システムソフトウェアと
オペレーティング・システム 67－5
(1994. 12. 8)

並列性と移植性を考慮したユーザレベル・スレッドライブラリ PPL の実装と評価

坂本 力†, 宮崎 輝樹‡, 桑山 雅行†, 大石 幸雄¶, 最所 圭三¶, 福田 晃¶

†九州大学大学院工学研究科情報工学専攻,
‡新日本製鐵株式会社
¶奈良先端科学技術大学院大学情報科学研究科

〒630-01 奈良県生駒市高山町 8916-5

あらまし 現在、並行/並列処理の実行単位であるスレッドを、ユーザレベルで提供するスレッドライブラリが多く実現されている。本稿では、並列性と移植性を考慮して開発を行なっているユーザレベルスレッドライブラリ PPL について述べる。PPL では、並列性の実現のため、ユーザレベル・スレッドをマルチルーチン方式を用いて実現した。また、PPL 内部を仮想プロセッサ依存部と非依存部を分離した構造とすることで、ライブラリ自体の移植性の向上を図った。実際にいくつかのシステムに実装し、PPL の提供するスレッドの軽量性を他のユーザレベル・スレッドと比較し、マルチプロセッサ・システム上での並列性の評価を行なった。

和文キーワード ユーザレベルスレッド, スレッドライブラリ, 仮想プロセッサ, 移植性, 性能評価

Parallel Pthread Library(PPL): User-Level Thread Library with Parallelism and Portability —— Implementation and Evaluation ——

Chikara Sakamoto†, Teruki Miyazaki‡, Masayuki Kuwayama†, Yukio Oishi¶, Keizo Saisho¶ and Akira FUKUDA¶

†Department of Computer Science and Communication Engineering, Kyushu University,
‡Nippon Steel Corporation
¶Graduate School of Information Science, Nara Institute of Science and Technology,

Takayama 8916-5, Ikoma, Nara 630-01, JAPAN

Abstract

There have been many user-level thread libraries that support manipulation of thread at user level. We are developing Parallel Pthread Library(PPL), a user-level thread library with parallelism and portability.

For parallelism, we employ the multiroutine approach. For portability, we devide PPL into two parts, virtual prosessor dependent part and independent part.

We compare the basic performance of PPL with that of other existing user-level thread libraries, and evaluate parallelism of PPL on a multiprosessor system.

英文 key words user-level thread, thread library, virtual processor, portability, performance evaluation

1 はじめに

近年の並行/並列処理の研究においては、実行単位としてスレッド（軽量プロセス）と呼ばれる概念が用いられるようになってきた。スレッドとは、従来のUNIXにおけるプロセスのような実行単位から、プログラムカウンタやスタックポインタ、およびスタックなどのような実行軌跡となる部分を分離・独立させた制御の流れである。スレッドは、プロセスと比較してその生成や消滅などの操作のオーバヘッドを小さく抑えることができる、同期や通信を高速かつ容易に行なうことができるなどの利点を持つ。

現在、このスレッドの実現方法としていくつか提案されているが、本稿では、それらの実現方式における問題点を指摘し、それらの問題点を解決するうえで必要な要件をみたすために我々が研究・開発しているParallel Pthread Library(PPL)[1] について述べる。また、PPL を実際のシステムに実装して行なった性能評価についても述べる。

2 スレッドの実現方式と問題点

2.1 スレッドの実現方式

現在、スレッドシステムについて多くの研究がなされ、様々なスレッドシステムが実現されている。それらの実現方式は、以下の二つに分類される。

2.1.1 カーネル・スレッド

オペレーティングシステム（以下OS）のカーネルによって提供されるスレッドである（図1）。カーネル・スレッドはMach[2] などのマルチプロセッサ・システムや分散環境を対象としたOSで実装されている。

カーネル制御方式では、OSのカーネルがスレッドを制御するため、1つ1つのスレッドがプロセッサなどの割当の単位となる。そのため、システムの持つ並列性を充分に考慮したスレッドのスケジューリングを行なうことができるという利点を持つ。しかし問題点として、スレッドの生成・消滅などの全ての操作にカーネルが介在するため、スレッド操作のオーバヘッドが大きくなるという点が挙げられる。

