるように，CellSs を改良する手法を提案する．提案手法では，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．これにより，プログラマが並列に実行する箇所やその単位を意識すること無く，逐次プログラムを，複数の SPE を使用して並列実行

```
1  int main(void){
2     // 計算量が軽量なループ
3    for(i = 0; i < NUM; i++){
4      array[i] = i;
5    }
6
7     // I/O 処理を含むループ
8    for(i = 0; i < NUM; i++){
9      printf("%d,", array[i]);
10   }
11 }
```

図 5: 並列化すべきで無いループの例

されるプログラムに変換可能にする．また，タスクを並列実行するためにリダクション演算が必要な場合，各タスクの結果を統合するコード (以降リダクションコード) を自動生成する．さらに，タスク内で間接参照によりデータにアクセスするプログラムであっても，動作を保証できるようにプログラムを変換する．以降， 3.1 節でタスクの自動切り出しについて， 3.2 節でリダクションコードの自動生成について， 3.3 節で間接参照に対応するための変換について述べる．

3.1 タスクの自動切り出し

既存の CellSs を用いる場合，プログラマが明示的にタスクを指定する必要がある．そのため，プログラマはどの箇所をタスクに指定すべきか，及びタスクの粒度をどの程度にすべきかを考えなければならない．そこで，提案手法では逐次プログラムからタスクを自動的に切り出す機能を提供する．これにより，プログラマによる指示無しに，逐次プログラムを，複数の SPE を使用して並列実行されるプログラムに変換することが可能になり，プログラマが並列性を意識する必要が無くなる．このとき，逐次プログラム中のループを並列化の対象とし，タスクとして切り出すこととする．これは，一般的にループは並列性を持つことが多いためである．

しかし，ループでも並列化すべきでない場合が存在する．並列化すべきでないループの例を**図 5** に示す．図 5 の 3-5 行目のループのボディでは，配列 array の各要素に

変数 i の値を代入する処理のみを行っており，その計算量は非常に軽量である．そのため，このループの各イタレーションを並列に実行したとしても，並列実行によって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にかかる時間の方が上回ってしまう．そのため，このようなループは，並列化するとかえって実行時間が増加してしまう．このようなループ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，実行時間を静的に見積もることは難しい．そのため，このような計算量が非常に軽量なループは，プログラマが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．プリプロセッサに指示を出す具体的な方法については，後の 4.1.1 項で述べる．

また，図 5 の 8-10 行目のループは，配列 array の各要素を順番に出力するループである．このループの各イタレーションをを並列に実行すると，出力される要素の順番が変わってしまうことがある．このように，ループ内に I/O 処理を含む場合は，並列実行するとその結果を一意に決定できないため，I/O 処理を含むループは並列化の対象から自動的に除外する．

以上の方針により，並列化すべきでないループを並列化対象から除外し，並列化の対象とするループを決定した後，そのループからタスクを生成する．生成されるタスクでは，並列化対象ループの各イタレーションが実行される．このとき，SPE プログラムの起動コストを削減するため，1 タスクで 1 イタレーションを実行するのではなく，1 タスクで複数のイタレーションを実行するようにタスクを生成する．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，LS の空き容量やループ内で扱う変数のデータ量を考慮して自動的に決定する．そのため，プログラマは，切り出されるタスクの粒度を意識する必要は無い．

3.2 リダクションコードの自動生成

並列に実行される複数の処理において，ある共通する 1 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．このとき，単純に並列化すると処理の結果が一意に決定しないため，リダクション演算が必要になる．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．各タスクは，リダクション演算が必要である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

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

図 6: 読み出し及び書き込みの判定

変数の値を逐次的に統合する．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し，リダクションコードを自動的に生成・挿入する．

プログラム中の各変数が，リダクション演算が必要な変数であるか否かの判定には，次の条件を用いる．

条件 (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 変数に対して読み出し及び書き込みを行っていることが挙げられる．ある処理で変数からデータを読み出す場合，他の処理の同じ変数への書き込みの結果によって，読み出されるデータは異なるが，それらの処理を並列に実行すると，処理の順が一定でなくなるため，読み出されるデータも一意に定まらない．

条件 (2) 処理の実行順序に依存関係が無い

ある処理の結果によって，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，処理の順序に依存関係がある (以降，順序依存がある) か否かを判定する．

各変数が上記の 2 つ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．以降，この条件の具体的な判定方法について述べる．

条件 (1) 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 6** 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); といった代入文において，図 6 中の 2 行目のように (左辺) が var であれば書き込みと判定し，3 行目のように (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 4 行目のように+=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，1 行目のように if 文や for 文，while 文の条件文に var

```
1  for(i = 0; i < loop; i++){
2    var = (var + num1[i]) * 2;              // 加算と乗算を混在して使用
3  }
4  for(i = 0; i < loop; i++){
5    var = var / num2[i] - 2;                // 減算と除算を混在して使用
6  }
7  for(i = 0; i < loop; i++){
8    var = var + 2;                          // 加算のみを使用
9  }
10 for(i = 0; i < loop; i++){
11   var = var / num1[i] * (num2[i] + 10); // 乗除算のみを使用
12 }
```

図 7: 加減算と乗除算を混在して使用しているか否かの判定

が使われている場合は読み出しのみを行っていると判定する．

条件 (2) は，次の 4 つの条件に分けられる．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．

条件 (2-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

条件 (2-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その if 文のボディで代入されている

条件 (2-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

条件 (2-D)　ライブラリレベルで定義されている関数の呼び出しの際に，対象変数を引数として渡している

以降，**図 7**，**図 8**，**図 9** に示すプログラムの例を用いて，それぞれの条件について具体的に述べる．図中のループを並列化の対象とし，各ループのイタレーションを並列に実行する場合を考える．なお，説明では変数 var を対象変数とする．また，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 7 を例に，条件 (2-A) について説明する．図 7 の 2 行目は，var に対して num1 の値を加算してから 2 を乗算しているが，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 5 行目は，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混在して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

```
1 for(i = 0; i < loop; i++){
2   if(var > num1[i]) var = num1[i] + 10;  // 比較した値に加算してその値を代入
3 }
4 for(i = 0; i < loop; i++){
5   if(var > num1[i]) var = num2[i];       // 比較した値と異なる値を代入
6 }
7 for(i = 0; i < loop; i++){
8   if(var > num1[i]) var = num1[i];       // 比較した値と等しい値を代入
9 }
```

図 8: 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

結果が変わってしまう．そのため，1-3 行目のループと 4-6 行目のループの各イタレーションは，実行順を変えることができない．一方，8 行目は var に対して+のみ適用しており，11 行目は var に対して/と*のみを適用している．このような場合は，対象変数に対して処理を適用する順番を入れ替えても最終的な結果は変わらない．そのため，7-9 行目のループと 10-12 行目のループの各イタレーションはどのような順番で実行しても，結果は変わらない．

次に図 8 を例に，条件 (2-B) について説明する．図 8 の 2 行目は，条件文で比較した num1[i] に 10 を加算し，その結果を var に代入している．また 5 行目は，条件文で比較した num1[i] とは異なる値 num2[i] 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 の値が変わってしまう．そのため，1-3 行目のループと 4-6 行目のループの各イタレーションは，実行順を変えることができない．一方 8 行目は，条件文で比較した num1[i] を，var に代入している．これは，いわゆる num1[i] の最小値を求める処理であり，num1[i] 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．そのため，7-9 行目のループの各イタレーションはどのような順番で実行しても，結果は変わらない．

次に図 9 を例に，条件 (2-C) について説明する．図 9 では，2 行目で var に対して 10 を加算し，2 行目でその結果を num1[i] に代入している．このように，値が書き換えられた var を num1[i] に代入すると，イタレーションの実行順によって最終的な num1[i] の値が変わるため，実行順を変える事ができない．

最後に，条件 (2-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

```
1 for(i = 0; i < loop; i++){
2   var += 10;      // var の値を書き換え
3   num1[i] = var; // 値が書き換えられた var を代入
4 }
```

図 9: 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

内でどんな処理がされているか不明であるため，ライブラリ関数の引数に対象変数を指定した場合，その変数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，各関数に引数として渡した場合に順序依存があるか否かは，予め解析することが可能なため，その解析結果を用いて判定する．以上の方法により，条件 (1) と条件 (2) を満たしているか否かを判定し，2 つの条件を満たしている場合，その変数をリダクション演算が必要な変数と判定する．

リダクション演算が必要な変数が検出された場合，その変数毎にリダクションコードを生成する．リダクション演算は結果を逐次的に統合するが，生成されたリダクションコードは SPE で実行される．これは，PPE よりも SPE の方が演算性能が高く，1 SPE で実行する場合でも，PPE で実行する場合より高速な演算が期待できるからである．よって，SPE で実行するために，リダクションコードはタスクとして生成する．

しかし，既存の CellSs ランタイムライブラリには，生成したリダクションコードを実行する機能が無いため，リダクションコードを挿入するだけでは，一時変数に格納された各タスクの結果を統合することはできない．そのため，提案手法では，各タスクの結果を，リダクションコードを用いて自動的に統合するように，CellSs のランタイムライブラリを拡張する．これにより，プログラマはリダクション演算が必要か否を考えてプログラムを記述する必要がなくなる．

3.3 間接参照への対応

既存の CellSs では，タスク内で間接参照を用いてデータにアクセスするプログラムの動作は保証されていない．これは SPE がメインメモリを直接参照できないことが原因である．正しく間接参照できない場合の例を**図 10** に，正しく間接参照できる場合の例を**図 11** に示す．図 10 の例ではポインタの指す領域 (以降，参照先領域) がメインメモリに配置されており，ポインタもメインメモリのアドレスを指している．先述した通り，SPE はメインメモリを直接参照することができないため，この場合は正しく参

