卒業研究論文

Cell/B.E.向けスケルトンライブラリ BlockLibの評価

指導教官

松尾 啓志 教授

津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科

平成 17 年度入学　17115080 番

鈴木　麗佳

平成 21 年 2 月 10 日

Cell/B.E. 向けスケルトンライブラリ BlockLib の評価

鈴木　麗佳

内容梗概

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はかっている．また，比較的単純なアーキテクチャで高い性能を出すことができる SIMD 命令を充実させ，ベクトル処理性能を高めるという手法も，最近のプロセッサでは用いられている．その一つに，ヘテロジニアスマルチコアプロセッサの CELL があり，高い処理性能を目指した SIMD 命令を持つが，その高い処理性能を生かすためには，高度なプログラミング技術が必要であり，Cell プログラマーを育成する妨げとなっている．そのため今日，Cell プログラミングの煩わしさを少しでも軽減できるようなライブラリが検討および開発されている．

本論文ではその一つであるスケルトンライブラリ BlockLib に注目したが，現在配布されている BlockLib ライブラリおよび，それを構成する NestStep ライブラリは libspe1 に準拠した仕様となっており，最新の Cell SDK 3.1 に含まれる libspe2 の仕様ではない．このため，BlockLib ライブラリを Cell SDK 3.1 を用いて動作させるために，現在配布されている NestStep ライブラリを libspe2 に準拠した仕様に変更を行い，スケルトンライブラリ BlockLib 上で動作するプログラムの実行速度と BlockLib を用いないプログラムの実行速度を比較し，評価を行った．

評価の結果，libspe1 と libspe2 を用いて記述したプログラムはほぼ同じ実行時間が得られ，ライブラリを使用したことによるオーバーヘッドの時間が発生するが，自分の手で記述し最適化を施したプログラムに近い実行速度を出すことができた．

本稿では，BlockLib であらかじめ用意されているテストプログラムのみを動作対象としたため，テストプログラムで用いていない関数は libspe2 には対応していない．そのため，今後は全ての NestStep ライブラリ関数の修正を行いたい．また，現在の BlockLib では未実装の Cycric Distribute Array を用いたプログラムを実装し，Block Distribute Array と比較し，Cell/B.E. 上で有効なデータ配列の決定を図る．

Cell/B.E. 向けスケルトンライブラリ BlockLib の評価

目次

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はかっている．また，比較的単純なアーキテクチャで高い性能を出すことができるSIMD 命令を充実させ，ベクトル処理性能を高めるという手法も，最近のプロセッサでは用いられている．

一方でCell Broadband Engine (以下Cell/B.E.) [1] は，ヘテロジニアスマルチコアプロセッサの1つであり，SONY，東芝，IBM の3社によって共同で開発されたプロセッサである．Cell/B.E. はエンタテイメント向け専用機に搭載することを主な目的として，高い処理性能を目指したSIMD 命令を持つマルチコアプロセッサである．1つの汎用プロセッサPPE (PowerPC Processor Element) と8 つの演算プロセッサ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．

しかし，その高い処理性能を生かすためには，高度なプログラミング技術が必要であり，Cell/B.E. プログラマを育成する妨げとなっている．そのため，プログラミングの煩わしさを少しでも軽減できるようなライブラリが検討されている．

本論文では，Cell/B.E. プログラミングに関する技術的障壁を少しでも緩和することを目的としたスケルトンライブラリBlockLib に注目した．現在配布されているBlockLib ライブラリは，BlockLib を開発したLinköping University で同じく提唱されたNestStep と呼ばれるライブラリ上で動作する．Cell/B.E. でSPE を使用するプログラムを記述する場合には，libspe と呼ばれるライブラリが必要である．libspe の仕様には，libspe1 とlibspe2 が存在する．現在ではlibspe2 が主流となりつつあるが，NestStep ライブラリはlibspe1 に準拠した仕様となっており，IBM によって提供されている最新のCell SDK 3.1 では動作しない．このため，BlockLib ライブラリをCell SDK 3.1 を用いて動作させるためには，現在配布されているNestStep ライブラリをlibspe2 に準拠した仕様に移植する必要がある．

本研究では，NestStep ライブラリをlibspe1 からlibspe2 に対応した仕様に移植するとともに，libspe2 に移植したNestStep およびその上で動作するスケルトンライブラリBlockLib 上で動作するプログラムの実行速度を評価する．2 章では，NestStep ライブラリ 及びBlockLib ，Cell/B.E. のアーキテクチャ，動作の仕組について概略を述べ

る．第 3 章では，NestStep ライブラリの移植をする際に修正が必要な関数及び，プログラム記述に関する注意事項について述べる．4 章では，移植した NestStep ライブラリ及び，BlockLib の動作を 2 種類の Cell/B.E. で評価を行い，最後に 5 章で結論をまとめる．

2 背景

本章では，本研究で取り扱ったスケルトンライブラリ BlockLib，NestStep，および Cell/B.E. について述べる．

2.1 Cell/B.E.

Cell/B.E. は，テレビゲーム機に搭載することを主な目的として SONY，東芝，IBM の 3 社により共同開発された，高い処理性能を目指したマルチコア SIMD プロセッサである．Cell/B.E. は， 1 つの汎用プロセッサ PPE (PowerPC Processor Element) と 8 つの演算プロセッサ 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．シングルスレッド時の性能よりもむしろ，マルチスレッド時の性能を目指したプロセッサであり，9 つのコアをあわせた浮動小数点演算能力は最大時で 200GFLOPS を超える．Cell/B.E. アーキテクチャの概略を図 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いる．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いる．SPE はそれぞれ 256KB のローカルストア (以下 LS) と呼ばれるスクラッチパッドメモリを持つ．メインメモリへのアクセスは LS を介してのみ行う．また，SPU（Synergistic Processor Unit）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ず，Memory Flow Controller (以下 MFC) と呼ばれるユニットを利用する必要がある．この LS とメモリ間でのデータ転送に擁する時間は非常に大きいため，メモリレイテンシを隠蔽する手法として，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，片方のバッファがメインメモリにアクセスしている裏で，もう片方のバッファで計算を行うという方法である．

また SPE は，128bit の SIMD 演算を行うパイプラインを持ち，LS へのアクセスを行うパイプラインと合わせて，2Way のスーパースカラパイプライン構造となってい

