

マルチコア・メニーコア時代に向けた ランタイムシステムの設計と Cell/B.E. による実装

太田 篤志[†] 並木 美太郎[‡]

Cell Broadband Engine (Cell/B.E.) は、汎用的なプロセッサコア PPE と演算用のプロセッサコア SPE とで構成されたヘテロジニアスマルチコア CPU である。大量のデータ処理に特化したアーキテクチャである SPE をいかに効率よく使うかがポイントとなるが、Cell/B.E. 上で動作する Linux に用意されている既存環境 (spufs, libspe2, MARS) ではアプリケーションが SPE を細かく制御しなければならなかったり、SPE ヘタスクを割り当てる際に無駄なオーバーヘッドが存在している。本研究では SPE のタスクを一元管理するプロセスマネージャと、SPE 上に実装した軽量カーネルが連携して動作することで、アプリケーションに対して SPE を効率的に割り当て、タスク割り当てにかかる時間を短縮させる。評価の結果、空のタスクを SPE で動作させたときの実行時間は最大で 4.32 倍、クイックソートの実行時間は最大 6.43 倍の高速化を実現することができた。また本論文では来るべきマルチコア、メニーコアの時代に向けたランタイムシステムについても言及する。

Design of a Runtime System for Multicore/Manycore Architecture and its Implementation on Cell/B.E.

ATSUSHI OHTA[†] and MITARO NAMIKI[‡]

The Cell Broadband Engine (Cell/B.E.) is the heterogeneous multicore processor including PPE(s) for general-purpose processings and SPE(s) for computation. It is important for processing a great deal of data to program SPEs effectively. However, operations imposed on applications to use SPEs is main problem to reduce overhead allocating tasks to SPEs in the existing environments (spufs with libspe2, and MARS) of the Linux running on a Cell/B.E. This paper describes design of lightweight kernel for multicore/manycore processor that application can use cores more effectively and faster by collaboration with the process manager which unifies tasks to execute on them. The kernel is implemented on Cell/B.E and evaluated. The execution speed of empty processes is up to 4.32 times faster than the existing environments, and the one of quicksort is up to 6.43 times faster, as results of the evaluation.

1. はじめに

近年の CPU はプロセッサ単体の複雑化と高コスト化から性能向上が頭打ちになっていることを受け、1 プロセッサに複数のコアを搭載するマルチコア構成が主流となっている。しかしシングルコアのアーキテクチャをベースにしたホモジニアスなマルチコアでは、この問題は根本的に解決していない。そこで注目を集めているのが、ヘテロジニアスマルチコアやメニーコアといったプロセッサ構成である。ヘ

テロジニアスマルチコアは単体で高速化された従来の CPU コア 1 個～数個と、構造を単純化した——あるいは特定処理に特化した——多数の CPU コアを同居させることによって、シングルスレッド、マルチスレッド両方の性能を発揮しつつ、かつ同程度の性能のホモジニアスマルチコアと比べて発熱量や消費電力をも削減することができるというものである。またメニーコアは従来の CPU コア自体の構造を単純化し、これを多数 (10 個以上^{[1])} 集積することで性能と電力効率を向上させている。コア間でアーキテクチャが変わらないため、ヘテロジニアスマルチコアとは違ってどちらのコアでタスクを実行させればよいかを考える必要がないというメリットもある。さらにはここ数年で高集積化、性能向上の著し

[†] 東京農工大学大学院工学府
Graduate School of Engineering, Tokyo University of Agriculture
and Technology.

[‡] 東京農工大学大学院共生科学技術研究院
Institute of Symbiotic Science and Technology, The Graduate
School at Tokyo University of Agriculture and Technology.

い GPU を、グラフィック処理だけでなく他の演算処理にも利用しようという GPGPU (General-Purpose computing on GPU) という技術も登場[2]し、マルチコア、メニーコアを取り巻く環境はより複雑になってきている。

本論文では、来るべきマルチコア、メニーコアのアーキテクチャに対するランタイムシステムについて言及したのち、ヘテロジニアスマルチコアの 1 つである「Cell Broadband Engine」へのこのランタイムシステムの実装についてを述べる。