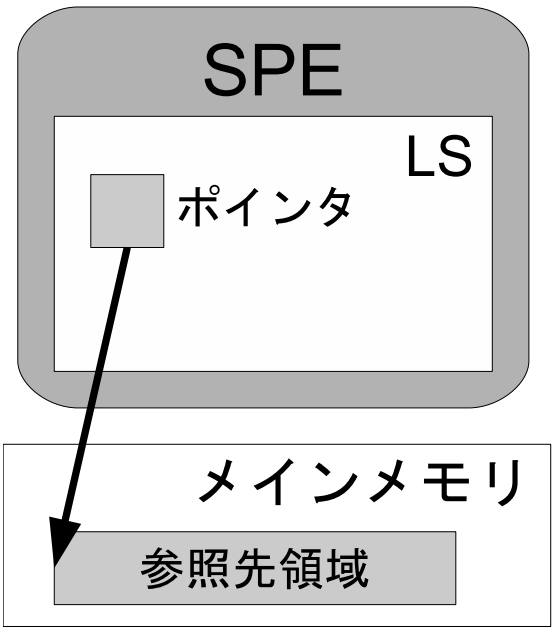

図 10: 正しく参照できない場合

図 11: 正しく参照できる場合

照することができない．一方，図 11 では参照先領域が LS に配置されており，ポインタも LS のアドレスを指しているため，正しく参照することができる．

タスク内で間接参照を用いてデータにアクセスするプログラムを，既存の CellSs を用いて動作させると図 10 のような状況になり，正しく参照がなされない．これは，既存の CellSs が，変換前の逐次プログラムと，変換後の PPE プログラム，SPE プログラムのアドレス空間の違いを考慮していないためである．**図 12** に変換前のプログラムのメモリアクセスの様子を，**図 13** に変換後のプログラムのメモリアクセスの様子を示す．CellSs において，変換前のプログラムでは，2.2 節で述べた通り，タスクは関数として記述されている．そのため図 12 のように，タスクとタスクの呼び出し部はともに，1 つのプログラム中に存在する．このとき，タスクとタスクの呼び出し部で，メインメモリのアドレス空間を共有しているため，ポインタの値のみを引数としてタスクに渡し，タスク内でそのアドレスにアクセスすることで，そのアドレスのデータを正しく参照することができる．しかし，図 13 のように，変換後のプログラムではタスクは SPE プログラムに，タスクの呼び出し部は PPE プログラムに記述される．そして，PPE プログラムはメインメモリの，SPE プログラムは LS のアドレス空間を持つ．そのため，変換前と同様にポインタ変数の値をタスクに渡すと，そのアドレスはメインメモリのアドレスであるため，データを正しく参照できない．変換後の SPE プログラムで，データを正しく参照するためには，図 11 のように参照先領域を LS に配置す

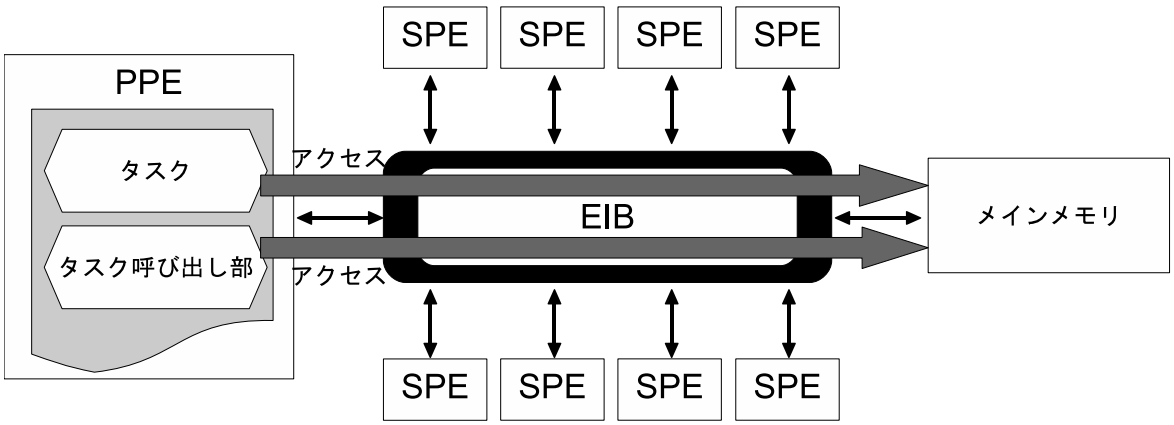

図 12: 変換前のプログラムのメモリアクセス

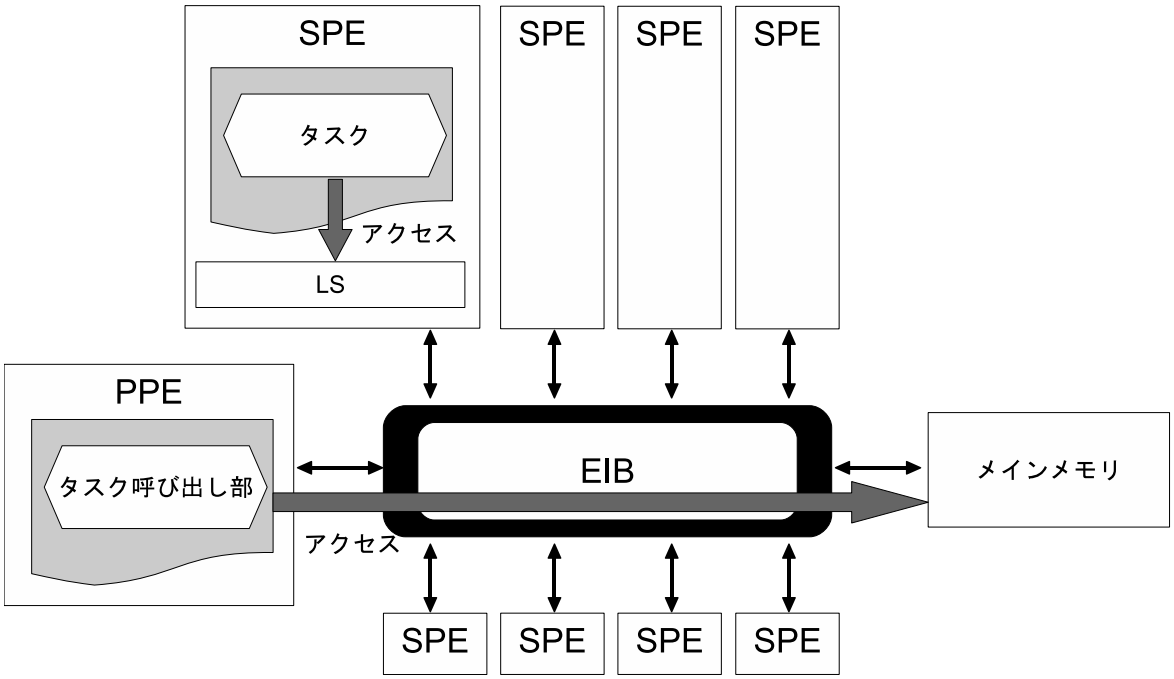

図 13: 変換後のプログラムのメモリアクセス

る必要がある．

そこで，提案手法ではタスク内での間接参照の動作を保証するために，ポインタの値と同時に参照先領域も LS に転送する．しかし，単純に参照先領域を転送しただけでは，ポインタの指すアドレスはメインメモリのアドレスのままである．タスク内で正しく間接参照するには，図 11 のようにポインタの指す先が LS に転送した参照先領域のアドレスでなければならない．そのため，提案手法では参照先領域を LS に転送した後，ポインタの指す先を LS 上の参照先領域のアドレスに書き換える．このようにする

ことで，タスク内で間接参照するプログラムの正しい動作を保証することができる．

4　解析部の実装

提案手法により，逐次プログラムを，複数の SPE を使用して並列に実行されるプログラムへ自動変換するにあたり，プログラムを解析することで変換に必要な情報を得る必要がある．本章では，そのプログラム解析部の実装について述べる．

4.1　ループの解析

逐次プログラムからタスクを切り出す際，プログラム中のループを並列化対象とする．そのため，プログラムの変換に必要な情報をループを解析して取得する．解析により，並列化対象とするループを抽出し，そのループ内で扱う変数のデータ量を見積もる．

4.1.1　並列化対象ループの抽出

提案手法では逐次プログラム中のループをタスクとして切り出し，複数の SPE を使用して並列に実行する．しかし， 3.1 節で述べたように，I/O 処理を含むループや，計算量が軽量であるため並列実行の効果が得られないループは並列化対象から除外する．そのため，そのような並列化対象外のループを検出し，それ以外のループをタスクとして切り出す．

まず，I/O 処理を含むループの検出について述べる．一般的に，C プログラム中で I/O 処理を行うためには，`printf` や `fprintf` といった特定のライブラリ関数を使用する．そのため，ループ中にこれらのライブラリ関数の呼び出しを検出した場合，そのループを I/O 処理を含むループと判定し，並列化の対象から除外する．

一方，計算量が軽量なループは検出が難しい．並列化の効果が得られるか否かは，並列化によって削減される実行時間と，並列化によって発生するオーバヘッドを比較する必要がある．しかし，これらの実行時間を静的な解析から見積もることは困難である．そのため，プラグマによる明示的な並列化対象の除外指定により，計算量が軽量なループを並列化することによる低速化に対処する．並列化対象除外用のプラグマとして，新たに `css nonparallel` を定義する．**図 14** に `css nonparallel` を使用したプログラムの例を示す．図 14 中の 3-6 行目のループは，配列 `arrayA`, `arrayB` の各要素を初期化しており，その計算量は小さい．そのため，2 行目のように，ループの直前に `css nonparallel` を記述し，プリプロセッサに対して，並列化の対象から除外するように指示を出す．これにより，プリプロセッサは 8-13 行目のループのみを並列化

```
1  int main(void){
2  #pragma css nonparallel
3    for(i = 0; i < NUM; i++){
4      arrayA[i] = i;
5      arrayB[i] = 0;
6    }
7
8    for(i = 0; i < NUM; i++){
9      for(j = 0; j < NUM; j++){
10       power *= arrayA[j] + i;
11     }
12     arrayB[i] = sqrt(power);
13   }
14 }
```

図 14: 並列化対象除外用プラグマの使用例

の対象とする．ただし，`css nonparallel` による明示的な並列化対象の除外は，プログラムを低速化させないようにするためのものであり，必ずしも必要では無い．そのため，`css nonparallel` の使用の有無により，プログラムの実行結果が変わってしまうことは無い．以上の判定により，並列化すべきでないループを並列化の対象から除外し，除外されなかったループをタスクとして切り出す．

4.1.2 扱うデータ量の見積り

逐次プログラムを変換してタスクを生成する際，SPE プログラムの起動にかかるオーバヘッドを削減するために，1 タスクに複数のイタレーションを実行させる．1 タスクに割り当てるイタレーションの数 (以降，割り当てイタレーション数と呼ぶ) は，LS の容量と 1 イタレーションで扱われるデータ量を考慮して決定する．これは，LS の容量は有限であり，あまり多くのイタレーションを割り当てると，扱うデータ量が LS の容量を越えてしまうからである．CellSs では，タスクの実行に必要なデータや，タスクの実行結果を，タスクの実行の前後に一括で転送するため，1 タスクで扱われるデータ量が LS の容量を超えてはいけない．

割り当てイタレーション数を決定するとき，配列と配列以外の変数のデータ量を別々

```
1  int main(void){
2    int    i;
3    char   minus[NUM];
4    double sum;
5    double arrayA[NUM], arrayB[NUM];
6
7    for(i = 0; i < NUM; i++){
8      if(minus[i] > 0){
9        sum -= arrayA[i] * arrayB[i];
10     } else {
11       sum += arrayA[i] * arrayB[i];
12     }
13   }
14 }
```

図 15: プログラム例 (データ量見積り)