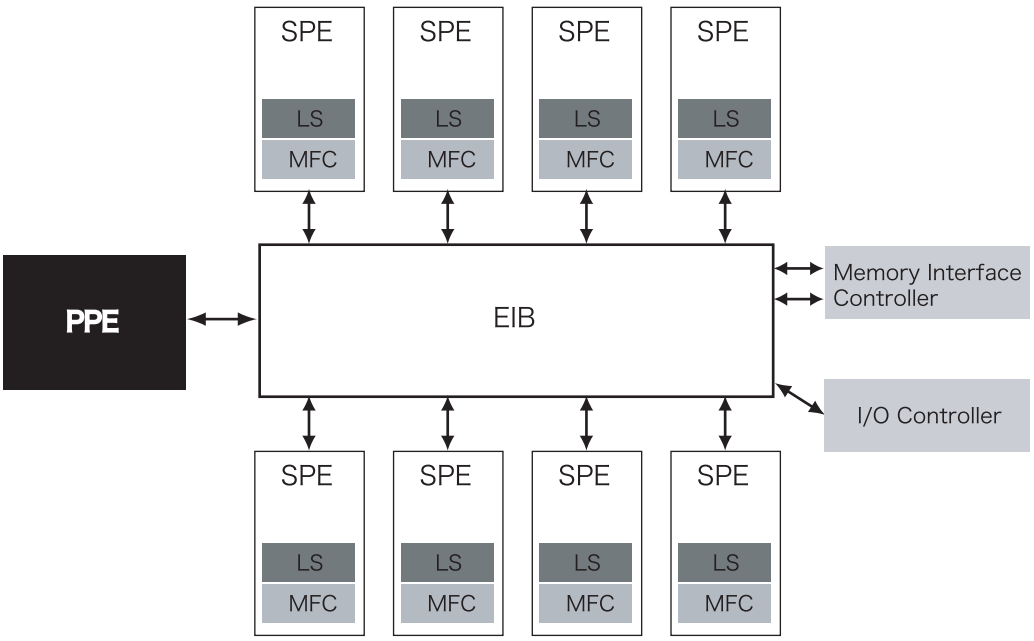

図 1: Cell/B.E. アーキテクチャ

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，複数のデータに対する処理を行うことで，レイテンシを隠蔽したプログラミングが可能である．一方で、Cell/B.E. は独特なアーキテクチャであるが故に，プログラム開発を行う際に注意を払う必要がある．これを具体的に，以下に挙げる．

(1) Cell/B.E. の特徴を活かしたプログラムの開発にはまず，Cell/B.E. に搭載されている，性質が異なる 2 種類のコア (PPU と SPU) で動作するプログラム (PPE プログラムと SPE プログラム) をそれぞれのプロセッサに対して用意する必要があり，かつそれらが協調するような設計にする必要があるため，並列分散プログラミングの技術が必要となる．

(2) 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，ハードウェアのアーキテクチャ的な詳細を理解する必要がある．これは DMA 転送と呼ばれる Cell/B.E. 独自のデータ転送方式の理解や，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を理解する必要があるためである．

また、Cell/B.E. の種類によっては，搭載されている SPE の個数が異なったり，Cell/B.E. のハードウェア提供ベンダやその上に載せる OS，低レベルライブラリによって SPE を制御する方法が異なるため，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 は特に，SPU SIMD 命令等の組み込み関数を用いることで，同じような計

算を単純に繰り返すようなマルチメディア系の処理を得意としている．しかしながら，コンパイラなどによる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ようなプログラムの箇所を抽出することがまだ困難であるため，SPE の演算性能を引き出すような SPE プログラミングの技術的障壁はかなり高いと言える．

2.2 NestStep

NestStep は，並列計算用 BSP モデルを採用した並列プログラミング言語である．NestStep について述べる前に，まず BSP モデルについて簡単に述べる．

2.2.1 BSP モデル

BSP（Bulk Synchronous Parallel ）[2] モデルとは，1990 年に Valiant らによって実装されたもので，Oxford 大学により提案された並列アーキテクチャのひとつである．これはプログラムを計算，大域通信，バリア同期の 3 ステップの繰り返しで計算されると考えるモデルである．この 3 ステップをまとめて superstep と呼び，3 つのステップは具体的に以下の 3 つの役割を担う．

(1) 各プロセッサ上で計算を行う段階．各プロセッサは局所変数もしくはリモート変数のコピーのみに対してアクセス可能である．

(2) 次の superstep に必要なデータをプロセッサ間で通信する段階.

(3) 各 superstep を待ち合わせるバリア同期の段階.

また，superstep は大域的な境界で区切られている．superstep の概念を図 2 に示す．BSP モデルでは，実際に計算を行うプロセッサを worker と呼び，worker の動作を指示するプロセッサを master と呼ぶ．BSP モデルの特徴として，実際の worker 間の通信は master を中継して行われるが，プログラムの記述時には master を中継することは明記する必要がないことが挙げられる．この worker 間の通信部分は，図 2 の計算部分とバリア同期の間で行われる通信部分にあたる．BSP モデルでは，superstep 内で計算に使われた値などを，任意の時刻に，任意の worker 同士で自ら通信することはできない．

BSP モデルでは，プログラムで記述された superstep の順序通りに，p 個のプロセッサ（もしくはスレッド）が並列計算を行う構造を持つ．図 2 では合計 4 つのスレッドで superstep を実行している．

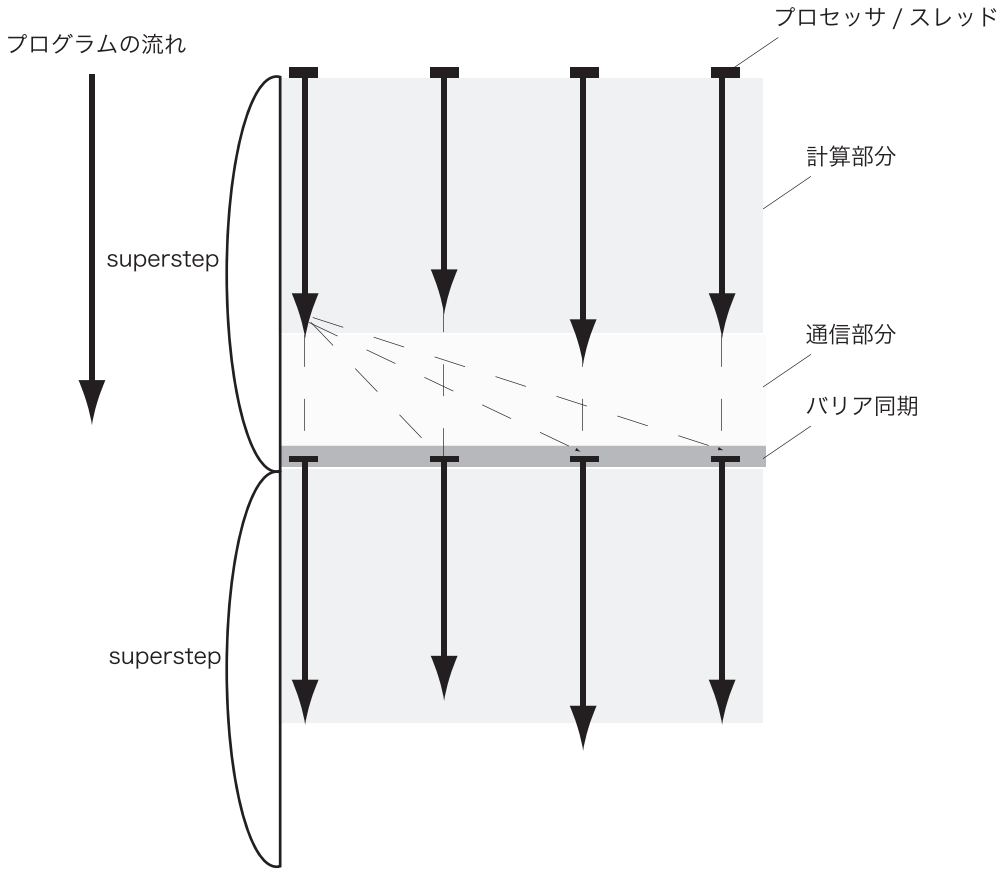

図 2: BSP モデルと superstep

次に，BSP モデルにおける superstep での実行時間とプログラムの総実行時間の関係を次に表す．ここで，

- バリア同期（オーバーヘッド）：L
- 通信にかかるデータ転送率：g
- 各プロセッサのプログラム最大実行時間：w
- 各プロセッサの最大通信量：h

とする．

各 superstep の実行時間 t(step) には次の関係がある．

$$t(step) = w + hg + L$$

superstep の実行時間は，計算部分，通信部分，バリア同期にかかる時間の総和で表される．まず，superstep の計算部分にかかる時間は，各プロセッサで最も計算に時間がかかったスレッドの実行時間が適応されるため，図 2 の例では左から 3 番目のプロ