2. マルチコア・メニーコア時代に向けたランタイムシステム

マルチコアの潮流は、ホモジニアスやヘテロジニアスという CPU 構成の差こそあれ、コア数を増やすことによって高いパフォーマンスを得る方向へと向かっている。デュアルコア、クアッドコアの CPU は現在では当たり前のように PC に用いられ、GPU においては既に 100 を超える数のコアを搭載するものも登場している[3]。増加の一途を辿る CPU コアを簡易に利用できるようにするための管理機構を考えたときに、すべてのコアを単一のソフトウェアで管理する方式ではいずれ限界を迎えることになるだろう。例えば本研究の実装対象である Cell/B.E.には汎用的な処理を行うメインの CPU コアとは別に、演算用途に使われる小規模のサブのコアが複数あるが、これを仮想化して論理的なコアを提供し、内部で論理コアと実コアのスケジューリングを行うシステムソフトウェアが既に用意されている。しかしこれではサブコアの数が増えるとボトルネックになる可能性がある。またタスクをサブコアに割り付ける処理もこのシステムソフトウェアで行っているため、タスクが増えてくるとそれだけスケジューリングのオーバーヘッドが増大するという問題もある。

そこで本論文では、タスクをロード、実行する処理をサブコア側に実装することで、メインのコアに実装されたシステムソフトウェアにかかる負荷を分散する手法を提案する。メインコアのシステムソフトウェアとサブコアの実行基盤が連携動作してタスクの実行を行うことで、サブコアをより効率的に利用することができる。本論文では次章以降でこの手法を Cell/B.E.に対して実装し、既存環境との比較評価を行う。

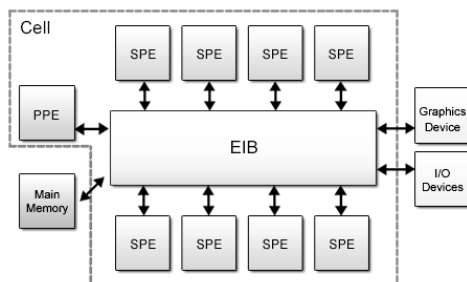


図 1 Cell/B.E.の全体構成

3. Cell Broadband Engine アーキテクチャと既存環境

3.1 アーキテクチャ概要

本研究の対象となる「Cell Broadband Engine」(本論文では以降「Cell/B.E.」と表記する)[4][5][6]は、汎用的なプロセッサコアである PPE (PowerPC Processor Element) と演算用プロセッサコアである複数の SPE (Synergistic Processor Element) により構成されたヘテロジニアスマルチコア CPU である。Cell/B.E.の全体構成を図 1 に示す^{*1}。PPE や SPE は、メインメモリや他のデバイスなどとともに EIB (Element Interconnect Bus) とよばれる高速なバスで接続されている。Cell/B.E.は SPE を用いた高い並列演算性能が特徴であり、2.4GHz 駆動の Cell/B.E.を用いたある計算実験[7]では、3 つの SPE を用いた場合に Pentium4 3.6GHz クラス CPU の 2 倍のパフォーマンスを実現するのに対し、SPE を用いない場合は Pentium4 CPU の 20%程度のパフォーマンスにまで低下するという結果が出ている。よって Cell/B.E.の持っている能力を最大限に引き出すためには、SPE の稼働率を高めてやらなければならない。

SPE は Cell/B.E.独自の SIMD 型アーキテクチャを採用する CPU コアであり、ストリームデータを処理することに特化している。SPE の構成を図 2 に示す。PPE はメインメモリを直接参照することができるのに対して、SPE が直接参照できるのはコア内部に有する 256KB のローカルストア (Local Store: 本論文では以降「LS」と表記する) のみである。PPE と SPE はキャッシュを共有しないため、メインメモ

^{*1} http://cell.fixstars.com/ps3linux/images/5/5a/FI_GURE-01-01.png より引用。

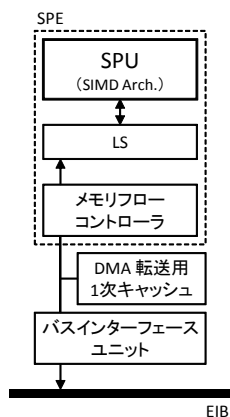


図2 SPEの構造

リと LS との間のデータ転送は DMA を用いてソフトウェアが明示的に行う必要がある。

アプリケーションで SPE を利用するためには、まずメインメモリに展開した SPE 用のプログラムを LS へロードし、SPE を起動してロードしたプログラムの実行を開始する。SPE はメインメモリの内容を直接参照できないので、SPE 用のプログラムはまず処理に必要な入力データを格納するバッファを LS 内に確保し、DMA 転送を用いてメインメモリからバッファへ転送する。その後 SPE 用のプログラムは入力データに対して演算を行い、処理の結果となる出力データを用意する。出力データは入力データのとくと同じく、SPE 用のプログラムが DMA 転送で LS からメインメモリへ転送する。この転送処理は SPE 内の MFC (Memory Flow Controller) が行い、演算装置である SPU (Synergistic Processor Unit) とは切り離されているため、DMA 転送は命令実行とは非同期に処理することができる。SPE 用のプログラムを素直に実装すると、データ転送の際に“待ち”が入ってしまうが、このしきみを用いることにより、LS 内に 2 つのバッファを有し、一方のバッファの転送を行っている最中にもう一方のバッファに対して演算を行うことで、SPE の稼働率を上げることができる。