に考慮することとする．これは，配列は割り当てイタレーション数が増えると扱うデータ量も増えるが，配列以外の変数は割り当てイタレーション数にかかわらず扱うデータ量は一定であるためである．$Variable$ を配列以外のデータ量，$Array$ を 1 イタレーションで扱う配列のデータ量，Ite を割り当てイタレーション数，$Capacity$ を LS の空き容量とすると，Ite の値は次の条件を満たす必要がある．

$$Variable + Array \times Ite \leq Capacity \tag{1}$$

図 15 の例を用いて，扱うデータ量の具体的な見積り方について述べる．変数のデータ量は環境によって異なるが，今回は char 型を 1byte，int 型を 4byte，double 型を 8byte であるとする．この例ではループ中で配列以外の変数 `i`，`sum` と，配列 `minus`，`arrayA`，`arrayB` を扱っている．変数のそれぞれの型は図 15 の 2-5 行目の変数宣言を解析することで取得する．

まず，配列以外の変数のデータ量を求める．配列以外の変数は int 型変数が 1 つ，double 型変数が 1 つであるので，$4 + 8 = 12$byte となる．次に，配列のデータ量を求める．この例では 1 イタレーションで，char 型配列を 1 要素，double 型配列を 2 要素扱っ

```
1 int main(void){
2   int i;
3   for(i = 0; i < loop; i++){
4     sum += d[i + (i / 3) * 3];
5   }
6 }
```

図 16: 扱うデータが連続していない例 (プログラム)

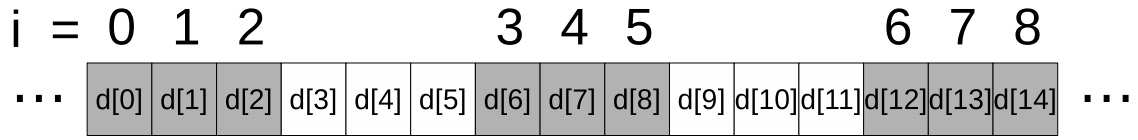

図 17: 扱うデータが連続していない例 (データアクセス)

ているので，$1 + 8 \times 2 = 17$byte となる．これらを式 (1) に当てはめると，この例では Ite が満たすべき条件は $12 + 17 \times Ite \leq Capacity$ である．$Capacity$ の値が 128Kbyte であるとすると，Ite は 7,709 以下の数でなければならないことが分かる．

しかし，扱うデータ量が LS の空き容量よりも少ない場合でも，LS にデータを転送できない場合がある．**図 16** に転送できない場合のプログラム例を示す．この例では配列 d の要素に添字をいてアクセスしており，添字は $i + (i \div 3) \times 3$ の式で計算される．i は int 型であるため，$i \div 3$ の余りは計算時に切り捨てられ，i が 3 増加するたびに $(i \div 3) \times 3$ の結果は 3 増加する．この例におけるデータアクセスの様子を，**図 17** に示す．図 17 中の灰色の箇所がアクセスされる要素であり，上部の数字は要素がアクセスされる際の i の値である．この例では，i = 0,1,2でアクセスする d[0]，d[1]，d[2] は連続しているが，i = 3 でアクセスする d[6] と d[2] は連続していない．同様に，d[6]，d[7]，d[8] は連続しているが，d[12] と d[8] は連続していない．Cell/B.E. では，連続していないデータを一度の DMA 転送で転送することはできないため，この例では，連続している 3 イタレーション分のデータまでしか一度に転送することができない．そのため，式 (1) を満たす場合でも，3 イタレーションまでしか 1 タスクに割り当てることができない．このように，Ite はタスクの実行に必要なデータが連続していることを保証できる数である必要がある．Ite は以上の 2 つの条件をともに満たす最大の自然数に決定する．

また，動的確保された領域を扱う場合を考慮して，Ite はプログラム実行中に計算する．図 15 の例のように，ループ内で使われているデータを格納している領域が，全て静的に確保されるものであれば，そのデータ量も静的に解析することができる．しかし，動的確保される領域が含まれる場合，そのデータ量は静的に解析することができない．そのため，動的確保される領域については，動的確保時にそのデータ量をデータ量保持用の変数に格納しておき，その値を用いて Ite をプログラム内で計算する．計算された Ite は他の引数と同様にタスクに渡され，タスクはその値に合わせて実行するイタレーション数を変える．このとき，動的確保された領域と，データ量保持用の変数の対応付けを容易にするために，ポインタ変数の値の変更を禁止する．値が代入された後，四則演算や再代入等によって，値が変更されたポインタ変数をループ内で使用している場合，そのループをタスクとして切り出さないことにする．これにより，動的に確保される領域を扱う場合でも，扱うデータ量を考慮して，1 タスクに割り当てるイタレーションを決定することができる．

4.2 リダクション演算が必要な変数の検出

3.2 節で述べたように，ループ内の変数に対して，その変数がリダクション演算が必要な変数であるか否かを判定する必要がある．3.2 節で述べた各条件を，AST(Abstract Syntax Tree) を探索して判定する．CellSs はプログラムを変換する際，AST を生成するため，今回はその AST を用いてプログラムを解析することとした．生成された AST の各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 18** 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

以下の 3 つの点から判定を行う．

- 書き込みを行っているか否か
  代入文の左辺が対象変数である場合，対象変数に対して書き込みを行っていると判定する．
- 読み出しを行っているか否か
  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る場合，対象変数に対して読み出しを行っていると判定する．
- 読み出し及び書き込みの両方を行っているか否か
  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

```
1  void Task(int a, int *var1, int *var2){
2    *var1 = *var1 + a;
3    if(*var2 > a * 2) *var2 = a + 3;
4    *var1 += ++(*var2);
5    *var1 = a + *var1 - a * 10;
6    *var2 = (*var2 - a) / 10 + a;
7    funcLib(var2, a);
8  }
9
10 int main(void){
11   for(i = 0; i < NUM; i++){
12     Task(a, &var1[i], &var2[i]);
13   }
14 }
```

図 18: プログラム例 (リダクション演算が必要な変数の検出)

デクリメントされている場合，対象変数に対して読み出し及び書き込みを行っていると判定する．

図 18 の 2 行目から生成される AST を**図 19** に，3 行目から生成される AST を**図 20** に，4 行目から生成される AST を**図 21** に示す．代入文の左辺は図 19 のように，= ノードの第 1 子として保持される．そのため，= ノードの第 1 子を探索することで，対象変数に対して書き込みを行っているか否かを判定できる．また，図 20 から代入文の右辺は = ノードの第 2 子に，if 文の条件式は condition ノードをルートとする部分木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，対象変数が読み出しを行っているか否かを判定できる．さらに，図 21 から加算代入演算式の左辺は+=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，対象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できる．

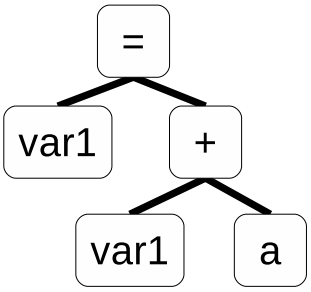

図 19: 代入文

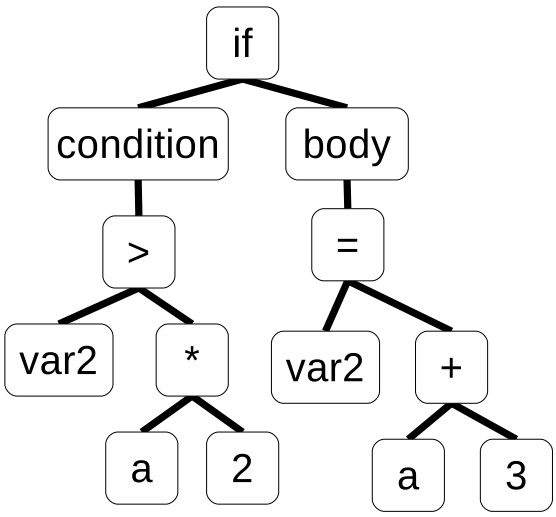

図 20: if 文

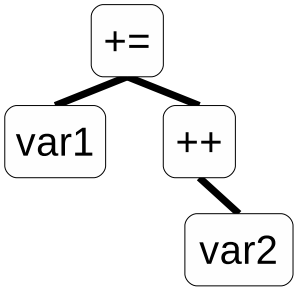

図 21: 複合代入文

条件 (2-A) 加減算と乗除算を混在して使用しているか

図 18 の 5 行目から生成される AST を**図 22** に，6 行目から生成される AST を**図 23** に示す．図 22，図 23 のように，ある変数に適用される演算子のノードは，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子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を判定できる．図 22 では，`var1` を遡っていくと `-` ノードと `+` ノードしか存在しないので，`var1` には加減算しか適用されていないことが分かる．一方，図 23 では，`var2` を遡っていくと `-` ノード，`/` ノード，`+` ノードが存在するので，`var2` には加減算と乗除算を混在して適用していることが分かる．

条件 (2-B) 比較した値と異なる値が代入されているか

if 文の条件式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否かを判定する．この条件の判定は，まず，条件式の中で対象変数が使われているか否かを判定し，使用されていた場合，if 文のボディでその対象変数に対して

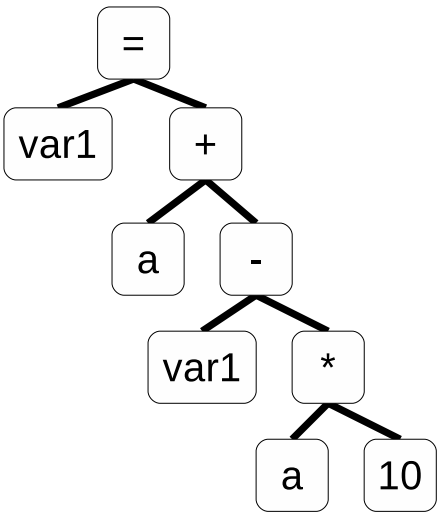

図 22: 複数の演算が使用された文 (1)

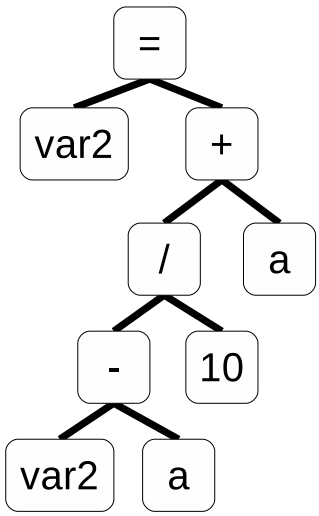

図 23: 複数の演算が使用された文 (2)

値が代入されているか否かを判定する．対象変数に値が代入されていた場合は，その値が条件式で比較した値と等しいか否かを判定する．

図 20 に示すように，if 文の条件式は `condition` をルートとする部分木に保持されているため，この部分木を探索することで，対象変数が条件式で使用されているか否か判定できる．対象変数が条件式で使用されていた場合，`body` ノードをルートとする部分木を探索し，対象変数が `=` ノードの第 1 子にある場合，`=` ノードの第 2 子が条件式で比較した値と同じであるか否かを判定する．この判定では比較演算子ノードの対象変数以外の子と，`=` 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する．