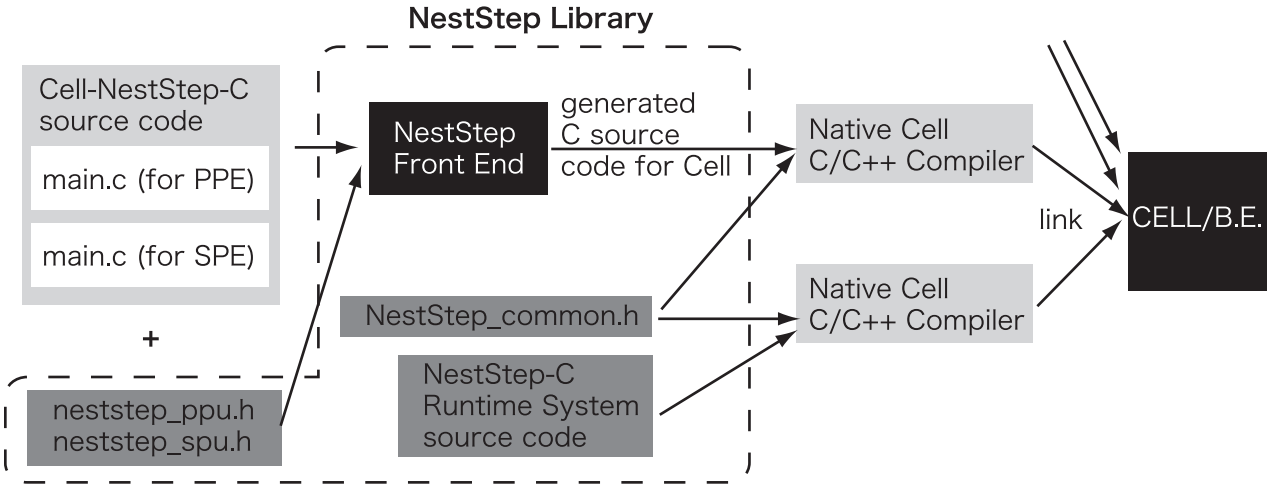

図 3: Cell-NestStep-C ライブラリの概念図

セッサでのプログラム実行時間が w となる．次に，通信部分にかかる時間は，各プロセッサの最大通信量 h と通信にかかるデータ転送率 g の積で表される．そして，バリア同期の部分にかかる時間は，バリア同期のオーバーヘッド L である．

また，BSP モデルのプログラム全体の実行時間 t(prog) は，上で示した各 superstep の実行時間 t(step) を全て足し合わせたものである．よって，次のように表すことができる．

$$t(prog) = \sum_{step} t(step)$$

2.2.2 Cell-NestStep-C

NestStep[3] は，前節で述べた並列計算用 BSP モデルを採用した並列プログラミング言語であり，Christoph W. Kessler によって提唱された（1998 - 2000）．1998 年に Java を用いて拡張された NestStep-Java，2000 年に C 言語によって拡張された NestStep-C がある．NestStep-C は 2006 年に改訂され，MPI を用いたクラスタ上で動作が可能になった Cluster-NestStep-C も存在する．そして，2007 年に拡張された Cell-NestStep-C により，Cell/B.E. 上での動作が可能になった．NestStep は既存の BSP モデルを拡張することで，階層的なプロセッサ組織概念と superstep の入れ子による動的多層並列処理をサポートしている．ここで，Cell-NestStep-C ライブラリの概念図を図 3 に示す．Cell-NestStep-C を用いて記述されたプログラム（C もしくは C++で記述可能）を用意し，NestStep ライブラリをリンクさせて Cell/B.E. の汎用コンパイラでコンパイルすることで，Cell/B.E. 上で動作が可能になる．

さらに NestStep は，BSP モデルにはなかった，ソフトウェアによる仮想共有メモ

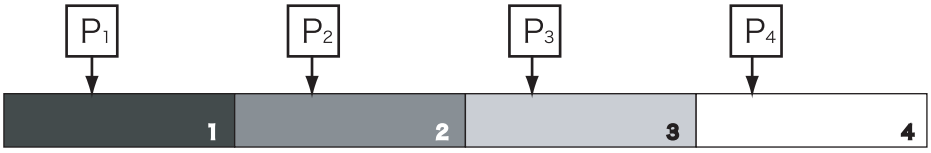

図 4: Block Distributed Array

リを実現しており，分散したメモリ領域ではなく一つの大きなメモリ領域とみなすことによって，superstep の同期を実現している．NestStep のサブセットプロトタイプは Java のようなシーケンシャルベース言語をベースに実装されており，Java や C のような命令型プログラミング言語を拡張させることで設計された．

NestStep は、オブジェクト言語構造や run-time サポートにより、プログラム実行の明確な制御とオブジェクトの共有化を可能にしている．

一方で NestStep では，共有およびプライベートな変数と配列が用意されている．プライベートな変数と配列は，一般的な C 言語のデータ型のように扱うことができる．また，各変数および配列を扱うプロセッサ側からのみアクセスが可能である．共有変数は，各 superstep ごとに同期を行う．

さらに、NestStep は、block distributed array と cyclic distributed array という 2 種類の分散データ型をサポートしている．block distributed array は，配列をプロセッサ数に分割し，その分割したブロックを各プロセッサに割り当てる．図 4 は，要素 16 個の配列を 4 つのプロセッサに均等に割り当てる block distributed array の例である．この例では要素 0〜3 はプロセッサ P1，要素 4〜7 はプロセッサ P2 に割り当てられる．cyclic distributed array は，配列をプロセッサ数で分割した block distributed array のブロックをさらに小さなブロックで仕切り，その仕切られたブロックを周期的にプロセッサに割り当てられる．図 5 では，要素 0，4，8，12 はプロセッサ P1，要素 1，5，9，13 はプロセッサ P2 に割り当てている例である．記述するプログラムの特性に合わせて，2 種類の array を使い分けることで，プログラムの動作がより効率的になる．ここで，特に block distributed array に注目しておく．後述で述べるスケルトンライブラリ BlockLib は，block distributed array の特性を上手く活かして設計されたライブラリである．