2.1.2 ユーザレベル・スレッド

OSのカーネルが制御を行なわずユーザレベルで実現・操作を行なうスレッドをユーザレベル・スレッド

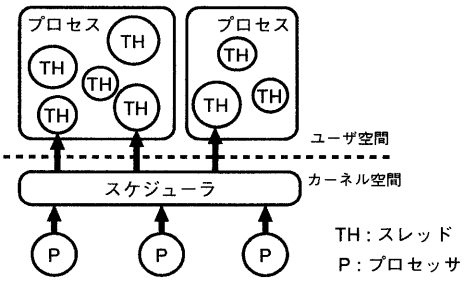

図 1: カーネル制御方式

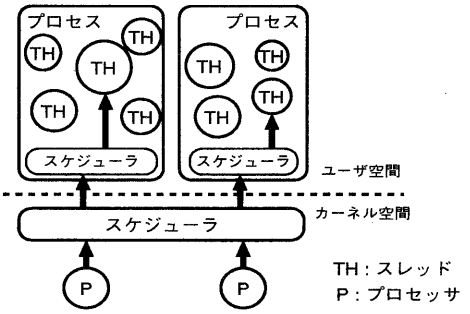

図 2: コルーチン方式

と呼ぶ。ユーザレベル・スレッドは以下のコルーチン方式、マルチルーチン方式に分類される。

本方式の利点は軽量性である。本方式においては、スレッドの生成や消滅などの全てのスレッド操作をユーザレベルで行うため、オーバヘッドを非常に小さく抑えることができる。

コルーチン方式

スレッドをプロセス上のコルーチンとして実現する方式である（図2）。コルーチンは、複数の手続きに順に制御を移動させながら実行するもので、ライブラリや言語環境としてその操作が提供される。コルーチン方式の例は、SunOS の LWP(Light-Weight Process)[3] などがある。

本方式の利点は実現が容易なことである。しかし、問題点として、各スレッドをコルーチンとして実現しているため、スレッドを逐次的にしか実行できないという点が挙げられる。

マルチルーチン方式

マルチルーチン方式ではユーザレベル・スレッドを、OS が提供する複数の仮想プロセッサ上で実行することで実際に並列実行を行なうことができる（図3）。仮

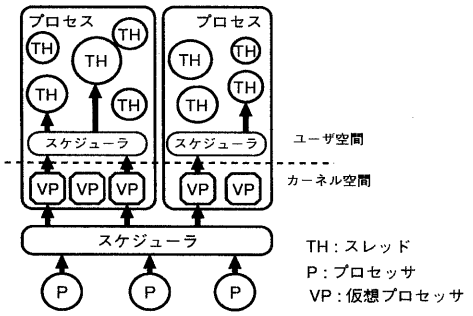

図 3: マルチルーチン方式

想プロセッサとは、マルチプロセッサ・システム上において、OS がプロセッサ割当の単位として提供するプロセスやカーネル・スレッドを指す。マルチルーチン方式を用いることによって、ユーザレベル・スレッドを並列に実行することができ、軽量性と並列性を兼ね備えたスレッドシステムを実現することが可能となる。しかし、ライブラリ自体の持つデータ構造やスレッド間で共有しなければいけないデータについて仮想プロセッサ間で一貫性を保つことが必要である。そのため、その実現がコルーチン方式と比較して困難である。Mach の Cthreads は Mach のカーネル・スレッドを仮想プロセッサとして動作するマルチルーチン方式として実現されている [4]。

2.2 問題点

既存のスレッドシステムが持つ問題点として移植性があげられる。それらは実装されているシステムに大きく依存しているものが多く、そのため移植は困難である。また、それらのスレッドシステムは、ユーザに対して独自のインタフェースを提供しているため、ユーザがスレッドを用いた移植性の高いアプリケーションを記述することが難しい。そのため、多くの OS 上でユーザに対して共通のスレッドインタフェースを提供できる移植性の高いスレッドシステムの実現が望まれる。