図 20 の例では，まず `condition` をルートとする部分木を探索すると，部分木の中に `var2` が存在するため，`var2` が条件式で使われていることが分かる．そのため，次に `body` をルートとする部分木を探索する．この部分木を探索すると `=` ノードの第 1 子に `var2` が存在するため，if 文のボディで `var2` に値を代入していることが分かる．そのため，条件式で `var2` と比較する値を保持する，`*` ノードをルートとする部分木と，代入する値を保持する，`+` ノードをルートとする部分木を比較する．すると，部分木が一致しないため，条件式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 (2-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

対象変数の値が書き換えられたか否かを判定し，書き換えられていた場合，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か否かを判定する．この判定では，値が書き換えられたか否かを記憶しておくため，変数毎にフラグを用意する．対象変

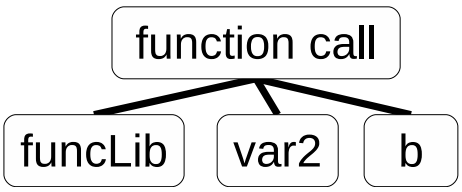

図 24: 関数呼び出し

数の値が書き換えられる処理か否かは，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．対象変数の値が書き換えられたと判定されたら，その変数に対応するフラグを立て，他の変数に代入しているか否かの判定に移る．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 子が他変数である代入演算子ノードの，第 2 子をルートする部分木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する．対象変数の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する．以上の方法で判定するためには，実行される順番で各文を探索する必要がある．CellSs で生成される AST では，深さ優先で探索することで，実行される順番で各文を探索可能であるため，この条件の判定では深さ優先で探索する．

図 20, 図 21 を例に判定の様子を説明する．なお，図 20 の AST に対応する式はプログラム中で図 21 に対応する式よりも先に出現するものとし，このため AST も図 20, 図 21 の順に検索されるものとする．まず，図 20 の AST を探索すると = ノードの第 1 子に `var2` が存在するため，`var2` に対応するフラグを立てる．ここで，この代入文は if 文の中にあり，実行されるか否かは分からないが，本手法では実行されるものとして判定する．そして，図 21 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 子をルートとする部分木に `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 (2-D) ライブラリレベルで定義されている関数の引数として対象変数を渡しているか

図 18 の 7 行目から生成される AST を**図 24** に示す．

呼び出す関数の名前は，`function call` ノードの第 1 子に保持されているため，それを基に，プログラマが記述したコードの中にその関数が定義されているか否かを調査する．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として対象変数を渡しているか否かを判定できる．図 18 の例では，プログラム中に funcLib 関数の定義が記述されていないため，ライブラリ関数の引数として var2 を渡していることが分かる．

4.3 ポインタ変数の解析

タスク内で間接参照を用いてデータにアクセスする場合，正しい動作を保証するためには，参照先領域を転送し，ポインタ変数の値を書き換える必要がある．そのため，解析により間接参照を用いたデータのアクセスを検出する．タスクはループから生成されるため，間接参照を用いているか否かの判定はループ内に対してのみ行えばよい．間接参照を用いているか否かは，ポインタ変数を解析することで判定する．並列化対象となるループのボディで，ポインタ変数やポインタ変数をメンバに持つ構造体を使用しているか否かを解析し，それらを使用していた場合，そのループは間接参照を含むと判定する．そして，そのループをタスクとして切り出した後に，参照先領域の転送やポインタの書き換えを行うコードを挿入する．

ただし，ツリーやリストのような再帰的な構造を持つデータを使用するループは，並列化の対象外とする．再帰的なデータ構造では，間接参照を再帰的に繰り返すため，実行時の間接参照の回数は不定である．間接参照を複数回用いる場合，LS に転送しなければならないデータはその回数に応じて増加するが，提案手法では，静的な解析の結果によって，LS に転送するデータの数を決定するため，間接参照の回数が不定である再帰的なデータ構造を転送することはできない．よって，再帰的な構造を持つデータをタスク内で使用することは禁止とし，そのようなデータを使用するループはタスクとして切り出さないことにする．

ループ内で使用しているデータが，再帰的な構造を持つか否かは，構造体の宣言部からメンバの型を解析することで判定する．**図 25** に示す構造体の宣言の例を用いて，具体的な判定方法について説明する．まず，図 25 中の 1-4 行目の StrcutA_t 型の宣言部を解析する．すると，この型は構造体のポインタ変数をメンバに持つことが分かる (図 25, 3 行目)．そのため，このメンバの型である StructB_t 型の宣言部を解析する．すると，この型も構造体へのポインタ変数をメンバに持つことが分かる (図 25, 8 行目)．このとき，このメンバの型は既に解析済みの StrcutA_t であるため，この 2 種類の構造体は再帰的なデータ構造を持つことが分かる．このように，構造体のメンバの型を順に解析していき，同じ構造体の型を検出した場合，その構造体は再帰的なデータ構造を持つと判定する．

```
1 typedef struct {
2   int       a;
3   StructB_t *ptr;
4 } StructA_t;
5
6 typedef struct {
7   int       b;
8   StructA_t *ptr;
9 } StructB_t;
```

図 25: 構造体の宣言部

5　コード生成部の実装

本章では，4 章で述べた解析の結果を基に，提案手法を用いて，逐次プログラムを，複数の SPE を用いて並列実行されるプログラムに変換するための実装について述べる．

5.1　変換フロー

CellSs では図 3 に示したように，逐次プログラムを変換し，複数の SPE を用いて並列に実行されるプログラムを生成する．CellSs のプリプロセッサによる変換の様子を**図 26**，**図 27**，**図 28** に示す． 図 26 は CellSs のプラグマを用いたプログラムの例であり，図 26 のプログラムを CellSs のプリプロセッサによって変換した後のプログラムが図 27 と図 28 である．

CellSs のプリプロセッサによって，逐次プログラムは PPE プログラムと SPE プログラムの 2 つに変換される．PPE プログラムはタスクとして指定された関数以外の部分から構成される．初期化用プラグマ `css start`(図 26, 12 行目) は初期化用関数 `css_init` に (図 27, 4 行目)，終了処理用プラグマ `css finish`(図 26, 16 行目) は終了処理用関数 `css_finish` にそれぞれ変換される (図 27, 9 行目)．そして，タスクとして指定された関数の呼び出しが (図 26, 14 行目)，CellSs のランタイムライブラリの関数である，タスク実行用関数 `css_addTask` の呼び出しに変換される (図 27, 7 行目)．`css_addTask` の引数にはタスクの各引数を保持する配列 `css_parameters` と，タスクの引数の数が渡される．この `css_parameters` には，プリプロセッサによって挿入されるコードによっ

```
1  // タスク指定用プラグマ
2  #pragma css task input(a[64],b[64]) output(c[64])
3  void mul_64(double *a, double *b, double *c) {
4    for(i = 0; i < 64; i++){
5      c[i] = a[i] * b[i];
6    }
7  }
8
9  int main(void){
10   int A[64][64], B[64][64], C[64][64];
11
12 #pragma css start  // 初期化用プラグマ
13   for(i = 0; i < 64; i++){
14     mul_64(A[i], B[i], C[i]);
15   }
16 #pragma css finish // 終了処理用プラグマ
17 }
```

図 26: CellSs を用いたプログラムの例

て，タスクの実行前に，各引数とデータ量等の引数情報が格納される (図 27, 6 行目).

SPE プログラムはタスクとして指定された関数 mul_64(図 26, 1-7 行目，図 28, 1-6 行目) と，それを呼び出す mul_64_adapter_cssgenerated という関数 (図 28, 8-11 行目) で構成される．SPE プログラムでは，CellSs のランタイムライブラリが内部で mul_64_adapter_cssgenerated を呼び出し，この関数内で，メインメモリから LS へ転送される css_parameters を引数として mul_64 を呼び出す (図 28, 10 行目).

提案手法では，以上のようなプリプロセッサによる変換の前後に，4 章で述べた解析を含めたいくつかの処理を加える．提案手法の処理のフロー図を**図 29** に示す.

提案手法では既存の CellSs の変換工程に加え，3 つの工程を追加する．3 つの工程を加えた変換では，まず，逐次プログラムに対して (1) プログラムの解析，(2) フロントエンド変換を行う．その結果を CellSs のプリプロセッサで変換した後，(3) バックエンド変換を行い，最後に変換により得られたプログラムをコンパイラでコンパイルす

```
1  int main(void){
2    int A[64][64], B[64][64], C[64][64];
3
4    css_init();   // 初期化用関数
5    for(i = 0; i < loop; i++){
6      css_parameters[] = { ··· };      // 引数情報の格納
7      css_addTask(css_parameters, 3); // タスク呼出用関数
8    }
9    css_finish(); // 初期化用関数
10 }
```

図 27: プリプロセッサによる変換後 (PPE プログラム)

```
1   // タスクとして指定された関数
2  void mul_64(double *a, double *b, double *c){
3    for(i = 0; i < 64; i++){
4      c[i] = a[i] * b[i];
5    }
6  }
7
8   // タスクを呼び出す関数
9  void mul_64_adapter_cssgenerated(css_paramter_t *css_paramters){
10   mul_64(css_parameters[0],css_parameters[1],css_parameters[2]);
11 }
```

図 28: プリプロセッサによる変換後 (SPE プログラム)

る．(1) プログラムの解析では変換するために必要な情報を，4 章で述べた手法で逐次プログラムを解析することにより得る．(2) フロントエンド変換では逐次プログラム中のループからタスクを生成し，CellSs のプリプロセッサで変換可能なコードに変換する．また，リダクション演算が必要な場合はリダクションコードを生成し，挿入する．続く CellSs のプリプロセッサによる変換によって，既存の CellSs と同様に，逐次プロ

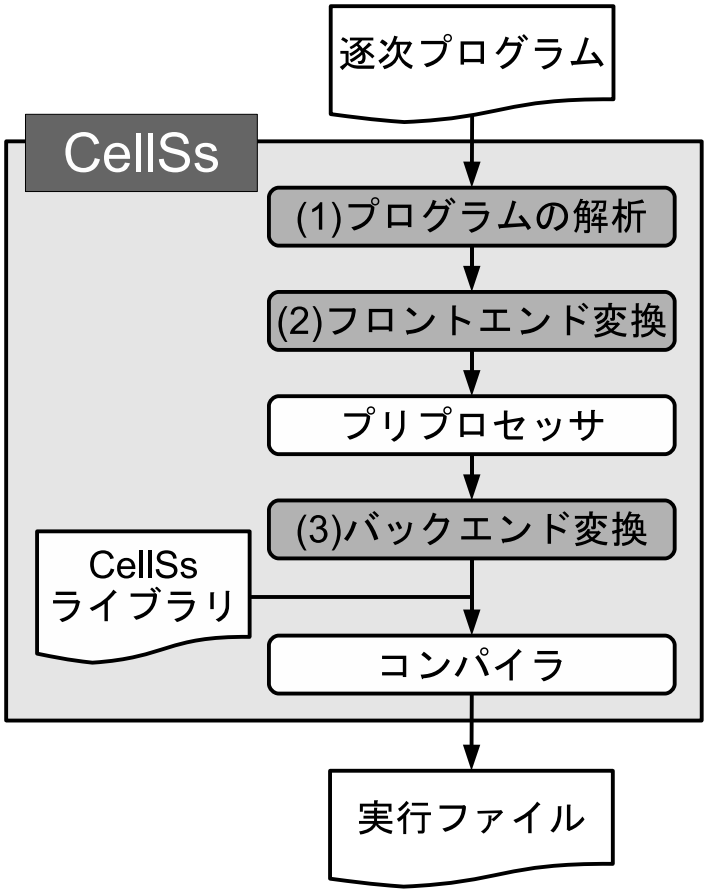

図 29: 提案手法の変換フロー

グラムが PPE プログラムと SPE プログラムに変換される．このとき，PPE プログラムは，タスクとして切り出された箇所以外の実行と SPE プログラムの起動を行うように，また，SPE プログラムは，切り出されたタスクと，リダクションコードを実行するように変換される．最後に，(3) バックエンド変換において，間接参照に対応するための，参照先領域の転送とポインタ張り替えのための変換を行う．