2.3 BlockLib

BlockLib [4] は，Cell/B.E. プログラミングの際に必要となるメモリマネジメント，SIMD 最適化手法，並列プログラミングをより簡素に記述できるようなスケルトンラ

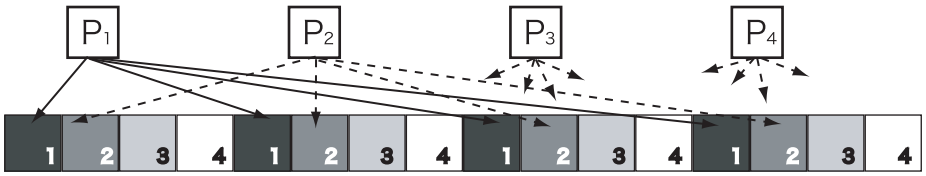

図 5: Cyclic Distributed Array

イブラリである．BlockLib は数個の計算パターンのスケルトンで実装されており，その内 map と reduce は現在でもよく用いられている手法である．BlockLib の実装の一つは，map と reduce の組み合わせによる，性能の最適化である．もう一つは，一つの要素とその近傍の要素を同時に計算可能な map であり，並列スケルトンは C 言語ライブラリと同様な方法で実装されている．以下に，map，reduce，双方を組み合わせた map-reduce の概念について簡単に述べる．

map とは，ある大きなデータに対して，ひとまとまりのデータから新しい有用なデータへと変換するプロセスである．map では様々なデータを受け取り，後述で述べる reduce で処理できる形式に変換する．map の動作を論理式を用いて表現したものを以下に示す．

$$\forall i \in [0, N-1], r[i] = f(a_0[i], ..., a_k[i])$$

redece とは，前述で述べた map から得られた複数のデータを加工し，最終的に得たいデータとして出力を行うプロセスである．reduce の動作を論理式を用いて表現したものを以下に示す．

$$r = a[0]\ op\ a[1]\ op\ .....\ op\ a[N-1]$$

map-reduce とは，前述で述べた map と reduce を組み合わせた手法である．主に，分散環境上でデータの抽出や統計などに良く使われており，Google の検索システムで使われていることでも有名である．これは，元になるデータが大規模になった場合や，サーバが数千台になった場合でも，リニアに分散可能な非常に優れた方式である．map-reduce の概念図を図 6 に示す．また，map-reduce の動作を論理式を用いて表現したものを以下に示す．

$$f(a_0[1], ..., a_k[1])\ op\ f(a_0[2], ..., a_k[2])\ op\ ...\ op\ f(a_0[N-1], ..., a_k[N-1])$$

BlockLib の機能は，前述で述べた NestStep の superstep 1 つ 1 つの塊に作用する．また、map の機能は複数の superstep をまとめた単位でも作用する．

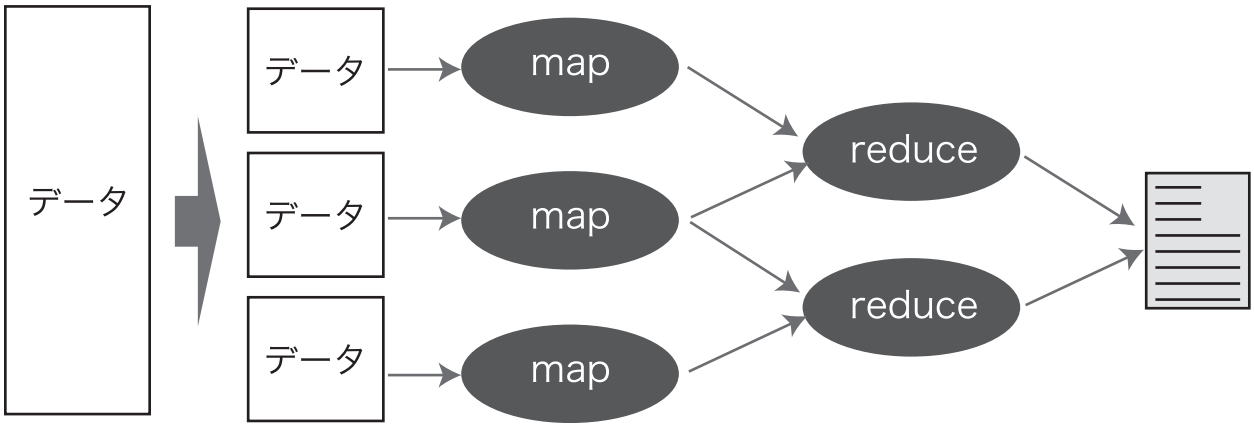

図 6: map-reduce の概念図

2.4 問題点

2.4.1 libspe

Cell/B.E. 上では、アプリケーションは libspe (SPE Runtime Management Library) と呼ばれる標準ライブラリを利用して，SPE を使用する．

libspe には，libspe1 系と libspe2 系の 2 種類が存在する．libspe1 から libspe2 へのアップグレードでは，PPE から SPE プロセスを管理する方法が全体的に見直された．このため，2 つのライブラリを区別する時には libspe の後ろに 1 または 2 を付けて区別して表現する．

2009 年 1 月現在での最新版 Cell SDK 3.1 では，libspe 2 バージョンのライブラリが IBM から提供されている．libspe1 が同梱されていたバージョンの Cell SDK は，現在入手は不可能となっている．なお libspe2 は，Cell SDK 2.1 から libspe1 に代わり正式に採用されている．しかし，現在でも libspe1 向けのプログラムは少なくないため，libspe1.2 （libspe1 の最終版）は現在でも入手が可能である．

libspe 2 系のライブラリはこれまでの libspe 1 系の API を 1 から再設計し直したもので、libspe 1.x に比べるとより低レベル (OS による抽象化に近いレベル) から高レベルまで、幅広いプログラミング要求をカバーするような API 設計となっている．現在配布されている libspe 2.1 では、そのうちもっとも基本的なレベル (システムコールに近いレベル) である base 層とその上の event 層の API が利用可能となっている．libspe2 の base API は、libspe1 で提供されていた API よりやや細粒度の API となっている．なお，現在公開されている libspe 2 ライブラリでは，libspe1 互換の API は提供されておらず，libspe1 で書かれたプログラムの場合，libspe2 形式に書き換える必要がある．

現在配布されている NestStep ライブラリは，Cell SDK 2.1 以前のバージョンで用いられていた Libspe1 に準拠した書き方となっている. そのため，現在最新の Cell SDK 3.1 で動作可能となるよう，libspe1 プログラムから，libspe2 プログラムに書き換える必要がある.

また，Cell/B.E. 上で動作するプログラムをコンパイルする際には，Cell/B.E. 用のコンパイラ（ppu-gcc, spu-gcc）が必要である．使用する ppu-gcc のバージョンによっては，デフォルトで生成されるプログラムが 32bit プログラムになる場合と，64bit プログラムになる場合があり，注意が必要である．現在，使用するコンパイラによって，配列を動的に確保した場合に，アライメント境界が意図したバイト数にならず，NestStep ライブラリが意図した通りに動作しない問題がある.

3 提案

本章では NestStep の libspe1 から libspe2 への仕様変更に伴うプログラムの流れ，および制約を記述し，NestStep が libspe2 でも使用可能であることを提案する.