3.2 spufs と libspe2

Cell/B.E.上で動作する Linux には、SPE を仮想化して論理的な SPE を提供する仮想ファイルシステム「spufs」^[8]とそれを利用するためのライブラリ

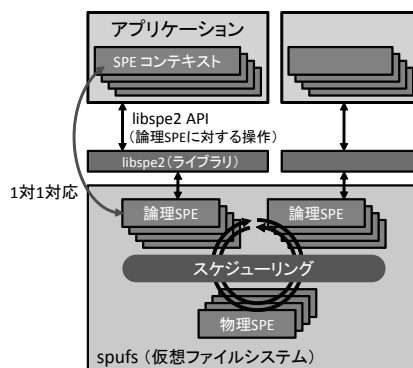


図3 spufs/libspe2によるSPEの利用方法

「libspe2」^[9]が用意されており、SPE を用いるアプリケーションを開発できる環境が既に確立されている。spufs と libspe2 を用いた SPE の利用方法を図 3 に示す。PPE 上で動作するアプリケーションは、libspe2 を用いて「SPE コンテキスト」を生成する。この SPE コンテキストは spufs が提供する論理 SPE と 1 対 1 で対応しており、この論理 SPE は spufs によって実 SPE にスケジューリングされる。SPE コンテキストの生成後は、libspe2 API を介してこのコンテキストに SPE 用のプログラムをロードし、実行を指示する。一度実行を指示すると、終了するまでアプリケーションに制御が返らないため、複数の SPE コンテキストを用いて並列実行させる場合は pthread などを併用する。

3.3 MARS (Multicore Application Runtime System)

「MARS」^[10]はマルチコアプロセッサの MPU (MicroProcessing Unit) 上で動作するユーザプログラムの生成や管理を容易にするためのライブラリである。MARS は 1 個以上の MPU と、それを管理、制御する単一のホストプロセッサとで構成されるアーキテクチャを仮定しており、その特徴から明らかなように Cell/B.E.を強く意識したものとなっている。実際にホストプロセッサは Cell/B.E.においては PPU (PowerPC Processor Unit : PPE の演算装置) と、MPU は SPU と同義であると明記されており、現時点では Cell/B.E.専用のライブラリといえる。

MARS を用いた SPE の利用方法を図 4 に示す。MARS は MPU に軽量の「MARS カーネル」をロー

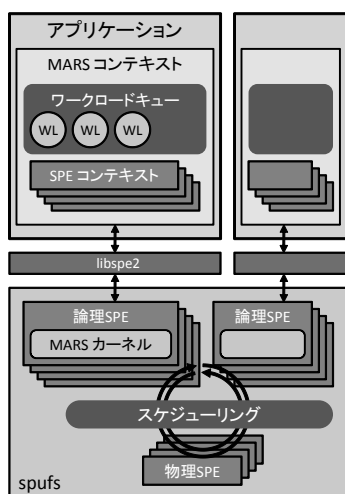


図4 MARSによるSPEの利用方法

ドし、実行待ちタスクのチェックやそのロードと実行、さらにタスクが同期をとるタイミングでの他のタスクとのコンテキストスイッチ処理などはこのカーネルが担当している。またアプリケーションは「MARS コンテキスト」を生成し、SPE で実行するタスクはこのコンテキスト内のワークロードキューにいったん保持される。MARS カーネルはこのキューからワークロードを取り出し、順次実行していく。

3.4 既存環境の問題点

spufs はアプリケーションが必要とする数の論理SPEを提供するが、その取り扱いはすべてアプリケーション側に任される。libspe2を利用してSPEコンテキストを生成し、プログラムをロードして実行を開始し、終了を待ち、ほかにSPE実行待ちのタスクがある場合は再度プログラムをロードして実行を繰り返す、これら一連の処理を、生成したSPEコンテキストの数だけプログラマが細かく制御しなければならない。またspufsによる論理SPEのスケジューリングにも難点があり、SPEのLSは256KBと小さくかつ仮想記憶を持たないため、コンテキストを待避しようとするLS内のすべてのデータをDMAを利用してメインメモリに転送しなければならない。さらにSPEはストリームデータを処理することに特化されたアーキテクチャであることから、コ