5.2 タスクの生成

解析により検出された並列化対象ループの各イタレーションを，複数の SPE を用いて並列実行されるように変換する．この変換は図 29 の (2) フロントエンド変換で行う．**図 30** のプログラムを変換した例を，**図 31** に示す．

まず，並列化対象ループ (図 30, 9-17 行目) の処理を切り出してタスクを生成する (図 31, 6-18 行目)．タスクは関数として記述され，そのタスクをループのボディで呼び出すように変換する (図 31, 27-31 行目)．このとき，タスクには，並列化対象ループの条件

```
1  typedef struct {
2    int  tag;
3    int *num;
4  } Threshold_t;
5
6  int main(void){
7    Threshold_t thr.num = malloc(sizeof(int) * length);
8
9    for(i = 0; i < loop; i++){
10     for(j = 0; j < length; j++){
11       if(thr.num[j] < num[i]) {
12         diff++;
13       } else if(thr.num[j] > num[i]){
14         diff--;
15       }
16     }
17   }
18 }
```

図 30: 逐次プログラムの例

文やループボディ内で使われている変数 thr, num, length, diff と，変換によって新たに追加された__taskid, __ite という変数を引数として渡す (図 31, 7,8 行目)．__taskid と__ite はタスク内で，イタレーションの初期化やループの終了条件 (図 31, 9 行目)，配列の添え字に用いる (図 31, 11,13 行目)．

__taskid はタスク内のループ制御に用いる変数であり，並列化対象ループの直前で初期化され (図 31, 25 行目)，1 度タスクを呼び出す毎にインクリメントされる (図 31, 29 行目)．__ite は割り当てイタレーション数を保持する変数であり，並列化対象ループの直前でその値を計算する (図 31, 26 行目)．この式中の LS_CAPA は LS の空き容量を表しており，この値とループ中で扱う変数のデータ量を，割り当てイタレーション数の計算に用いる．この例では，変数 thr のメンバ num の参照先領域を動的確保しているので (図 30, 7 行目，図 31, 21 行目)，そのデータ量を，新たに追加された変数__thr_size

```
1  typedef struct {
2    int  tag;
3    int *num;
4  } Threshold_t;
5
6  #pragma css task input(thr,num[__ite],length,__ite,__taskid) inout(diff)
7  void css_task(Theshold_t *thr, int *num, int length, int *diff,
8                int __taskid, int __ite){
9    for(i = __ite*__taskid; i < __ite*(__taskid+1); i++){
10     for(j = 0; j < length; j++){
11       if(thr->num[j] < num[i - __ite*__taskid]){
12         (*diff)++;
13       } else if(thr->num[j] > num[i - __ite*__taskid]){
14         (*diff)--;
15       }
16     }
17   }
18 }
19
20 int main(void){
21   Threshold_t thr.num = malloc(sizeof(int) * length);
22   __thr_size = sizeof(Threshold_t) + sizeof(int) * length;
23
24 #pragma css start
25   __taskid = 0;
26   __ite = (LS_CAPA - (__thr_size + sizeof(int)*4)) / sizeof(int);
27   for(i = 0; i <= loop - __ite; i+=__ite){
28     css_task(&thr, &num[i], length, &diff, __taskid, __ite);
29     __taskid++;
30   }
31   css_task(&thr, &num[i], length, &diff, __taskid, loop % __ite);
32 #pragma css finish
33 }
```

図 31: タスク生成後のプログラムの例

に保持しておき (図 31, 22 行目)，その値を割り当てイタレーション数の計算に用いる (図 31, 26 行目).

次に，ループの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__ite の値を加算する処理に変換し，タスクを呼び出す回数を調節する (図 31, 27 行目)．このとき，変換前のプログラムで実行されるイタレーションの総数が，割り当てイタレーション数の倍数でない場合，変換前と変換後で同様の条件をループの終了条件に用いると，実行されるイタレーション数が変わってしまう．そのため，ループの終了条件を変更する (図 31, 27 行目)．そして，実行すべきイタレーション数を割り当てイタレーション数で除算し，その剰余と同数のイタレーションを，ループ後に実行する (図 31, 31 行目)．これにより，変換前のプログラムで実行されるイタレーション数と同数のイタレーションを実行することができる．例えば，この例において，変数 loop の値が 128，__ite の値が 10 だった場合を考える．その場合，図 31 の 27-30 行目で 12 回タスクが呼び出され，120 イタレーションが実行される．その後図 31 の 31 行目で呼び出すタスクが，剰余分の 8 イタレーションを実行し，合計で 128 イタレーションが実行される．

最後に，変換したプログラムをさらに CellSs のプリプロセッサで変換するために，プラグマを挿入する．この例では 3 種類のプラグマを挿入しており，タスク呼び出し文の前にフレームワークの初期化用プラグマ css start を (図 31, 24 行目)，タスク呼び出し文の後に終了処理用のプラグマ css finish(図 31, 32 行目) を，タスクの前にタスク指定用のプラグマ css task をそれぞれ挿入している (図 31, 6 行目)．なお，タスク指定用のプラグマ css task では，引数の転送方法を指定する必要がある．転送方法は 3 種類あり，メインメモリから LS へ転送する引数を input，LS からメインメモリへ転送する引数を output，メインメモリから LS へ転送し，タスク実行後に LS からメインメモリに転送する引数を inout として指定する．そのため，読み出しのみを行っている引数を input，書き込みのみを行っている引数を output，読み出し及び書き込みを行っている引数を inout として指定する．今回の例では diff はインクリメントやデクリメントをしているので，inout として指定される．それ以外の引数は全て条件式や配列の添字アクセスにのみ使用されているため，input として指定される．以上の変換により，逐次プログラム中のループをタスクとして切り出す.

```
1  // 統合用関数
2 #pragma css task input(piece,original) inout(reduction)
3 void __aggregate_diff_spe(int *piece, int *reduction, int *original){
4    // 加算を用いた統合
5   *reduction += *piece - *original;
6 }
7
8  // 中継用関数
9 void __aggregate_diff_ppe(int *piece, int *reduction, int *original){
10   // 統合用関数の呼び出し
11  __aggregate_diff_spe(piece, reduction, original);
12 }
```

図 32: リダクションコードの例

5.3 自動リダクション

逐次プログラム中のループからタスクを生成した後，そのタスク内からリダクション演算が必要な変数が検出された場合，リダクションコードを生成し，挿入する．このリダクションコードの生成・挿入も，図 29 の (2) フロントエンド変換で行われる．また，生成されるリダクションコードを用いて各タスクの結果を自動的に統合できるように CellSs のライブラリを拡張する．

5.3.1 リダクションコードの生成

4.2 節で述べた解析により，リダクション演算が必要な変数を検出した場合は，リダクションコードを生成し，挿入する．図 30 の例から生成されるリダクションコードを**図 32** に示す．この例において，リダクション演算が必要な変数は，変数 diff である．

リダクションコードは，__aggregate_変数名_spe と __aggregate_変数名_ppe という 2 つの関数から構成され，この 2 つの関数はリダクション演算が必要な変数毎に生成される．以降，この 2 つの関数をそれぞれ統合用関数 (図 32, 1-6 行目)，中継用関数 (図 32, 8-12 行目) と呼ぶ．この 2 つの関数はそれぞれ 3 つの引数をとり，piece には各タスク実行後のリダクション演算が必要な変数の値が，reduction には全てのタスクのリダクション結果が，original にはタスク実行前の初期値が格納される (図 32, 3,9 行目)．なお，reduction の初期値は original と同じ値である．

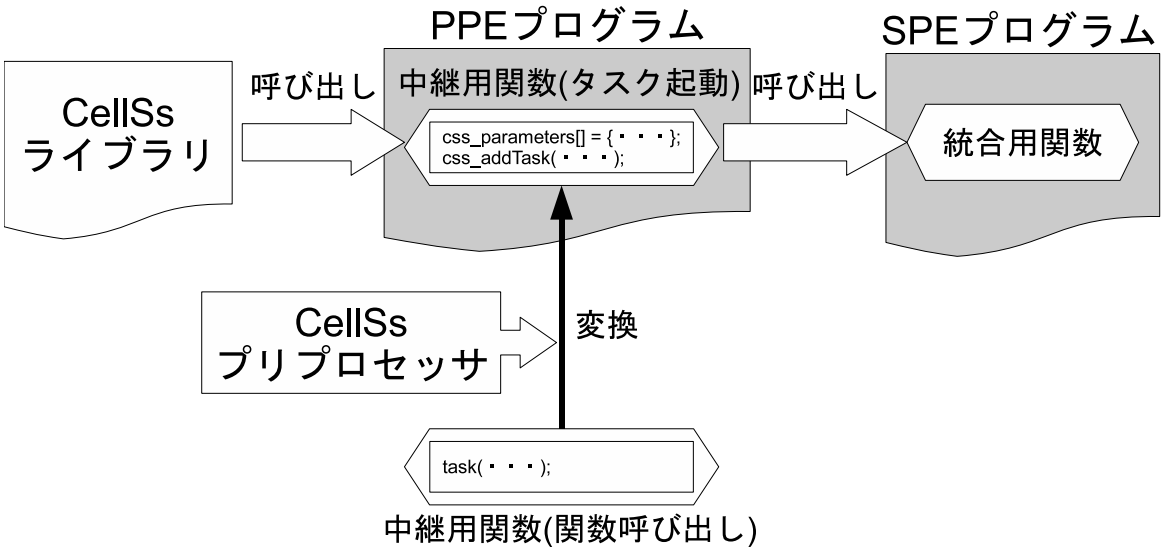

図 33: 中継用関数を用いた，統合用関数呼び出し

統合用関数は全てのタスクの結果を統合する関数であり，この関数内の処理は変数毎に異なる．この例では，リダクション演算が必要な変数 `diff` は，タスク内で加減算されているため (図 31, 12,14 行目)，加算を用いて統合する (図 32, 5 行目)．`diff` のリダクションでは，`piece` から `original` を減算することで，初期値からの差分を求め，それを `reduction` に足し込む．この統合用関数は，SPE で実行するため，タスク指定用プラグマ `css task` を挿入しタスクとして指定する (図 32, 2 行目)．