3 関連研究

現在、多くのユーザレベルのスレッドシステムがライブラリとして提供されている。一般に公開されているスレッドライブラリには、Mach の Cthreads、Florida 大学の Muller による SunOS 専用のスレッドライブラリ [5]（ここでは FPL と呼ぶ）、大阪大学の阿部らによる Portable Thread Library(PTL)[6] などがある。

Cthreads は Mach OS においてユーザにライブラリの形式で提供されており、スレッドは Mach thread を仮想プロセッサとして実行される。Mach の Cthreads はマルチルーチン方式を用いているため、並列性を備えている。しかし、仮想プロセッサである Mach thread に大きく依存しているため、Mach 以外の OS への移植は困難である。

FPL は対象を SunOS(Sparc) に限定することにより、高速なスレッドライブラリを実現している。そのため、並列性や移植性を持っていない。

PTL は、対象を BSD UNIX としており、ほとんど変更することなくそれらの UNIX 上で動作させることができ、移植性に優れている。また、スタックの自動拡張機能や、ノンブロッキング I/O、デバッグ機能等多くの機能を提供している。しかし、PTL においてはスレッドを一つの UNIX プロセス中のコルーチンとして実現しているため、並列性を持っていない。

このように、既存のスレッドシステムには並列性と移植性の 2 つの観点からみてまだ問題が残されているといえる。

4 PPL の設計

4.1 PPL の設計方針

第 2、3節で挙げた問題点、現状、それに対する要望に基づき、以下の 2 点を目標として PPL を設計した。

- マルチプロセッサ・システムにおいて、OS がプロセッサ割当の単位として提供する仮想プロセッサ上で並列に実行可能なユーザレベル・スレッドを提供する。
- 多くの OS で共通のスレッドインタフェースを提供するために、移植性を考慮したスレッドライブラリとして実現する。

4.2 並列性の実現

PPL では、マルチルーチン方式を用いてスレッドを実現することで並列性を実現する。その際、OS によって提供される仮想プロセッサのモデルは異なっているため、これらに対応できるようにする。

4.3 移植性の実現

多くのシステム上でユーザに対して共通のインタフェースを提供するために、PPL では以下の 2 点を考慮した。

1) 移植性の高いライブラリ
2) ユーザに対する共通のインタフェースの提供

ライブラリの移植性

ライブラリの移植時に問題となるのは、システムが提供する仮想プロセッサが OS によって異なる場合、仮想プロセッサのモデルが OS によって違ってくることである。例えば、UNIX ではプロセス、Mach では Mach thread という形で仮想プロセッサが与えられ、その生成や消滅などのインタフェースはそれぞれ異なった形で与えられる。

PPL では、ライブラリの内部構造を、仮想プロセッサとのインタフェースを扱う仮想プロセッサ依存部と、依存しない仮想プロセッサ非依存部とに明確に分離した（図 4）。仮想プロセッサ依存部から仮想プロセッサ非依存部へのインタフェースは仮想プロセッサの生成（vp_create)、消滅（vp_terminate）などのように VP インタフェースと呼ぶインタフェースに統一している。これによって、ライブラリの移植時に変更しなければならない部分は仮想プロセッサ依存部のみに限定することができ、PPL 自体の移植性を向上できる。仮想プロセッサ依存部、非依存部については、4.4で詳細を示す。

ユーザ・インタフェース

ユーザに対して共通のインタフェースを提供するために、PPL ではスレッドインタフェースとして pthread インタフェースを採用し、ユーザに提供する。

pthread インタフェースとは、IEEE が POSIX において規定しているスレッドを扱うための標準的なインタフェース (POSIX 1003.4a Thread Extension) を指す。今後この pthread インタフェースを備えたスレッド機能が多くの OS において提供されることが予想され、PPL においても pthread インタフェースを提供することにより、ライブラリ上のアプリケーションの移植性を向上させることができる。

4.4 PPL の構成

4.4.1 仮想プロセッサ依存部

仮想プロセッサ依存部は OS によって提供される様々な仮想プロセッサが与えるインタフェースの違いを吸収し、仮想プロセッサ非依存部に対して、統一的なインタフェース（VP インタフェース）を与えるものである。