ンテキストスイッチが発生するとパフォーマンスが著しく低下する。そのため一度タスクが実行されたら終了するまでSPEを占有した方がよい。これらを踏まえてSPEにタスクを効率的に割り当てようとするときには、必然的にプログラマがアプリケーション内に簡易なSPEスケジューラを備えておかなければならない。

またMARSはMARSカーネルがワークロードを順次実行するため、アプリケーションがSPEを事細かに制御する必要はないが、MARSを用いている複数のアプリケーション間でワークロードキューを共有する設計にはなっていないので、あるアプリケーションで遊んでいる論理SPEがあったとしても、他のアプリケーションのワークロードを実行することができない。またMARS自身がspufsおよびlibspe2を利用して実装されているため、MARSのワークロードのスケジューリングと、spufsの論理SPEのスケジューリングで二重のオーバーヘッドが発生することになる。

4. 本研究の目標

spufsとlibspe2でSPEのタスクを実行する際には、アプリケーションが論理SPE（SPEコンテキスト）にタスクをロードし、実行を指示するという手順であった。このときアプリケーションが行うのはあくまでSPEに対する細かい制御であり、これによってSPEの使い方が複雑化してしまっている。そこで本研究ではアプリケーションに対して“SPEを直接操作する”のではなく、“SPEを仮想化したタスクを操作する”ためのインターフェースを提供する。SPEの確保やSPEに対するタスクの割り当てなど、タスクの操作に必要なSPEの制御は隠蔽し、アプリケーションはタスクの実行に必要な最低限の情報を与えるだけで、タスクの生成、実行、同期を行うことができるようにする。

次にSPEで実行するタスクの管理方針としてアプリケーションごとに管理するのではなく1ヶ所に集約する。また既存環境では各アプリケーションがタスクをSPEに割り当てていたが、各SPEにタスクをロード、実行する処理を担当させることで、割り当て処理をSPE自身が行うようにする。こうすることで空いたSPEが実行待ちのタスクにアクセスしやす

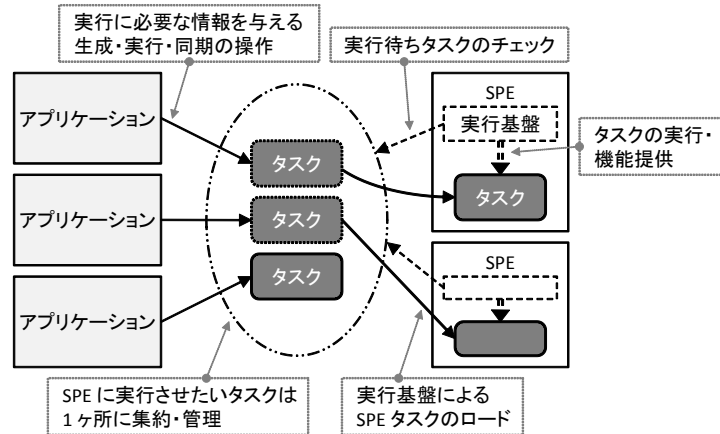


図5 本研究の全体構成

表1 spe_process_context_write_data_t 型

フィールド	型	説明
program_start	uint64_t	SPE 用プログラムの先頭を指すアドレス
program_size	uint32_t	SPE 用プログラムのサイズ
arg	int64_t	SPE 用プログラムに渡す引数
input_data_start	uint64_t	入力データの先頭を指すアドレス
input_data_size	uint32_t	入力データのサイズ
output_data_start	uint64_t	出力データの先頭を指すアドレス
output_data_size	uint32_t	出力データのサイズ

くなる。

さらに SPE のタスクの先頭と末尾にはそれぞれ入出力データの転送処理が記述されるが、これは MFC に対する問い合わせ処理の連続である。そこで本研究ではデータ転送を自動化するなどの、SPE のタスクにとって有用な機能を提供するための実行基盤を SPE 内に設けて、アプリケーションプログラマがデータ転送をとまう SPE のタスクを容易に記述できるようにする。