中継用関数は統合用関数を呼び出す関数である．各タスクの結果を統合する際，リダクションコードを CellSs のランタイムライブラリ内で実行する．これにより，プログラマにリダクション演算を意識させることなく，自動的に各タスクの結果を統合する．しかし，統合用関数を CellSs のランタイムライブラリ内から直接呼び出すことはできない．統合用関数を SPE で実行するためには，統合用関数をタスクとして指定する必要があるが，タスクを実行するためには，タスクに合わせてプログラムの記述をプリプロセッサで変換する必要がある．そのため，ランタイムライブラリ内から直接タスクを呼び出すことにすると，プログラム毎に，ランタイムライブラリ内の記述を CellSs のプリプロセッサで変換しなければならず，その度にランタイムライブラリをコンパイルしならない．よって，中継用関数を用いて，ランタイムライブラリ内から統合用関数を呼び出すこととする．**図 33** に，中継用関数を用いて統合用関数を呼び出す様子を示す．生成した中継用関数を PPE プログラム中に挿入し，それを CellSs のプリプロセッサで変換する．プログラム実行時に，ランタイムライブラリ内から中継用

関数を呼び出すと，その中継用関数が統合用関数を呼び出す．これにより，ランタイムライブラリ内で統合用関数を実行することができる．

以上の 2 つの関数を用いることにより，全てのタスクの結果を統合する．以上のコードを，リダクションコードとして生成する．

5.3.2 CellSs ランタイムライブラリの拡張

5.3.1 項で述べたように，リダクション演算が必要な変数がある場合，リダクションコードを生成し，挿入するが，それだけでは，タスクの結果を統合することはできない．生成されたリダクションコードを用いて，タスクの結果を自動的に統合するために CellSs のランタイムライブラリを拡張する必要がある．

まず，リダクションコードを使用するために必要な情報を，CellSs のランタイムライブラリ内で取得するために，既存の CellSs ランタイムライブラリで定義されている構造体を拡張する．ライブラリ内でタスクの結果を統合するためには，リダクション演算が必要な変数とそうでない変数を区別する必要がある．そのため，CellSs のランタイムライブラリ内で定義されている，各引数の情報を管理するために使われる構造体に，その引数がリダクション演算が必要な変数であるか否かを示すフラグ (以降，リダクションフラグ) を新たに追加する．引数がリダクション演算が必要な変数である場合にそのフラグをセットし，そうでない場合はフラグをリセットする．

各タスクの結果の統合は，中継用関数を介して統合用関数を呼び出すことで行われる．しかし，これらの関数はリダクション演算が必要な変数毎に生成されるため，変数毎に呼び出さなければならない関数が異なる．そのため，ライブラリ内で正しく関数を呼び出すには，変数と関数を関連付ける必要がある．そこで，先述した引数情報管理用の構造体に，その引数に対応する呼び出し用関数の関数ポインタを保持するメンバも追加する．ライブラリ内ではこの関数ポインタを用いることで，対応する中継用関数を呼び出すことができる．

先述した，各引数の情報は，CellSs のプリプロセッサによって生成されるコードにより，それを管理するための構造体へ格納される．そのため，CellSs のプリプロセッサを拡張し，リダクションフラグと中継用関数の関数ポインタを引数情報として格納するコードを生成するようにする．**図 34** に，拡張した引数情報管理用の構造体の宣言と，図 31 のプログラムをプリプロセッサで変換した際に生成されるコードを示す．なお，図 34 では `length` と `diff` 以外の引数の情報については省略している．引数の情報は上から順に引数のアドレス，引数のデータ量，転送方式，リダクションフラグ，中継用関数の関数ポインタである (図 34, 3-7 行目)．CellSs のプリプロセッサによって，こ

```
1  // 引数情報を格納する構造体
2  typedef struct {
3    void *address;       // アドレス
4    size_t size;         // データ量
5    long direction;      // 転送方式
6    int reduction_flag; // リダクションフラグ
7    void *aggr_p;        // 中継用関数の関数ポインタ
8  } css_parameter_t;
9
10 css_parameter_t css_parameters[];
11 css_parameters = { … ,// 引数情報の格納
12                    {&length, sizeof(int), INPUT, 0, NULL},
13                    {&diff,   sizeof(int), INOUT, 1, __aggregate_diff_ppe},
14                     … };
```

図 34: 引数情報格納コードの例

れらの情報を引数毎に格納するようにコードを生成する (図 34, 11-14 行目)．引数のアドレス，引数のデータ量，転送方式については，既存の CellSs プリプロセッサが生成するコードを用いる．提案手法で新たに追加した，リダクションフラグと中継用関数の関数ポインタの値は， 4.2 節で述べた手法により得た情報によって決定される．ある引数が，リダクション演算が必要な変数と判定された場合，リダクションフラグをセットし，対応する関数関数ポインタを格納する．そうでない場合は，リダクションフラグをリセットし，NULL ポインタを格納する．この例では，diff はリダクション演算が必要な変数であり，length はリダクション演算が必要無い変数である．以上のようなコードを生成するように，プリプロセッサを拡張することで，新たに追加した引数情報をライブラリ内で扱うことができる．

次に，新たに追加した引数情報を用いて，リダクションコードを実行し，各タスクの結果を自動的に統合するように，CellSs のランタイムライブラリを拡張する．CellSs のランタイムライブラリでは，プログラムの実行時に引数のアドレスを解析し，アドレスが等しい引数は同一の変数であると判断する．そして，その情報を基にタスク間の依存関係を解析し，各タスクをスケジューリングすることで，タスクの結果を一意

に決定している．よって，リダクションコードを用いることで競合を回避できる場合でも，既存のライブラリの動作では，タスク間に依存関係があると判断してしまい，各タスクが逐次に実行されてしまう．そこで，リダクションコードによって競合を回避できる場合は，タスク間の依存関係が無いと判断するようにライブラリを拡張する．

CellSs では，タスクを実行するために `css_addTask` 関数を呼び出した際，タスク間の依存関係を解析する．`css_addTask` 関数を呼び出すと，まず，タスクの状態を待機状態にする．そして，実行中のタスクと待機状態のタスクの依存関係を解析する．この依存関係の解析では，各タスクの引数にアドレスが等しいものがあるか否かを判定する．等しいものがある場合，その引数の転送方式を調べる．引数の転送がメインメモリから LS への転送のみであれば，その引数は読み出しのみ行うと判断し，LS からメインメモリへの転送がある場合は，書き込みを行うと判断する．読み出しのみ行う場合は，タスクを待機状態にした時にその引数の値は既に渡されているため，依存関係は無い．しかし，書き込みを行う場合は依存関係があるため，CellSs ではそのタスクが終了するまで，タスクを実行しないようにする．このとき，引数がリダクション演算が必要な変数であると判定されている場合，リダクション演算を用いることで競合は回避可能なため，本研究の改良により依存関係は無くなる．そのため，拡張した CellSs のランタイムライブラリでは，リダクションフラグがセットされていた場合，他のタスクの転送方式に関わらず，依存関係は無いと判定する．これにより，リダクション演算を用いることで競合が回避可能なタスクも並列に実行される．

拡張した CellSs のランタイムライブラリの具体的な動作を，**図 35** に示すプログラムを例に説明する．この例は，あるタスクを引数を変えて 3 回実行するプログラムである．この例で実行されるタスクは 2 つの引数を持ち，ともに書き込みを行う引数である．また，第 2 引数はリダクション演算が必要な変数である．まず，最初に `css_addTask` を呼び出す際 (図 35, 7 行目) は，既に実行しているタスクや，待機状態のタスクが無いため，そのままこの 1 つ目のタスクを実行する．次に，もう一度 `css_addTask` を呼び出す (図 35, 12 行目) と，実行中のタスクと第 1 引数のアドレスが等しいため，2 つ目のタスクは待機状態になり，1 つ目のタスクの終了を待つ．一方，3 回目に `css_addTask` を呼び出す際 (図 35, 17 行目) は，実行中の 1 つ目のタスクと第 2 引数のアドレスが等しいが，第 2 引数はリダクションフラグがセットされているため，1 つ目のタスクの終了を待たずに 3 つ目のタスクを実行する．

全てのタスクが終了すると，CellSs では終了処理を行う．終了処理では，一時変数に格納されている各タスクの結果を元の変数に書き戻す．このとき，書き戻す変数が

```
1  int main(void){
2    css_init();
3
4     // 1 つ目のタスクの実行
5    css_parameters[] = {{&a[0], sizeof(int), INOUT, 0, 0},
6                        {&b[0], sizeof(int), INOUT, 1, __aggregate_b_ppe}};
7    css_addTask(css_parameters, 2);
8
9     // 2 つ目のタスクの実行
10   css_parameters[] = {{&a[0], sizeof(int), INOUT, 0, 0},
11                       {&b[1], sizeof(int), INOUT, 1, __aggregate_b_ppe}};
12   css_addTask(css_parameters, 2);
13
14    // 3 つ目のタスクの実行
15   css_parameters[] = {{&a[1], sizeof(int), INOUT, 0, 0},
16                       {&b[0], sizeof(int), INOUT, 1, __aggregate_b_ppe}};
17   css_addTask(css_parameters, 2);
18
19   css_fiish();
20 }
```

図 35: 複数のタスクを実行する例

リダクション演算が必要な変数である場合，全ての結果を統合して，その値を書き戻すようにライブラリを拡張する．CellSs において，各タスクの結果はリスト構造で管理されているため，そのリストを辿り，各タスクの結果をリダクションコードを用いて順に統合する．リダクション演算は SPE でタスクとして実行されるため，すべての結果を統合し終わるまでタスクの終了を待ち，終了した後その結果をメインメモリの元の変数に書き戻す．以上の拡張により，CellSs のランタイムライブラリ内でリダクションコードを用いて，自動的にタスクの結果を統合することができる．

5.4 参照先領域の転送

タスク内で間接参照を用いてデータにアクセスするプログラムを正しく動作させるため，CellSs のプリプロセッサによって変換されたプログラムをさらに変換する．この変換は，図 29 の (3) バックエンド変換で行われる．図 31 を CellSs のプリプロセッサで変換し，その出力をさらにバックエンド変換した後のプログラムを**図 36** と**図 37** に示す．なお，本来は `css_parameters` には，データ量や転送方式といった引数の情報も格納されるが，この図ではそれらを省略している．(図 36, 15,16,22,23 行目)．

図 36 では，参照先領域を LS に転送するために，参照先領域である `thr.num` を引数に加えるように変換している (図 36, 16,23 行目)．その際，転送する領域の数が増えるため，`css_addTask` の第 2 引数の値を，転送先領域の数だけ増加させる (図 36, 18,25 行目)．この例では引数を 1 つ増やしたため，第 2 引数の値を 6 から 7 に増加させている．これにより，他の引数と同様に参照先領域をメインメモリから LS へ転送することができる．