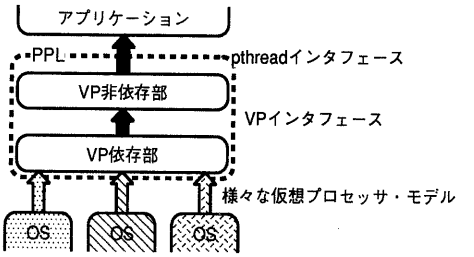

図 4: PPL の構成

VP インタフェース

仮想プロセッサ依存部から非依存部に対して提供する VP インタフェースにおいて、仮想プロセッサの操作に関するものとして以下の 4 つを提供する。

- vp_create()
  仮想プロセッサを生成する。
- vp_terminate()
  指定した仮想プロセッサを終了させる。
- vp_suspend()
  この関数を呼び出した仮想プロセッサを停止する。
- vp_resume()
  停止中の仮想プロセッサを再起動し、その仮想プロセッサ上でスレッドの実行再開などを行なう。

また、排他制御を行なうための機構として、スピンロックを提供する。スピンロックに関する関数として、以下の 4 つを提供する。

- spin_init()
- spin_lock()
- spin_try_lock()
- spin_unlock()

ここで、spin_lock() はロックが獲得されるまで待ち続けるもの、spin_try_lock() はロックが獲得できなかった場合は獲得失敗の値を返す関数である。

仮想プロセッサ生成

VP インタフェースを用いて仮想プロセッサを生成するタイミングとその個数については、静的に固定数用意する方法と動的に生成、消滅を行なう方法の 2 通

りが考えられる。仮想プロセッサの生成、消滅はカーネル内で行なわれるので、ユーザレベル・スレッドの生成、消滅と比較してオーバヘッドが大きい。

そこで、PPLでは、ライブラリ初期化時に固定数の仮想プロセッサを生成し、ライブラリの終了時にすべての仮想プロセッサを終了させる方法をとる。その時の個数は、最大でも実プロセッサ数と同数までとする。

しかし、将来的にはアプリケーションの並列度の変化に応じて、仮想プロセッサを動的に生成、消滅させ最適な並列実行を行なうことや、アプリケーション数の変化に応じて各アプリケーションに割り当てる仮想プロセッサ数を調節することなども考慮する必要があると思われる。

4.4.2 仮想プロセッサ非依存部

仮想プロセッサ非依存部では、仮想プロセッサ依存部によって提供される仮想的なマルチプロセッサ・システム上で並列動作するマルチルーチンとして、ユーザレベル・スレッドを実現する。またその際、ユーザに対してpthreadインタフェースを提供する。

以下に仮想プロセッサ非依存部について、特に移植性に影響を与えると思われる点、及び移植性を上げるための実現方法について述べる。

コンテキスト切替え

通常コンテキスト切替えは、プロセッサのレジスタ情報の保存・復元のためにアセンブリ言語によって記述され、システムに大きく依存する。このことが移植時の問題となる。

PPLでは、移植性の良いコンテキスト切替えを実現するために、C言語の標準関数であるsetjmp·longjmpを用いたコンテキスト切替えを行なう[6]。

スタックの確保

各スレッドには独立したスタックを割り当てなければならない。スレッドのためのスタックを確保する方法として、以下の4つの方法が考えられる[6]。

1. 静的データ領域
   データ領域に静的にスタックを確保する方法である。もっとも単純であり、どのようなOSでも利用できるが、静的にしかスタックを用意できないために、柔軟性がない。また、スタックの溢れを自動的に検出できない。

2. ヒープ領域
   mallocによってヒープ領域にスタックを確保する方法である。多くのスレッドライブラリで使われている方法であり、ほとんどのOSで利用できる。データ領域に静的に確保する場合と異なり、動的に任意の大きさのスタックを確保することができる。しかし、スタックの溢れを自動的に検出できない。

3. 共有メモリ
   共有メモリ機構をカーネルがサポートしている場合、共有メモリセグメントをスタック領域として用いることができる。共有メモリ領域をスタックとして用いることにより、スタックが成長して共有メモリの下限を越えた場合、不正アクセスのシグナルが発生するので、シグナルを捕らえることによりスタックの溢れを検出することができる。