以上の目標を反映した全体構成を図5に示す。なお本章までで述べた、“SPE で実行するタスク”のことを次章以降は「SPE プロセス」または文脈に応じて単に「プロセス」と表記する。また SPE で実行するタスク（つまり SPE プロセス）を操作するためのインターフェースは、SPE をプロセスを管理するものとして「SPE プロセスマネージャ」と、SPE 内に設ける実行基盤は「SPE 向け軽量カーネル」とそれぞれ名前を付ける。

5. 設計

5.1 SPE プロセスマネージャの機能

アプリケーションが SPE プロセスを操作するために、SPE プロセスマネージャは以下に示す機能を備える。

1. SPE プロセスの実行に必要な情報を受け付ける機能
2. アプリケーションからの要求に応じて SPE プロセスを開始したり、一時停止したり、強制終了したりする機能
3. SPE プロセスの実行状況をアプリケーションに通知する機能

SPE プロセスの実行に必要な情報は表1に示す `spe_process_context_write_data_t` 型のデータとする。SPE プロセスとなるプログラムは直接与えるわけではなく、アプリケーションのメモリ空間に保

表 2 spe_process_context_read_data_t 型

フィールド	型	説 明
status	int32_t	SPE プロセスの実行状況
ret	uint64_t	SPE プロセスの戻り値
spe_no	int32_t	プロセスを実行した SPE

持しておき、そのアドレスとサイズをデータに含める。SPE は DMA 転送を用いてこのメモリ空間にアクセスできるため、プログラムを複数の SPE プロセスで使いたいときに個別に転送する必要がなくなる。また SPE プロセスの実行状況は表 2 に示す `spe_process_context_read_data_t` 型のデータである。このデータを確認して、SPE プロセスの終了を監視する。

5.2 SPE 向け軽量カーネルの機能

メインメモリと LS の間で起こる入出力データの転送処理は、既存環境ではアプリケーションプログラマが記述しなければならないが、本研究ではデータの転送処理は SPE 向け軽量カーネルの機能として SPE プロセスに提供される。データの転送元・先アドレスや転送サイズは表 1 で示した `spe_process_context_write_data_t` 型のデータに含まれている。SPE 向け軽量カーネルは、SPE プロセスを実行する前にメインメモリから入力データを転送し、同時に出力データの保存領域を確保する。その後、入力データの格納領域、出力データの保存領域それぞれの先頭アドレスを SPE プロセスに渡す。SPE プロセスはこの先頭アドレスをもとに間接参照でデータを読み書きすることができる。SPE プロセスが終了すると、SPE 向け軽量カーネルは出力データをメインメモリへ転送する。

6. 内部構成

6.1 SPE プロセスのコンテキストと SPE 実行待ちプロセスキュー

アプリケーションが SPE プロセスを開始すると、SPE プロセスのコンテキストが「SPE 実行待ちプロセスキュー」に追加される（ただしプロセスを実行していない空きの SPE がある場合を除く。6.2 節で後述）。ここで、SPE プロセスのコンテキストは 5.1 節で述べた `spe_process_context_write_data_t` 型

（表 1）のデータおよび `spe_process_context_read_data_t` 型（表 2）のデータの 2 つを指す。SPE 向け軽量カーネルが既に実行している SPE プロセスを完了すると、SPE 向け軽量カーネルは実行待ちプロセスキューをチェックし、プロセスがある場合には（この SPE がプロセスを実行するため）キューからこれを削除し、プロセスのロード、実行を行う。

6.2 SPE プロセスマネージャと SPE 向け軽量カーネルの連携

アプリケーションが SPE プロセスを開始してからプロセスの実行が終了するまでの内部動作を図 6（SPE プロセス実行前）と図 7（SPE プロセス終了後）に示す。SPE プロセスマネージャは SPE プロセスの実行開始要求を受ける（図 6(3)）と、プロセスを実行していない空きの SPE を探す。もし SPE に空きがない場合はそのまま SPE プロセスをキューイングして待つが、そうでない場合は空き SPE の軽量カーネルに対して実行待ちの SPE プロセスがあることを伝え、実行待ちプロセスを実行するよう指示する（図 6(5)）。指示された SPE 向け軽量カーネルは SPE プロセスのコンテキストにしたがって、プロセスとなる SPE 用のプログラムと入力データを DMA で転送し、SPE 用プログラムに制御を渡して SPE プロセスを開始する（図 6(7)～(9)）。SPE プロセスが終了すると、出力データを DMA で転送し、SPE プロセスマネージャにプロセスの終了を通知する（図 7(10)～(13)）。この時点でもし実行待ちの SPE プロセスがあれば、これを実行するよう指示する（図 7(14), 図 6(5)）。この繰り返しとなる。以上のように、SPE プロセスマネージャと SPE 向け軽量カーネルが連携をとりながら SPE プロセスを実行していく。