図 37 では，間接参照に用いるポインタが LS のアドレスを指すように，ポインタの値を書き換えている (図 37, 21,23,27 行目)．メインメモリ上のアドレスを変数 `__thr_num_p` に退避しておき (図 37, 21 行目)，ポインタの値を LS に転送した参照先領域のアドレスに書き換える (図 37, 23 行目)．これにより，タスク内で正しく領域を参照することができる．そして，タスク実行後にポインタの値をメインメモリ上のアドレスに戻す (図 37, 27 行目)．これら 2 つの変換によりタスク内での間接参照の動作を保証する．

6 評価

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．今回評価に用いた環境では，搭載されている 8 基の SPE のうち，1 基は冗長性確保のため使用することができず，1 基は OS に常に確保されているため，アプリケーションが使用可能な SPE は最大で 6 基である．

評価には以下のプログラムを使用した．

- montecarlo：モンテカルロ法を用いた円周率の計算
- integral：積分法を用いた面積の計算
- dft：離散フーリエ変換
- matmul：行列積の計算
- lu：疎行列 LU 分解
- opticalflow：勾配法を用いたオプティカルフローの計算

```
typedef struct {
  int  tag;
  int *num;
} Threshold_t;

int main(void){
  Threshold_t thr.num = malloc(sizeof(int) * length);
  __thr_size = sizeof(Threshold_t) + sizeof(int) * length;

  css_init();
  __taskid = 0;
  __ite = (LS_CAPA - (__thr_size + sizeof(int)*4)) / sizeof(int);
  for(i = 0; i <= loop - __ite; i+=__ite){
     // 参照先領域を追加
    css_parameters[7] = {&thr,  &num[i],  length,
                         &diff, __taskid, __ite, thr.num};
     // 第 2 引数の値を増加
    css_addTask(css_parameters, 7);
    __taskid++;
  }
   // 参照先領域を追加
  css_parameters[7] ={&thr,  &num[i],  length,
                      &diff, __taskid, loop % __ite, thr.num};
   // 第 2 引数の値を増加
  css_addTask(css_parameters, 7);
  css_finish();
}
```

図 36: バックエンド変換後のプログラムの例 (PPE プログラム)

```
1  typedef struct {
2    int  tag;
3    int *num;
4  } Threshold_t;
5
6  void css_task(Theshold_t *thr, int num[], int length, int *diff,
7                int __taskid, int __ite) {
8    for(i = __ite*__taskid; i < __ite*(__taskid+1); i++){
9      for(j = 0; j < length; j++){
10       if(thr->num[j] < num[i - __ite*__taskid]){
11         (*diff)++;
12       } else if(thr->num[j] > num[i - __ite*__taskid]){
13         (*diff)--;
14       }
15     }
16   }
17 }
18
19 void css_task_adapter_cssgenerated(css_paramter_t *css_paramters){
20    // メインメモリのアドレスを退避
21   __thr_num_p = css_parameters[0]->num;
22    // LS のアドレスに変更
23   css_parameters[0]->num = css_parameters[6];
24   css_task(css_parameters[0],css_parameters[1],css_parameters[2]
25            css_parameters[3],css_parameters[4],css_parameters[5]);
26    // メインメモリのアドレスに戻す
27   css_parameters[0]->num = __thr_num_p;
28 }
```

図 37: バックエンド変換後のプログラムの例 (SPE プログラム)

表 1: 評価環境

プラットフォーム	PLAYSTATION3
CPU	Cell/B.E.
動作周波数	3.2GHz
使用 SPE 数	6
OS	Fedora 10
Cell Superscalar version	2.2
コンパイラ	ppu-gcc 4.3.2
	spu-gcc 4.3.2
最適化オプション	-O2

これらの内，matmul と lu は CellSs のサンプルプログラムであり，既存の CellSs を用いて実行する場合でも，Cell/B.E. を十分に活用できるプログラムである．また，opticalflow は OpenCV のソースを基に記述したプログラムである．なお，CellSs を用いた並列プログラムにおいて，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ため，montecarlo と integral においては，SPE の数も考慮して，600 程度のタスクを実行することにした．

評価の結果を**図 38** に示す．縦軸は実行時間を表しており，既存の CellSs による実行時間を 1 として正規化した．グラフは左が既存の CellSs，右が提案手法を実装した CellSs の結果を表している．なお，opticalflow はタスク内で間接参照を用いてデータにアクセスするプログラムであり，既存の CellSs を用いて動作させた場合では正しく動作しないため，逐次プログラムを PPE のみを用いて実行した場合と比較した．また，図 38 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 はリダクション処理にかかった時間である．

図 38 より，montecarlo, integral において，提案手法を用いることで大きく速度向上できていることが分かる．これらのプログラムでは，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，全てのタスク間で依存関係があるため結果的に 1 基の SPE しか同時に動作しない．一方で，提案手法を用いた場合，リダクション演算の追加により競合を回避することで，6 基の SPE を用いて並列実行可能となり，このような性能向上が得られた．

しかし，dft では montecarlo, integral に比べあまり大きな速度向上は得られなかっ

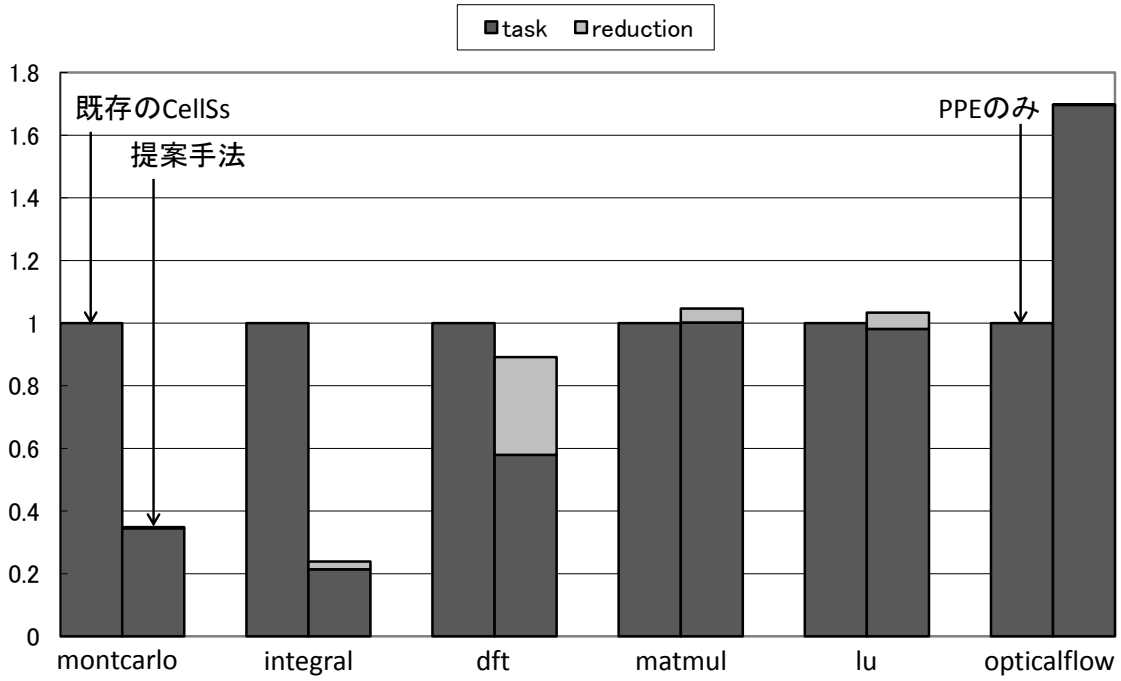

図 38: 評価結果

た．この原因はリダクションのオーバヘッドにある．dft は montecarlo, integral と同様に，リダクション演算の追加により競合を回避することで，6 基の SPE を用いて並列実行可能となるプログラムであり，タスクの実行に要する時間を削減することができている．しかし，montecarlo, integral に比べ，リダクションの処理に要する時間が非常に長いことが分かる．この違いは実行するタスク数の差が原因である．各タスクの結果を統合する時には，統合するタスクの結果と同じ数だけ統合用関数が実行される．統合用関数自体もタスクであり，統合用関数を 1 回実行する毎にタスクの起動オーバヘッドがかかるため，統合する結果の数が多い程大きなオーバヘッドがかかる．各プログラムで実行されるタスクの数は，montecarlo, integral では 600 程度，dft では 80,000 程度となっており，これらのタスクは全てリダクションが必要なタスクである．そのため，dft では montecarlo, integral に比べ，非常に大きなリダクションのオーバヘッドがかかってしまい，このような結果となった．

また，mutmul, lu では，数%程実行時間が増加してしまった．matmul, lu では，montcarlo, integral, dft と異なり，全てのタスクに依存関係が存在するわけではなく，一部のタスクの間にのみ依存関係が存在する．そのため，既存の CellSs を使用する場合でも，6 基の SPE を用いて並列にタスクを実行することができる．よって，タスクの実

行時間は既存の CellSs を使用した場合と提案手法を使用した場合で変わらない．そして，提案手法を使用した場合はタスクの実行時間に加え，リダクションの時間がかかるため，このような結果となった．これら 2 つのプログラムにおいて，既存の CellSs を用いた場合と提案手法を用いた場合で，タスクの実行時間が変わらない理由は，使用できる SPE の数にある．今回評価を行った環境では，SPE を 6 基までしか利用できない．そのため，すべての SPE を活用するには，並列に実行可能なタスクの数が 6 以上であれば良い．しかし，利用できる SPE の数がより多い環境では，より多くのタスクが並列に実行可能でなければ，全ての SPE を活用することはできない．matmul, lu の 2 つのプログラムでは，提案手法を用いることで，既存の CellSs を用いて実行する場合よりも，並列に実行可能なタスクの数は増加している．そのため，より多くの SPE が利用可能な環境では提案手法による高速化が期待できる．

さらに，opticalflow では，他のプログラムとは異なり，大きく速度低下していることが分かる．opticalflow について詳細に調査した結果，速度低下の原因はページフォルトであることが分かった．opticalflow では扱うデータ量が大きく，そのデータを LS に転送する際，メインメモリ上にそのデータのページが無いため，DMA 転送の度にページフォールトが発生していた．ページフォールトの影響を調べるため，DMA 転送前に転送データに 0 を代入する初期化処理を加え，実行時間を比較した．その結果を**図 39** に示す．グラフは左が逐次プログラムを PPE のみで実行した結果，中央が提案手法を用いて変換したプログラムの結果，右が提案手法を用いて変換し初期化処理を加えたプログラムの結果である．また，凡例は task がタスクの実行にかかった時間，reduction がダクションにかかった時間，init が初期化処理にかかった時間である．グラフから，提案手法で変換したプログラムに初期化を加えたことにより実行速度が向上し，PPE のみを用いて実行する場合よりも高速化していることが分かる．今回評価に用いた PLAYSTATION3 は，メインメモリの容量が 256Mbyte と小さく，ページフォールトが発生しやすい．そのため，解析時にメモリ使用量を調査し，使用量が大きい場合，データ転送の直前に初期化処理を行うコードを自動挿入することで，ページフォールトの発生を防ぐことができると考えられる．