4. Redzone Protect スタック
   mprotectシステムコールがサポートされている場合、スタックの下限から一定領域を読み書き禁止に設定することにより、スタックの溢れを検出する。スタックの確保はヒープ領域上に行なう。

PPLでは、最近のほとんどのUNIXで実現されており、スタックのオーバフローの検出が可能であるということから、スタック領域として、共有メモリを用いる。

スタックポインタの設定

新しく生成したスレッドに初めてコンテキスト切替えを起こす際に、スタックポインタ（SP）をそのスレッドに割り当てられたスタックを指すように設定することが問題となる。SPを設定する方法には、以下の3つの方法が考えられる。

1. 直接設定する。
   SPレジスタに直接新しいSPの値を設定する。アセンブラで記述する必要があり、プロセッサおよびアセンブリ言語に関する知識が必要となる。そのためシステム間の移植性はほとんどない。

2. 間接的に指定する。
   一度setjmpを実行してコンテキストを保存し、コンテキストを保存するjmp_bufのSPを格納する場所に新しいスレッドのスタックを指すSPを設定する。ここで、longjmpを引き起こすことにより、新しいスタックを用いて、setjmpを起こした箇所から実行を行なうことができる。この方法はC言語でも記述することができるが、システ

ム毎にjmp_bufの構造が異なるため、システムのjmp_bufの構造を知っている必要がある。

3. シグナルスタックを用いる[6]。
   4.2BSD以降のバークレイUNIXではシグナルスタック機構が用意されている。シグナルスタック機構とは、シグナルハンドラを実行する際のスタックを指定することができる機能で、スタックのオーバフローによる不正アクセスのシグナルを受けとり、正常に対処するために用意された機能である。このシグナルスタック機構を用いて、実行を開始するスレッドに割り当てたスタックをシグナルスタックとして設定し、明示的にシグナルを引き起こすことにより、間接的にSPを設定することができる。

方法2は、ユーザレベルのみで実行可能であり高速なコンテキスト切替えを行なうことができる。システムへの依存性は、jmp_buf内のSPを格納する領域の先頭からのオフセットが異なるだけであるため、移植時の問題も少ない。

また、方法3を用いることにより、4.2以降のUNIXでは、変更なしにSPの設定を行なえる。この方法ではスレッド初期化時のコンテキスト切替えの際にシグナルを起こしてハンドラへジャンプするため、カーネルが介在する。しかし、これは初期化時のみであるので、この際のオーバヘッドは問題にはならない。

PPLでは、移植性の向上のため方法3を用いる。しかし、仮想プロセッサがシグナルを受け取れない場合には方法2を用いる。

5 PPLの実装

我々はSun IPX、OMRON LUNA88K、またマルチプロセッサ・システムであるSilicon Graphics CHALLENGE上にPPLを実装した。各システムにおけるプロセッサ、OS、仮想プロセッサを以下に示す（表1）。

Sun IPXにおいては、仮想プロセッサは1つとした。よって仮想プロセッサ依存部はなにも行なわず、スレッドの実行もコルーチン方式と同様になる。

ライブラリ内の排他制御の問題

CHALLENGEなどのマルチプロセッサ・システム上で並列実行を行なう場合、レディキュー等のライブラリ内部のデータ構造へのアクセスはクリティカルセクションとして排他制御を行なって実行する必要がある。この際、排他制御を取る単位が問題となる。

PPLでは、現在単純にライブラリ全体で1つのロック変数を用意している。この場合、ライブラリ全体でクリティカルセクションを実行できるスレッドは1つであり、それがたとえ別のクリティカルセクションであっても他のスレッドは実行を妨げられることになる。つまり、ロックによるオーバヘッドが発生する。これは、実際に並列に動作しているスレッドが多いほど顕著に現れる。

これを解決するため、各データ構造毎にロック変数を用意することが考えられるが、多くのロック変数を管理する必要があるため、ライブラリのプログラミングが複雑になる。また、多くの操作は複数の構造体に同時にアクセスする必要があるため、デッドロック状態を防ぐための処置も必要となる。