6.3 割り込みハンドラと SPE プロセス開始要求イベント

SPE 向け軽量カーネルがプロセスの実行を終了したことを SPE プロセスマネージャに通知する手段として、ストップ・アンド・シグナル (stop) 命令^[11]による PPE への割り込みを用いている。このとき、割り込みハンドラ内で SPE プロセスキューや終了した SPE プロセスのコンテキストを更新する必要がある。また SPE プロセスマネージャに対して SPE プロセスの開始要求がきた場合にも同様にコンテキスト

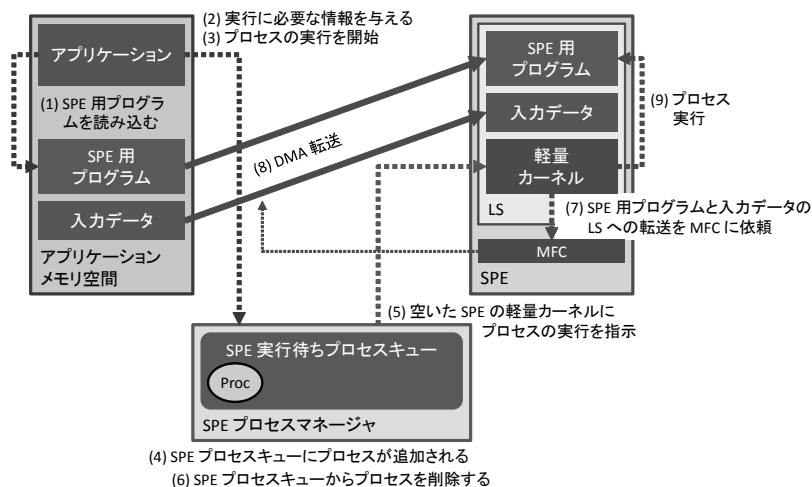


図 6 SPE プロセスマネージャと SPE 向け軽量カーネルの内部動作 (SPE プロセス実行前)

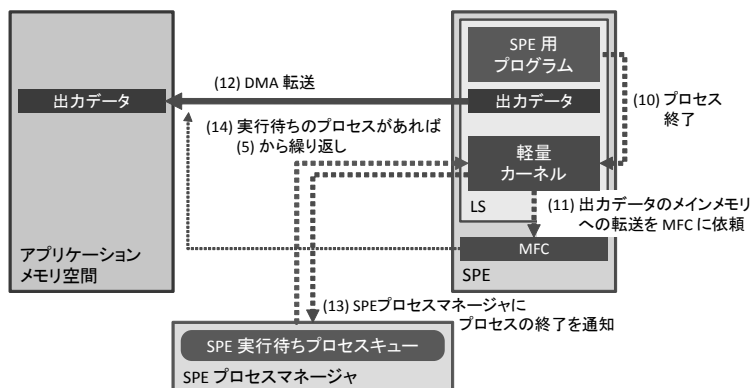


図 7 SPE プロセスマネージャと SPE 向け軽量カーネルの内部動作 (SPE プロセス終了後)

を更新しなければならない．そこで図 8 に示すようにセマフォとスピンロックを用いてクリティカルセクション内で更新処理を行うようにする．その後，割り込みハンドラでは実行待ちプロセスがあれば，SPE プロセス開始要求イベントのハンドラでは空きの SPE があれば，プロセス実行を軽量カーネルに指示する処理 `start_spe_process` を実行する．

7. 実装と評価

7.1 実装環境

SPE プロセスマネージャおよび SPE 軽量カーネルは，表 3 に示す環境で実装を行った．SPE プロセスマネージャは Linux のキャラクタデバイスとして実

表 3 実装環境

プラットフォーム	PLAYSTATION 3 (Cell/B.E. 3.2GHz, 7SPEs)
OS	Yellow Dog Linux 6.1
Linux カーネル	Version 2.6.28
gcc/spu-gcc	Version 4.1.2 (gcc) Version 4.1.1 (spu-gcc)

装し，SPE プロセスに対する操作は表 4 に示すシステムコールにそれぞれ対応している．6.3 節で述べた SPE プロセス開始要求イベントは，今回の実装では `ioctl1` システムコールのイベントにあたる．また本研究の実装にあたっては，`spufs` が提供する論理 SPE は本研究では不要になるため，`spufs` を用いず