しかし，init の時間を考慮しても，6 基の SPE を使用しているにもかかわらず，PPE のみで実行した場合と比べ 2 割程度の速度向上しか得られていない．これには，2 つの理由が考えられる．1 つ目の理由は，SPE が 6 基動作していない時間があることである．opticalflow では計算に必要なデータを生成するタスクと，オプティカルフローを計算するタスクの，2 種類のタスクが生成されるが，データを生成するタスクはリダ

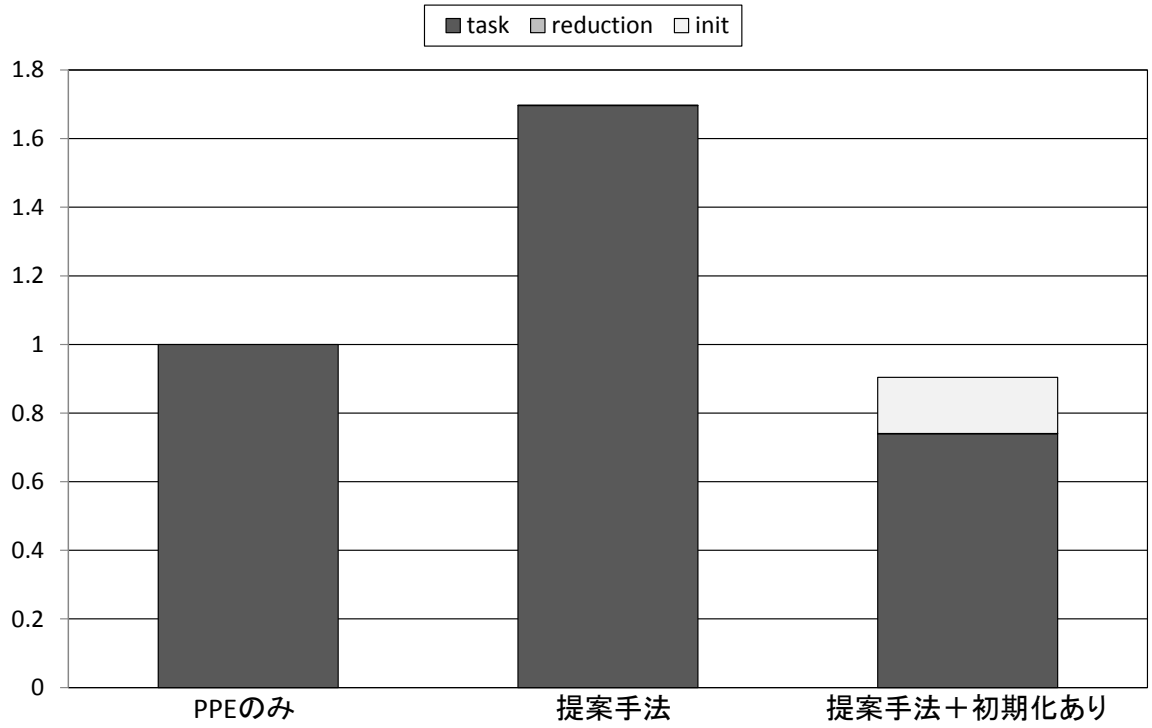

図 39: 初期化処理を加えた opticalflow の実行時間

クション演算を用いても競合を回避できないため，逐次的に実行される．もう一方のタスクではその結果を用いるため，データを生成されるタスクが終了するまで，実行することができず，結果的に 1 基の SPE しか動作していない時間がある．2 つ目の理由は，転送するデータ量が大きいことである．先述した通り，opticaloflow は扱うデータ量が大きいため，データを転送に要する時間が長くなってしまった．データ転送の間を隠蔽するために，ダブルバッファリングがよく用いられるが，CellSs ではダブルバッファリングを用いていないため，このように転送時間が大きなオーバーヘッドとなることがある．

7 関連研究

本研究では，Cell/B.E. 向けのアプリケーション開発を支援する開発環境を取り扱ったが，Cell/B.E. 以外のマルチコア環境向けのアプリケーション開発の支援も研究されている．

並列プログラミングを支援するフレームワークとしては，OpenMP[7] や Intel の TBB (Intel Threading Building Blocks)[8] が挙げられる．OpenMP を使用してアプリケーションを開発する場合，プログラマがプラグマを用いて並列化箇所を指示し，それを

コンパイラが解釈することで並列化プログラムを生成する．一方，TBB を使用する場合は，プログラマが並列実行したい処理をタスクとして記述し，TBB の API を用いることでそのタスクを並列に実行するプログラムを記述できる．これらを用いた場合，プログラマはスレッドの生成や，処理の割り当てなどを記述することなく，並列プログラムを記述できる．しかし，既存の CellSs と同様に，プログラマが並列に実行される箇所を意識してプログラムを記述しなければならないという問題点がある．また，これらは共有メモリしかサポートしておらず，Cell/B.E. のような分散メモリ環境向けの記述はできない．

これに対し，分散メモリをサポートする環境として，CUDA (Compute Unified Device Architecture)[9] が挙げられる．CUDA は GPU 向けの開発環境であり，プログラマは CUDA を用いることで，グラフィックスプログラミングの知識が無くとも GPU を活用できる．また，GPU はいくつかのローカルなメモリを搭載しており，CUDA を用いると，それらのメモリの領域確保や，メモリ間のデータ転送を記述することができる．しかし，CUDA を利用するには，GPU や CUDA に関する深い知識が必要である．そのため，CUDA をサポートする環境として OpenMPC[10] が研究されている．OpenMPC では，OpenMP のプラグマで指示されたプログラムを GPGPU 向けのプログラムに変換する．そのため，OpenMPC を使用する場合，プログラマは GPU や CUDA に関する知識を必要としない．しかし，OpenMPC は OpenMP のプラグマを使用するため，OpenMP と同様の問題点がある．

また，GPU や Cell/B.E. など多くの環境をサポートするフレームワークとして OpenCL (Open Computing Language)[11] がある．OpenCL を用いて開発されたアプリケーションでは，GPU やマルチコア CPU などを用いて処理を並列に実行する．OpenCL ライブラリを用いて実装されたアプリケーションは，デバイスや OS などのプラットフォームが変化しても，ソースコードレベルでの互換性が保持されるという特徴がある．しかし，ハードウェアの抽象化によってオーバヘッドが発生してしまい，速度向上を得にくいという問題点がある．また，OpenCL が想定するプラットフォームモデルには，計算ユニットと，それを制御・管理するホストが存在し，プログラマは，それぞれに対してプログラムを記述する必要がある．

以上のように，マルチコア環境向けのアプリケーション開発を支援する様々な環境が研究されている．これらの環境を用いることで，プログラマは，利用するアーキテクチャの構造を意識することなく，アプリケーションの開発が可能になる．しかし，これらを用いた場合でも，並列化箇所の指示や，コアに合わせたプログラムの書き分け

は必要である．一方，本論文で提案した手法を用いた場合，逐次プログラムから並列化箇所を自動的に抽出し，並列プログラムに変換する．そのため，プログラマは従来のシングルコア向けのプログラムと同様の記述をするだけで良く，提案手法は有用であると考えられる．

8 おわりに

本論文では，Cell/B.E. 向けフレームワーク CellSs の問題点を述べ，それを解決するために，CellSs に対して適用可能なプログラム範囲を拡大することによる改良を施すことを提案した．改良点は 3 つあり，1 つ目は，逐次プログラムからタスクを自動的に切り出すことで，プログラマの指示無しにプログラムの変換を可能にした．これにより，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，複数の SPE を使用して並列実行されるプログラムを生成することが可能な環境を提供した．2 つ目は，リダクション演算が必要な変数を検出し，リダクションコードを生成・挿入することで，タスクの結果を自動的にリダクションすることを可能にした．これにより，既存の CellSs では逐次的に実行されていたタスクの一部を，並列に実行することが可能になり，プログラムを書き換えること無く，最大で約 4.1 倍の速度向上が得られることを確認した．3 つ目は，タスク内での間接参照の正しい動作を保証するようなプログラムへの変換を可能にした．これにより，既存の CellSs では動作が保証されていないプログラムも，正しく動作することを確認した．

今後の課題として，より広い範囲のプログラムへの対応や，様々なリダクションコードの生成，データ転送の効率化が挙げられる．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，プログラム中にはループの他にも並列性を持つ箇所は存在する．またループであっても，値を変更したポインタ変数を使用する場合や，リストのような再帰構造を使用する場合は並列化の対象から除外している．そのような箇所に対しても並列化を適用可能にすることで，さらに Cell/B.E. の性能を引き出すことができると考える．

また，今回の実装で自動生成可能なリダクション演算は，総和・総乗を計算する処理や，最大値・最小値を求める処理などの，単純な処理に限られている．そのため，より複雑なリダクション演算を生成可能にすることで，より多くのプログラムで，タスクを並列実行可能になると考えられる．

また，評価に用いた opticalflow は，SPE での計算に要する時間に比べ，データ転送に要する時間が長いプログラムであった．そのため，データを効率的に転送すること

で，opticalflow はさらに速度向上すると考えられる．本論文で提案した手法では，1 タスクで複数のイタレーションを実行するため，あるイタレーションを実行している間に，次のイタレーションで扱うデータを転送することで，転送時間を隠蔽することができると考えられる．

謝辞

本研究のために多大な御尽力を頂き，日頃から熱心な御指導を賜った名古屋工業大学の津邑公暁准教授，松尾啓志教授，齋藤彰一准教授，松井俊浩准教授に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室，齋藤研究室，松井研究室の方々に深く感謝致します．

参考文献

[1] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

[2] Bellens, P., M.Perez, J., M.Badia, R. and Labarta, J.: CellSs: a Programing Model for the Cell BE Architecture (2006).

[3] Fixstars Corporation: CTK: Cell ToolKit Library, http://cell.fixstars.com/ctk/.

[4] Fixstars Corporation: OpenCV on the Cell, http://cell.fixstars.com/opencv/index.php/OpenCV_on_the_Cell.

[5] Bradski, G. and Kaehler, A.: *Learning OpenCV: Computer Vision With the OpenCV Library*, O'Reilly & Associates Inc (2008).

[6] Intel Corp.: *Open Source Computer Vision Library* (2001).

[7] Dagum, L. and Menon, R.: OpenMP: an Industry Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering*, Vol. 5 (1998).

[8] Intel Corporation: *Intel(R) Threading Building Blocks Reference Manual*, 1.9 edition (2008).

[9] NVIDIA Corp.: *NVIDIA CUDA Programming Guide*, 2.0 edition (2008).

[10] Lee, S. and Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs, *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, IEEE press (2010).

[11] Khronos OpenCL Working Group: *The OpenCL Specifiction*, 1.0 edition (2009).