しかし、スレッドの並列実行の効率を損なわないために、今後複数のロック変数を用意し別々のデータ構造へは並列にアクセス可能にする必要があると考えている。

6 評価

6.1 軽量性の評価

PPLによって提供するスレッドの軽量性の評価のために、スレッド操作の性能を評価する。評価は他のスレッドライブラリとの比較によって行なう。比較対象として、Sun IPX上ではPTL、FPTを、LUNA88K上では、PTL、Cthreadsを用いた。

比較項目

比較、評価を行なうスレッド操作としてコンテキスト切替えとスレッドの生成を行った。

- **コンテキスト切替え**
  2つのスレッドを生成する。それらのスレッドのコンテキスト切替えを1000回繰り返し、それにかかる時間を測定する。ただし、純粋にコンテキスト切替えの実行時間のみを測定するために、PPLの仮想プロセッサは1つで行なう。
- **スレッドの生成**
  100個のスレッドを生成し、それにかかる時間を測定する。

上記の2つのテストプログラムを用いて、実行を行なった結果を表2に示す。測定時間は実時間であり、他のユーザの影響を受けないよう配慮して測定を行なった。CHALLENGEにおいては、現在比較を行なうス

表 1: 実装を行なったシステムの概要

	Sun IPX	LUNA88K	CHALLENGE
プロセッサ	Sparc × 1	MC88100 × 4	MIPS R4400 MC × 36
OS	SunOS 4.1.3	Mach2.5	IRIX Release5.2
仮想プロセッサ	UNIX プロセス	Mach thread	UNIX プロセス

表 2: スレッド操作のオーバヘッド (μsec)

	Sun IPX			LUNA88K			CHALLENGE
	PPL	PTL	FPL	PPL	PTL	Cthreads	PPL
コンテキスト切替え	130	85	35	180	350	50	35
スレッド生成	1400	1800	1700	3000	2600	6500	1200

レッドライブラリを実装していないので、PPL における値のみを示す。

表2より以下のことがわかる。

- コンテキスト切替えの速さは、アセンブラによって切替えを行なっている FPL, Cthreads には及ばないが、setjmp、longjmp によって切替えを行なっている PTL と同程度とみなすことができる。
- スレッド生成の速さも他のライブラリと同程度である。

6.2 並列性の評価

マルチプロセッサ・システムである CHALLENGE において仮想プロセッサを 1 つとした場合と、仮想プロセッサを複数生成した場合を比較することによって、並列実行を行なうマルチルーチンの評価を行なうことができる。

テストプログラム

テストプログラムとして行列の乗算を行なうプログラムを用いた。このプログラムでは、N × N 行列の乗算において N 個のスレッドを生成し、各スレッドは 1 行（N 個の要素）の計算を行なう。スレッドの流れを図5に示す。今回の評価では 30 × 30 の行列の乗算を行ない全実行時間、並列実行時間を測定した。測定の結果を図6に示す。

図6より以下のことがわかる。

- 仮想プロセッサ数 8 程度までは、並列実行の効果を確認することができる。しかし、それ以上では効果が現れていない。これは、第5節で述べ

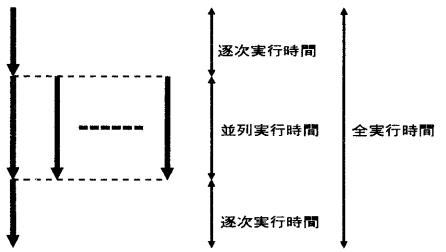

図 5: 行列乗算におけるスレッドの流れ

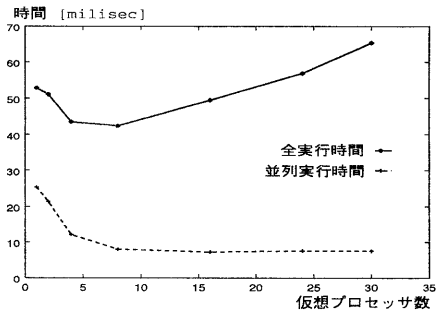

図 6: 行列乗算の実行時間

た、ロックによるオーバヘッドによるものと考えられる。

- 全実行時間の増加は、逐次実行時間に行なわれるスレッド生成の時間が増加したためである。これは、スレッド生成の間に他のアイドルプロセッサはビジーウェイトを行なっているため、共有メモリ上へのアクセス競合によってスレッド生成が妨げられていることによると考えられる。