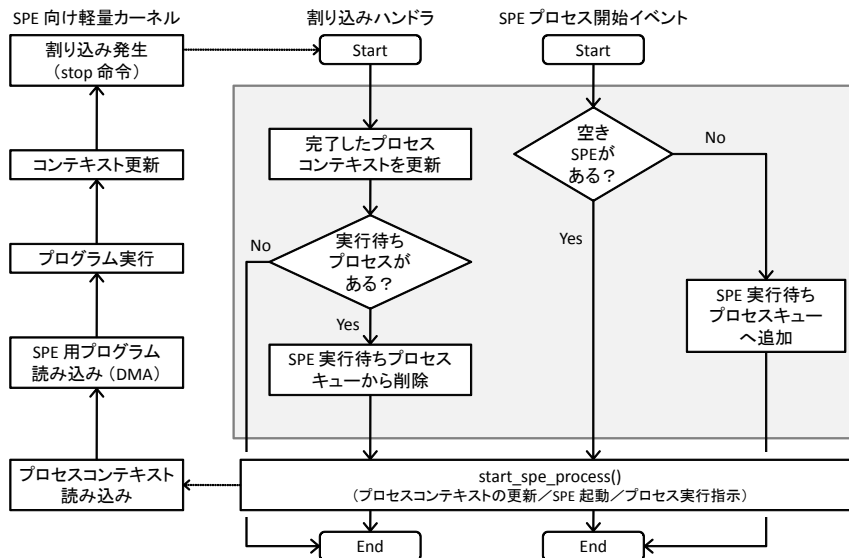


図 8 割り込みハンドラ，SPE プロセス開始要求イベントハンドラ内の処理の流れ
(網掛け部分はクリティカルセクションを表す)

表 4 システムコール一覧

open	SPE プロセスの登録準備
write	実行に必要なコンテキストを登録する
ioctl	SPE プロセスを操作する
read	現在のプロセスの実行状況を取得する
close	SPE プロセスの破棄

表 5 空タスクの実行時間（単位 ms）

タスク数	libspe2	MARS	本研究
10	15.492	5.031	4.240
50	66.914	26.783	17.547
100	141.764	59.183	32.599
500	692.639	249.081	160.219
1,000	(SEGV)	557.675	333.434

Cell/B.E.のハードウェアを操作して物理 SPE を直接管理することとした。これにより SPE プロセスの割り当てをより高速に行うことができるようにする。なお Yellow Dog Linux のカーネルイメージには標準で spufs が含まれているため、カーネルを再コンパイルして spufs が含まれていないカスタムカーネルイメージを作成して利用した。

7.2 空タスクの実行時間

本節では spufs と libspe2 を用いた環境（以降「libspe2」と表記する）、3.3 節で述べた MARS を用いた環境、および本研究の環境それぞれにおいて、SPE に何もしない空のタスク（本研究の環境では空の SPE プロセス）を実行させたときの実行時間を比較する。PLAYSTATION 3 で使用できる 6 個の SPE 全てを使い、空のタスクを 10, 50, 100, 500, 1,000 個作り、実行開始から全ての空タスクが終了するま

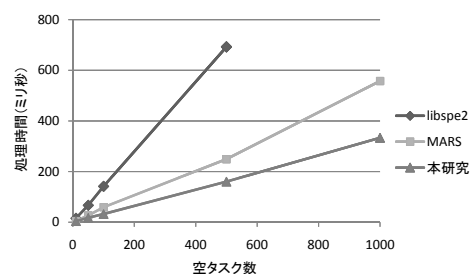


図 9 空タスクの実行時間

での時間を PPE アプリケーション側で計測する。計測の結果を表 5 および図 9 に示す。なお libspe2 の環境では、1,000 プロセス実行時にセグメンテーション違反が発生し、計測を行うことができなかった。

空タスクの実行時間を比較すると、本研究での実行時間は MARS の環境下で実行するよりも最大で 1.82 倍、libspe2 に至っては最大で 4.32 倍の高速化を

表 6 クイックソートの実行時間 (単位 ms)

要素数	libspe2	本研究
128	252.163	39.203
256	508.007	153.066
512	1,044.510	470.897
1,024	2,091.799	1,159.599