3.1 libspe1 から libspe2 への変更点

3.1.1 コンテキスト生成

libspe1 では，SPE のコンテキストの作成と実行は，`spe_create_thread()` という関数で提供されていた．以下に，libspe1 で SPE を用いるときの基本的なフローを示す.

1. `spe_open_image()`：SPE プログラムファイルをオープン
2. `spe_create_thread()`：SPE スレッドを生成 (同時に SPE スレッドが実行を開始)
3. `spe_wait()`：SPE スレッドの終了待ち (同時に SPE スレッド資源の破棄)
4. `spe_close_image()`：SPE プログラムファイルをクローズ

libspe1 では，SPE スレッドを生成すると必ず対になる PPE スレッド (pthread) が内部で生成される仕様であった．そのため，SPE スレッドの生成 API (`spe_create_thread()`) はすぐに終了し，呼び出したスレッドは続けて他の処理を行うことができる．したがって，複数個の SPE を使用する場合においては，単純に `spe_create_thread()` を複数回呼び出すことで実現可能であった.

一方 libspe2 では，SPE コンテキストの作成と，SPE コンテキストの実行を関数 spe_context_create()，spe_program_load()，spe_context_run() で行う仕様に変更となった．この時，spe_program_load() で LS にロードされた SPE プログラムを SPE 実行イメージという．また，複数の SPE を使う場合，pthread などの PPE 上のスレッドライブラリと組み合わせてプログラムの制御を行う必要がある．以下に，libspe2 で SPE を使うときの基本的なフローを示す．

1. spe_image_open()：SPE プログラムファイルをオープン
2. spe_context_create()：SPE コンテキストを作成
3. spe_program_load()：SPE 実行イメージを SPE コンテキストの LS にロード
4. spe_context_run()：SPE コンテキストを実行
5. spe_context_destroy()：SPE コンテキストの破棄
6. spe_image_close()：SPE プログラムファイルをクローズ

この時，SPE コンテキストを実行する spe_context_run() はブロッキングな API であることに注意する必要がある．spe_context_run() を呼び出すと，SPE プログラムが正常終了あるいは異常終了または，(stop シグナルなどで)PPE 側に処理を戻す処理が呼び出されたりしない限り，呼び出し側は先に進まない．

この libspe1 と libspe2 の両者のフローを比較したものが，図 7 である．

今回使用した NestStep の評価において，libspe1 から libspe2 の変更による影響は，SPE のスレッド管理に関する部分であり，API に自動的にスレッド化を処理させていたものが，プログラマ自身の手によってプロセスの各段階を制御可能となったことである．

3.1.2 複数個の SPE を使用した場合

N 個の SPE を使う場合の典型的なフローを，具体的に図 7 のプログラムフローを用いて説明する．libspe1 の場合，使用する SPE の数だけ spe_create_thread() を繰り返す．一方で，libspe2 の場合は，使用する SPE の数だけ PPE スレッド（pthread など）を生成し，実行後は pthread_join() でスレッドの終了を待ち合わせる．なお libspe1 と libspe2 との違いは，前述した通りコンテキスト処理の変更であり，スレッド管理の柔軟性向上を目指した仕様上の取り組みである．

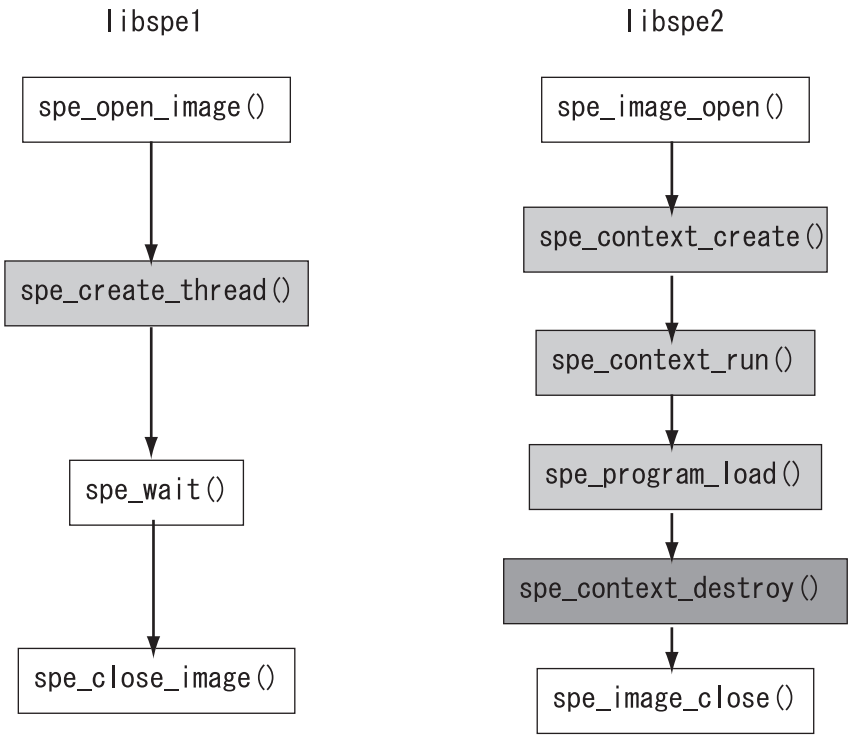

図 7: プログラムフロー

3.2 アライメント

Cell/B.E. において DMA 転送を行う場合に，以下の制約が存在する．

1. DMA 転送サイズ
   データサイズは基本的に 16Byte の倍数であり，データサイズを最大 16KByte 以下にしなければならない．
   （16Byte 未満のデータサイズ（1、2、4、8Byte）の DMA 転送もサポートはしている．）
2. DMA 転送アドレスのアラインメント
   - 16Byte 以上の DMA 転送の場合
     転送元と転送先のアドレスがそれぞれ 16Byte 境界に整列しなければならない．
     （最大の実行速度を実現できるのは 128Byte に整列しているときである．）
   - 16Byte 以下の DMA 転送の場合

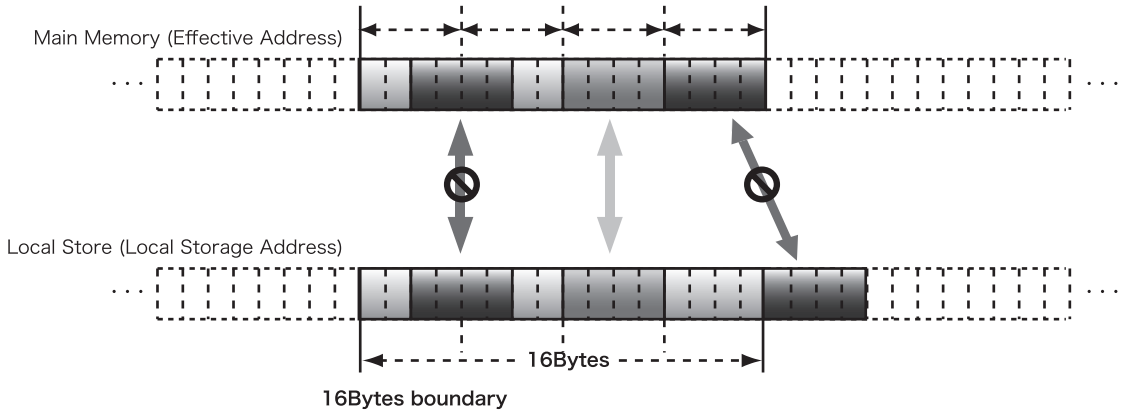

図 8: DMA 転送

転送元と転送先のデータ領域のアドレスについては、以下の条件を満たす必要がある．

(a) 転送サイズに応じた自然なアライメントである（転送サイズの Byte 境界に整列している）こと．

(b) 実効アドレスおよび LS アドレスの下位 4 ビット（16Byte 境界からの相対位置）が同一であること．

図 8 に示すように，4Byte のデータを DMA 転送する場合は，実効アドレスと LS アドレスがそれぞれ，4Byte 境界に揃えられ，かつ実効アドレスと LS アドレスの 16Byte 境界からの相対位置が同じである必要がある．