6.3 移植性の評価

移植性については、明確な評価基準が存在しないため、評価が困難である。

しかし、今回PPLをSun IPX(OS:SunOS 4.1.3)からLUNA88K(OS:Mach2.5)へ移植する際に変更した仮想プロセッサ依存部のコードは全体の約20%であった。またCHALLENGE(OS:IRIX Release5.2)への移植に際しては全体の約15%に変更、追加を行なった。このことから、現在仮想プロセッサ依存部として分離している部分が全体に占める割合はそれほど大きくなく、今後PPLを他のシステムに移植する際に、変更すべき部分がほぼこの15〜20%に限定されるならば移植性に優れるといえると思われる。

しかし、様々なシステムへの移植を通して今後仮想プロセッサ非依存部への変更が必要となることも充分考えられ、仮想プロセッサ依存部、非依存部の分離基準については今後も充分検討していかなければならない。

7 おわりに

本稿では、既存のスレッドシステムに見られる問題点を整理し、それらを解決するために必要とされる要件を満たすために、並列性と移植性とを考慮して我々が研究、開発を行なっているユーザレベル・スレッドライブラリPPLについて述べた。

PPLでは、様々なシステムにおいてOSがプロセッサ割当の単位として提供する仮想プロセッサ上で並列実行されるマルチルーチンとして、ユーザレベル・スレッドを実現した。これによって、ユーザレベル・スレッドの軽量性とカーネル・スレッドの並列性を兼ね備えたスレッドの提供が可能となる。また、移植性を向上させるために、PPLの内部構造を仮想プロセッサ依存部、非依存部に明確に分離した。これにより、移植時に変更しなければならない部分を仮想プロセッサ依存部のみに限定することができ、ライブラリ自体の移植性を向上させることができる。

更に、実際のシステム上に実装を行なったPPLについて、スレッドの軽量性を他のユーザレベル・スレッドライブラリと比較した。この結果、スレッド操作のオーバヘッドは他のユーザレベル・ライブラリと同程度であることが確認された。また、マルチプロセッサ・システムであるCHALLENGE上で、テストプログラムを用いて並列性の評価を行なった。この結果、並列実行の効果と共有メモリへのアクセス競合の影響が確認された。

今後の課題として、以下のものが挙げられる。

- 他のマルチプロセッサ・システム上への移植
- 他の様々なOS上への移植。
- 並列性のタイプの異なる実際のアプリケーションによる性能評価。
- 動的な仮想プロセッサ生成アルゴリズムの検討。
- 未実装のpthreadインタフェースの実装、およびpthreadインタフェースそのものの検討。
- 仮想プロセッサ依存部、非依存部、およびVPインタフェースの検討。

参考文献

[1] T.Miyazaki, C.Sakamoto, M.Kuwayama, K.Saisho and A.Fukuda: Parallel Pthread Library(PPL): User-level Thread Library with Parallelism and Portability, Proc. COMPSAC'94, pp.301-306(1994).

[2] M.Accetta, R.Baron, W.Bolosky, D.Golub, R.Rashid, A.Tovaniam and M.Young: Mach: A New Kernel Foundation for UNIX Development, Proc. of the Summer 1986 USENIX Technical Conf., pp.93-112(1986).

[3] Sun Microsystems: SunOS Reference Manual (1988).

[4] C.Cooper, P.Draves: CThreads, Technical Report CMU-CS-88-154, Carnegie Melon University(1988).

[5] F.Mueller: A Library Implementation of POSIX Threads under UNIX, Proc. the Winter 1993 USENIX Technical Conf., pp.29-41(1993).

[6] 安部 広太, 松浦 敏雄, 谷口 健一: BSD UNIXの下でのポータブルマルチスレッドライブラリPTLの実現, 情報処理学会OS研究会(1994).

[7] 多田 好克, 寺田 実: 移植性・拡張性に優れたCのコルーチンライブラリー実現法, 電子情報通信学会論文誌 D-I Vol.J73-D-I No.12 pp.961-970(1990).