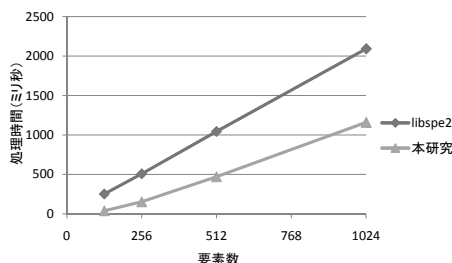


図 10 クイックソートの実行時間

実現することができた。既存の環境よりもオーバーヘッドが小さくなり、SPE プロセスマネージャおよび SPE 向け軽量カーネルによる SPE 割り当ての軽量化を実現できた。

7.3 クイックソートの実行時間

本節では libspe2 を用いた環境と本研究の環境のそれぞれにおいて、入力データに対してピボットを設定し、ピボットよりも大きい数と小さい数にグルーピングするまでの一連の処理を SPE プロセスとしたクイックソートの実行時間を比較する。7.2 節と同じく 6 個の SPE を全て使い、128, 256, 512, 1,024 要素のクイックソートにかかる時間を PPE アプリケーション側で計測する。計測の結果を表 6 および図 10 に示す。

クイックソートの実行時間の比較は libspe2 のみであったが、1,024 要素のソートで 1.80 倍、128 要素のソートでは最大で 6.43 倍の高速化を実現することができた。空のタスクだけでなく負荷のかかる一般的なアプリケーションでも本研究の環境によって高速化することが期待できる結果となった。

8. おわりに

本論文では来るべきマルチコア、メニーコアの時代に向けた、CPU コア間の連携によるタスク管理・実行システムを提示した上で、本研究ではこのシス

テムを Cell/B.E. に対して適用した。SPE プロセスマネージャおよび SPE 向け軽量カーネルを設計および実装し、この 2 つが連携動作することによって、アプリケーションが SPE を利用しやすくし、また SPE プロセスを容易に記述できるようにした。さらに Cell/B.E. のハードウェアを直接操作し、SPE プロセスの割り当てをより高速に行うことができるようにした。評価の結果、空のタスク、クイックソートとも既存環境より高速に実行することができ、本研究の有用性を示すことができた。

既存環境において、DMA 転送はすべてアプリケーションで管理しなければならず、本研究でもそれは変わらなかった。この DMA 転送の適切な管理方法を考えていくことを今後の課題としたい。この問題を解決し、SPE プログラムの生産性を向上させるために、DMA に関する面倒な手続きをどのように隠蔽していくかを今後考察していきたいと思う。

参考文献

- [1] 大塚 実: インテル、メニーコア化への取り組みなど、研究活動に関する説明会を開催。マイコミジャーナル <http://journal.mycom.co.jp/news/2005/11/09/003.html>, 2005.
- [2] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn and Tim Purcell: A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum* 26 (1), pp. 80-113, 2007.
- [3] Tom R. Halfhill: Parallel Processing With CUDA. *Microprocessor Report MPR Newsletter* 01/28/08-01, pp. 1-8, 2008.
- [4] International Business Machines Corporation, Sony Computer Entertainment Incorporated and Toshiba Corporation: Cell Broadband Engine Architecture Version 1.01. 2006.
- [5] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maerur and D. Shippy: Introduction to the Cell multiprocessor. *IBM Journal of Research and Development* 49 (4), pp. 589-604, 2005.
- [6] T. Chen, R. Raghavan, J. N. Dale and E. Iwata: Cell Broadband Engine Architecture and its first implementation. *IBM Journal of Research and Development* 51 (5), pp. 559-572, 2007.

- [7] 西川 善司: 西川善司の3D ゲームファンのための PS3 パフォーマンス考察講座～ PS3 は SPE で加速する！ SPE がなければ性能は Pentium 4 の 1/5 以下？. *GAME Watch*, 2006.
- [8] Arnd Bergmann: Spufs: The Cell Synergistic Processing Unit as a virtual file system. *IBM DeveloperWorks*, 2005.
- [9] International Business Machines Corporation: SPE Runtime Management Library Version 2.2. *Cell Broadband Engine Architecture Joint Software Reference Environment Series*, 2007.
- [10] Sony Corporation of America: MARS - Multicore Application Runtime System. <http://ftp.uk.linux.org/pub/linux/Sony-PS3/mars/latest/mars-docs-1.1.1/html/>, 2008.
- [11] International Business Machines Corporation, Sony Computer Entertainment Incorporated and Toshiba Corporation: Synergistic Processor Unit Instruction Set Architecture Version 1.2. 2007.