これらの条件が満たされていない場合，MFC の処理は一時停止し、DMA アライメント・エラーの割り込みが発生する．その結果，PPE プログラム，SPE プログラム双方が異常終了する場合があるため，この 2 つの制約を満たしているか否かを判断する必要がある．

NestStep は，制約条件を満たしているかどうかを判断する `check_dma()` という関数を実装している．しかしながら，使用するコンパイラによって，デフォルトでは 16Byte 境界にアライメントされないため，プログラムが意図した動作をしない．このような場合には，変数宣言時もしくは型宣言時に明示的にアライメントする Byte 数を指定する必要がある．もしくは，変数の宣言時に 16Byte 長になるように変数宣言に手を加えるという対処療法的な手段も存在する．

4　実装

本章では，libspe のライブラリ変更に伴い，NestStep ライブラリに変更を加えた際の手順及び，変更後にプログラムが正しく実行されない事への対処法について述べる．

4.1　libspe の変更に伴う書き換え

libspe1 から libspe2 への書き換えは，IBM のマイグレーションガイド [5] を参考に行った．変更箇所は，全て ppu から spu の動作を管理する部分に限られる．書き換え関数には，1 対 1 の場合，1 対 1 でない場合，関数が廃止されている場合の計 3 パターンが存在する．

4.1.1　1 対 1 対応の場合

関数が 1 対 1 対応である場合では，書き換えはさほど難しくはない．しかし，関数によっては同じ関数名でも，引数の順番，引数の数などが変化したものや，関数名が若干変更しているものがあるので，注意が必要である．

1 対 1 対応の例に該当する関数として，`spe_open_image`，`spe_close_image` の具体的な記述例をリスト 1 に示す．例では，関数は 1 対 1 対応をしており，引数も変化しないが，関数名が若干変更となっており，特に人の目で見た場合には，関数の文字列の順序が逆転していることに気付きにくい．コンパイラによって，この変更点を見つけ出すと共に，注意深く見定める必要がある．

4.1.2　1 対 1 対応でない場合

関数が 1 対 1 対応でない場合，すなわち 1 対多対応では，書き換えは若干難しくなる．それは libspe2 への変更に伴い，SPU スレッドの管理がプログラマの手によって厳密に行わなければならないという状況に起因する．すなわち，プロセスの各段階を直接制御できるようになったためである．IBM はこれによって記述が複雑になる場合には，プログラムにとって最も有効に機能するシーケンスをヘルパー関数にラップすることを推奨している．1 対 1 対応でない例に該当する関数として，`spe_create_thread` の具体的な記述例ををリスト 2-1，2-2 に示す．

例：spe_open_image，spe_close_image

libspe1 ppu example

```
#include <libspe.h>
...
spe_program_handle_t *
  <program_handle>;
...
<program_handle> =
  spe_open_image("<filename>");
...
spe_close_image(
  <program_handle>);
```

libspe2 ppu example

```
#include <libspe2.h>
...
spe_program_handle_t *
  <program_handle>;
...
<program_handle> =
  spe_image_open("<filename>");
...
spe_image_close(
  <program_handle>);
```

リスト１：1 対 1 対応の場合

libspe1 ではスレッド化を自動的に行っていたものを，libspe2 では pthread を用いることでスレッド化を行わせているため，必要な手続きが多くなってしまっている．そのため，前述したように最も有効なシーケンス，すなわち ppu_pthread_function という関数と，これを利用するための ppu_pthread_data_t という構造体を作ることにより，煩雑な手続きを 1 度で終わらせる工夫をとる必要がある．

4.1.3 関数が廃止されている場合

関数が廃止されている場合，これも書き換えが若干難しくなる．これは，代替手段がガイドで明示されている場合には，それをベースにしてコーディングを行うことは可能であるが，代替手段が明示されていない場合はプログラマのスキルによって，難易度が変化するからである．

例：spe_create_thread

libspe1 ppu example

```
#include <libspe.h>
...
spe_gid_t <group>;
spe_program_handle_t
  <spe_program>;
void *<argp>;
void *<envp>;
unsigned long <mask>;
int <flags>;
speid_t <speid>;
...
<speid> =
  spe_create_thread(<group>,
    &<spe_program>, <argp>,
    <envp>, <mask>, <flags>);
```

libspe2 ppu example 1/2

```
#include <libspe2.h>
#include <pthread.h>
...
typedef struct ppu_pthread_data {
  spe_context_ptr_t <speid>;
  pthread_t pthread;
  unsigned int entry;
  unsigned int <flags>;
  void *<argp>;
  void *<envp>;
  spe_stop_info_t stopinfo;
} ppu_pthread_data_t;
...
spe_program_handle_t
  <spe_program>;
void *<argp>;
void *<envp>;
int <flags>;
pthread_attr_t attr;
ppu_pthread_data_t ppdata;
...
```

リスト 2-1：1 対 1 対応でない場合

関数が廃止されている場合の例として spe_create_group を例に挙げ，この関数に対応する記述方法の一例をリスト 3-1，3-2 に示す．この例では，スレッドの優先度を pthread を利用して実現する方法を記述している．このように，代替案が明示されている場合には，それを活用することでプログラムが実行可能となる．

libspe2 ppu example 2/2

```
...
void *ppu_pthread_function(void *arg) {
  ppu_pthread_data_t *datap = (ppu_pthread_data_t *)arg;
  int rc;
  do {
    rc = spe_context_run(datap-><speid>, &datap->entry,
                         datap-><flags>, datap-><argp>,
                         datap-><envp>, &datap->stopinfo);
  } while (rc > 0);
  pthread_exit(NULL);
}
...
ppdata.<speid> = spe_context_create(<flags>, NULL);
...
spe_program_load(ppdata.<speid>, &<spe_program>);
...
ppdata.entry = SPE_DEFAULT_ENTRY;
ppdata.flags = <flags>;
ppdata.argp = <argp>;
ppdata.envp = <envp>;
pthread_create(&ppdata.pthread, &attr,
               &ppu_pthread_function, &ppdata);
```

リスト 2-2：1 対 1 対応でない場合

一方で前述したように，明示されていない場合 (例えば，spe_group_max など) には自身でその機能をコーディングする必要があり，プログラマに一定以上のスキルを必要とする．

例：spe_create_group

libspe1 ppu example

```
#include <libspe.h>
...
spe_gid_t <group>;
int <policy>;
int <priority>;
int <spe_event>;
...
<group> = spe_create_group(<policy>, <priority>, <spe_event>);
```

リスト 3-1：関数が廃止されている場合

4.2 コンパイラに依存しない変数宣言の記述

4.2.1 aligned 属性

整列したメモリを静的に確保する場合，変数を確保する時，もしくは型を宣言する時に，__attribute__((aligned(n))) を用いて aligned 属性をつけて領域を確保する．この時，aligned 属性の制限事項がある．PowerPC アーキテクチャではスタックポインタは 16 の倍数になることが保証されている．従って，コンパイラでは自動変数において 16 バイトまでは整列可能だが，それ以上を整列させることは保証されていない．また，16 バイト以上を指定しても，コンパイラはエラーや警告を出さない．

次に構造体のパディングの場合，ターゲットとなる CPU のアーキテクチャによって異なる．例えば，x86 の場合は 4byte 境界にパディングされるが，Cell/B.E. ではデータが 8byte 境界にアライメントされる．

ppu-gcc はバージョン 4.0.x ではデフォルトで 64bit コードを生成するが，バージョン 4.1.x ではデフォルトで 32bit コードを生成する．long 変数のデータ長に依存したプログラムを書く場合は，コンパイラオプションを用いて明示的に，生成するコードを指定するなどの注意が必要である．次に，動的にメモリを確保する場合について述べる．

libspe2 ppu example

```
#include <libspe2.h>
#include <pthread.h>
...
int <policy>;
int <priority>;
int <spe_event>;
pthread_attr_t attr;
struct sched_param param;
spe_context_ptr_t <speid>;
...
pthread_attr_init(&attr);
pthread_attr_setschedpolicy(&attr, <policy>);
param.sched_priority  = <priority>;
pthread_attr_setschedparam(&attr, &param);
...
<speid> = spe_context_create(
          <spe_event> != 0 ? SPE_EVENTS_ENABLE : 0, NULL);
```

リスト 3-2：関数が廃止されている場合

4.2.2 動的メモリ確保

動的にメモリを確保する場合では，コンパイラにより，8byte でアライメントされてしまう場合があり，この場合の動作は保証されないため，静的にメモリを確保するように変更するか，リスト 4 のように記述をすることで回避した．

これはコンパイラが，その変数の確保に必要なサイズを自動的に把握し，最適な位置から必要サイズ数を確保する．この仕様を利用し，そもそも確保するサイズを 16byte 長にすることで，コンパイラが必ず 16 バイト境界で整列させるよう実装したものがリ

スト 4 で示したコードである．

malloc 関数使用時

```
pi = (double *)
       malloc(sizeof(double)*2);
```

memalign 関数使用時

```
pi = (double *)
       memalign(16,
         sizeof(double)*2);
```

リスト 4：動的メモリ確保記述例

しかしながら，この手法には 1 つ問題点がある．それは，多くの変数にこの手法を多用してしまうと，膨大なメモリを必要とするようなプログラムであった場合に，256KByte である SPU の LS のメモリ領域を有効活用できない．この解決にはメモリ管理が重要である．

5 評価

libspe1 で記述されている NestStep ライブラリを libspe2 に移植し，修正を加えた NestStep ライブラリを使用して，BlockLib を動作させてプログラム実効速度の比較を行った．評価には，BlockLib に同梱されているテストプログラムを用いた．

5.1 予備評価

まず予備評価として，社団法人情報処理学会等が主催している Cell Challenge 2009 のサンプルプログラムを用いて，libspe1 と libspe2 間でプログラムの実行速度が同等かどうかを評価した．評価環境には SCE 製 PLAYSTATION3(以下，PS3) を用いた．結果を表 1 にまとめる．

表 1: 予備評価 実行結果

	PLAYSTATION3	
	libspe1	libspe2
平均	3.4062	3.4040

上記の結果から，libspe1 と libspe2 ではライブラリの仕様変更における実行速度の変化は見られない，もしくは限りなく小さいことがいえる．この結果をもとに，本評価を実施した．

5.2 評価環境および評価プログラム

5.2.1 評価環境

評価環境には PS3，東芝の Cell Reference Set (以下 CRS)[6] を用いた．各評価環境を表 1 に示す．

CRS は，Cell/B.E. とそのチップセット，I/O インターフェースを実装したハードウェア・プラットフォームである．CRS に搭載されている Cell/B.E. では，7 基の SPE が動作し，PS3 に搭載されている Cell/B.E. では，6 基の SPE が動作する．

NestStep ライブラリ (PPU 用，SPU 用) ならびに，使用した全てのテストプログラムを ppu プログラム用コンパイラ ppu-gcc(バージョン 4.1.1），SPU 用コンパイラ spu-gcc（バージョン 4.1.1），埋め込み SPU 用コンパイラ ppu-embedspu を用いてコンパイルした．CRS での制限事項として，今回使用する ppu プログラムは全て 32 ビットモード（-m32）でコンパイルした．また今回は，CRS では最大 7 基まで SPE を同時に実行可能であるが，PS3 では同時に実行可能な SPE が 6 基である．従って，テストプログラムを実行する場合に使用する SPE の数は 6 基に統一した．

表 2: 評価環境

	PS3	CRS
OS	FedoraCore9	Red Hat Linux 4.1
CPU	Cell/B.E. 3.2GHz	Cell/B.E. 3.2GHz Reference Set
コンパイラ	gcc 4.1.1	gcc 4.1.1
最適化オプション	O2	O2

評価プログラムには，map，reduce，map_reduce をそれぞれ実装した，NestStep ライブラリ及び BlockLib を使用しないプログラム（以下，original），BlockLib を使用し libspe1 に準拠した NestStep ライブラリを実行時にリンクさせたプログラム（以下，libspe1），および BlockLib を使用し libspe2 に準拠した NestStep ライブラリを実行時にリンクさせたプログラム（以下，libspe2）で評価した．評価するプログラムをこの

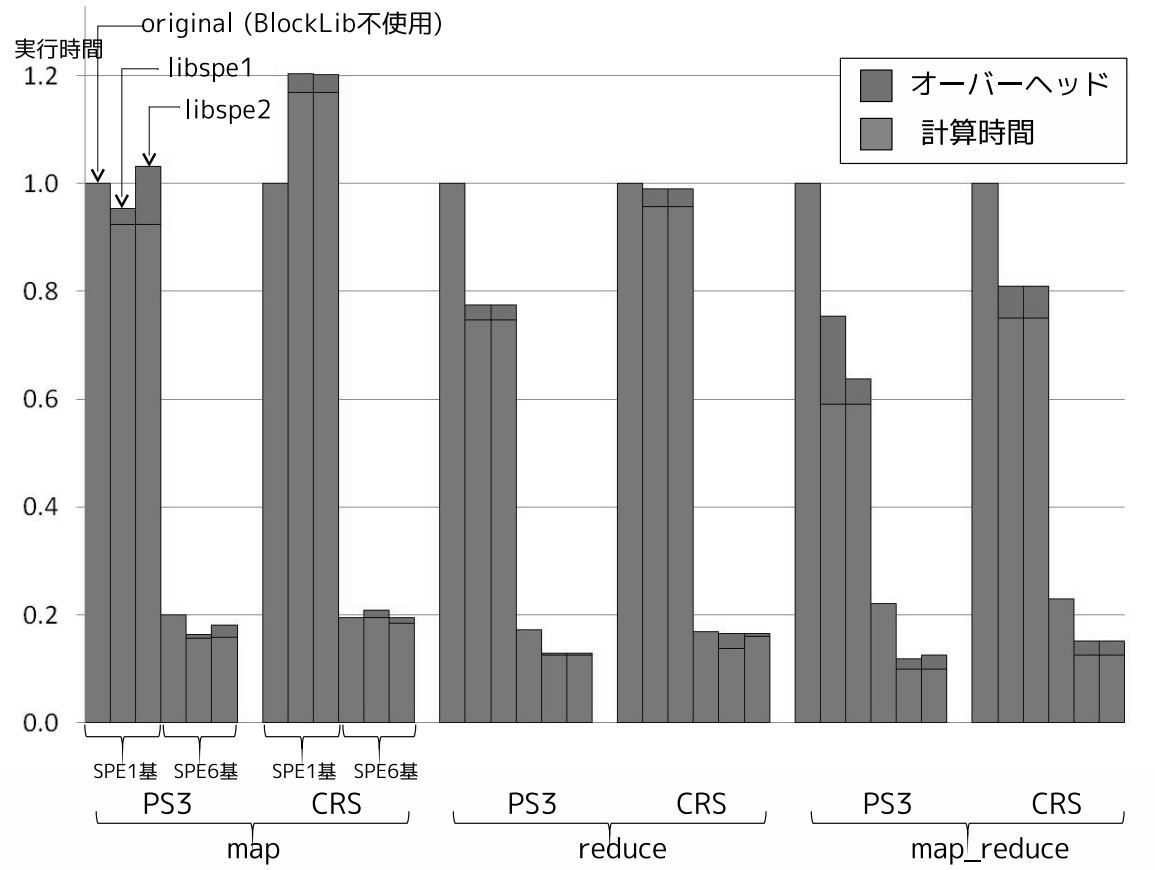

図 9: 実行速度の比較

ように選定した理由は，BlockLib の性能の評価と libspe1 と libspe2 との仕様変更における実行速度変化を再確認するためである．

さらに，計算に使用するデータの割り当ては，SPE を 6 基使用した場合，データサイズを 6 等分し，各 SPE に均等にデータを分配している．一方，SPE1 基で逐次実行した場合は，128Byte にデータをアライメントし，16KByte 単位で順にデータを DMA 転送した．

5.3 評価結果

評価結果を，図 9 にまとめる．3 つのテストプログラム map，reduce，map_reduce を PS3 と CRS で実行させ，6 種類のプログラムの実行時間を評価した．6 つのプログラムは左から，original(1 SPE)，libspe1(1 SPE)，libspe2(1 SPE) ,original(6 SPE)，libspe1(6 SPE)，libspe2(6 SPE) を実行させたもので，それぞれ各実行環境で実行した original(1 SPE) で正規化した．

5.4 考察

結果から以下の 2 点がいえる．まず第一に BlockLib を用いた libspe1 と libspe2 とでは，ほぼ実行速度に有意差はなかった．これは，libspe1 から libspe2 への仕様変更は，本実験でのプログラムの実行速度に影響を与えない程度の変更であることが確認できた．

次に，BlockLib を用いてプログラムを記述することにより，original に近いプログラムの実行速度を得ることができた．プログラムに施した SIMD 命令などの最適化の手法によっては，BlockLib を用いた場合の方が実行速度が速い可能性もある．逆に，計算部分だけに注目すると，original より BlockLib を用いた方が速度が速いが，BlockLib を用いた事によるオーバーヘッドの時間があるため，プログラム全体での速度は original に劣る．しかし，CELL/B.E. のプログラムに不慣れな場合であっても，すべてのプログラマに同じ速度を保証できるという点では，BlockLib の有用性は高い．

今回の場合，使用するデータを均等に SPE の数で分割し割り振ったため，各プログラムの 1 SPE の場合と 6 SPE の場合を比べると，実行時間は約 6 分の 1 に減っている．この結果から，SPE1 基で逐次実行するより，同じ計算を複数の SPE で並列に実行させることで，プログラムの実行速度は大幅に向上したことが分かる．

6 おわりに

本研究では，Cell/B.E. 向けのメモリマネジメントと SIMD 最適化を支援するスケルトンライブラリ BlockLib を，最新の Cell SDK 3.1 で標準利用となる Libspe2 へ対応できるよう移植を行った．移植を行った NestStep ライブラリ及び，BlockLib に同梱されているテストプログラムを用いて，両ライブラリが正常に動作することを確認した．動作確認および，評価には PLAYSTATION3，Cell Reference Set を用いた．結果から，BlockLib を用いてプログラムを記述すると，ライブラリを使用したことによるオーバーヘッドの時間が発生するが，自分の手で記述し最適化を施したプログラムに近い実行速度を出すことができた．CELL/B.E. のプログラムに不慣れな場合であっても，すべてのプログラマに同じ速度を保証できるという点では，BlockLib の有効性は高く，汎用性があるのではないかと期待される．今後の課題としては，次の 2 つの課題が考えられる．

第 1 の課題は，残る全ての NestStep ライブラリ関数を，libspe2 に移行することである．本研究で使用した NestStep ライブラリ関数は，BlockLib であらかじめ用意されているテストプログラムのみを動作対象としたため，テストプログラムで用いてい

ない関数は Libspe2 には対応していない．今後は 全ての NestStep ライブラリ関数の修正を行いたい．

第 2 の課題は，BlockLib を，Cyclic Distributed Array が使用できるよう拡張することである．BlockLib では，NestStep ライブラリで提供されている，Block Distributed Array を利用して動作を行っている．NestStep ライブラリでは，前述以外に Cyclic Distributed Array という異なる構造を持った配列が用意されている．現段階では，Cyclic Distributed Array を使用するライブラリ関数は用意されていない．今後は，BlockLib に，Cyclic Distributed Array を使用するライブラリ関数を新たに提供することで，さらなる拡張を行いたい．

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松尾啓志教授, 津邑公暁准教授, 齋藤彰一准教授, 松井俊浩助教に深く感謝致します. 特に，神谷優志氏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 ・ 津邑研究室の皆様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] Valiant, L.: A bridging model for parallel computation, *Communication of the ACM*, Vol. 33, pp. 103–111.

[3] Keβler, C. W.: NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model (1999).

[4] Alind, M.: A Skeleton Library for Cell Broadband Engine, Master theses, IDA Linkötopings universitet (2008).

[5] : *SPE Runtime Management Library Version 1 to Version 2 Migration guide*, IBM (2007).

[6] 上村剛, 大溝孝, 粟津浩一: Cell リファレンスセット概要, 東芝レビュー, Vol. 61, No. 6, pp. 25–29 (2006).