軽量プロセス・ライブラリ　Lesser Bear: スケジューラの並列化

小熊 寿　中山 泰一
電気通信大学　情報工学科
oguma-h@igo.cs.uec.ac.jp

筆者らは現在、SMP 型計算機の特徴を活かすため、並列性と移植性を備えた軽量プロセス (スレッド)・ライブラリ Lesser Bear の設計を行っている。Lesser Bear では、複数の UNIX プロセス (仮想プロセッサ) と各仮想プロセッサから均一にアクセスできる共有メモリ空間により、スレッドの並列動作を実現している。従来の Lesser Bear ではスレッドのコンテクスト切替えに、仮想プロセッサ間の排他制御が必要であった。そのため、スケジューラが仮想プロセッサ間で逐次化されていた。そこで本論文ではスケジューラを並列に動作させるために、1 つの共有メモリ空間を各仮想プロセッサごとの空間に区切り、その空間ごとにそれぞれ 2 つのキューを用意する。本論文では、キューへの挿入操作に一切のロック操作を必要としないアルゴリズムを利用する。このアルゴリズムにより、小さなオーバヘッドでスケジューリングを行う機構が実現できる。提案したスケジューリング機構を Lesser Bear 上で実現し、アプリケーションによる評価を行った。

1　はじめに

並列計算機は近年、パーソナル・コンピュータ (PC) でも実現されるようになり、より身近なものとなってきた。これによって、SMP 型計算機 (symmetric multiprocessor) と呼ばれる対称型の密結合型並列計算機に対応した UNIX が提供されるようになった。

このような SMP 型計算機に対応した UNIX 上で、効率的に並列処理をサポートする機構を実現することが急務とされる。とくに、UNIX プロセスより細かい実行単位 (軽量プロセスまたはスレッドと呼ばれる) による処理方法が必要とされている。

筆者らは現在、SMP 型計算機を活用するために、並列に動作し、移植性にも優れたスレッド・ライブラリ Lesser Bear の設計・実現を行っている [6]。

スレッド・ライブラリは高速に生成・切替えが可能という利点から、細粒度の並列アプリケーションで利用される。スレッドの粒度が小さい場合、スレッドの生成や消滅などを頻繁に行うためアプリケーションの実行時間に対し、スケジューリングなどのスレッド操作に要する時間の割合が増えてしまう。たとえば fork-join 型のアプリケーションでは、スレッド間で頻繁な同期操作を伴う。そのため、スレッドの粒度が細かくなるに従ってスケジューラが動作する割合が増える。

Lesser Bear ではスレッドを並列に動作させるために、複数の UNIX プロセスを仮想プロセッサとして生成している。そのため従来の Lesser Bear では、スレッドのスケジューリングを仮想プロセッサ間で排他的に行う必要があった。頻繁にコンテクスト切替えを行うような細粒度の fork-join 型アプリケーションが動作した場合、従来の Lesser Bear ではスケジューラが仮想プロセッサ間で逐次化してしまう。

本論文では、Lesser Bear 内部で並列にスケジューリングを行い、かつオーバヘッドを小さくしたスケジューリング機構の設計と実現を行う。スケジューリング機構を並列化するために、スレッドのスケジューリングは仮想プロセッサごとで行う。

従来の Lesser Bear では、内部で実行される数多くのスレッド・コンテクストが、すべての仮想プロセッサから均一にアクセスできる巨大な共有メモリ空間に保存されていた。提案する機構を実現するために本論文では、1 つの巨大な共有メモリ空間を各仮想プロセッサごとの空間に区切り、その空間ごとに、それぞれ 2 つのキュー (Protect Queue と Waiver Queue) を用意する。Protect Queue は担当する仮想プロセッサのみが挿入・削除を行うため、ロック操作を

必要としない。Waiver Queue は担当する仮想プロセッサのみが挿入を行い、任意の仮想プロセッサが削除を行う。そのため Waiver Queue への挿入には、ロック操作を必要としない。実現したスケジューリング機構では、キューへの挿入に一切のロック操作を必要としないため、小さなオーバヘッドでスレッドのスケジューリングが可能になる。

本論文で提案した新しいスケジューリング機構を、8 台の CPU をもつ SMP 型計算機上で性能評価実験を行った。アプリケーションによる実験を通じて、スケジューラの並列化による効果と、スケジューリングに要するオーバヘッド軽減の効果が確認された。

以下本論文では、2 章でこれまでの研究について述べ、3 章で新たに提案するスケジューリング機構について述べる。4 章では実験を通じて、提案したスケジューリング機構の評価を行い、5 章で本論文をまとめる。

2 従来のスレッド・ライブラリについて

本章では、スレッド・ライブラリに関するこれまでの研究と、筆者らが設計・実現しているスレッド・ライブラリ Lesser Bear について、その特徴を述べる。

2.1 関連研究

スレッドは、カーネルに手を加えて実現するもの (たとえば Scheduler Activations[2]) とユーザ・レベルのみで実現するもの (たとえば PTL[1]) とがある。カーネル・レベルでスレッドを実現すると、計算機アーキテクチャに適したシステムの構築が可能となるが、移植性を損なうという欠点がある。それに対しユーザ・レベルのみで実現するもの、すなわちスレッド・ライブラリは、アーキテクチャや OS に依らずに利用できるという利点がある。

このようなスレッド・ライブラリに関しては、これまで様々な研究が存在する [1, 3, 4, 7, 8, 9]。しかし移植性に優れていて、かつ、複数のスレッドを複数のプロセッサに割当て可能なライブラリはこれまで存在していない。

従来のスレッド・ライブラリの多くは、スレッドを処理する仮想プロセッサが、1 つしか存在していない。したがって、スレッドが並列に実行されることはない。

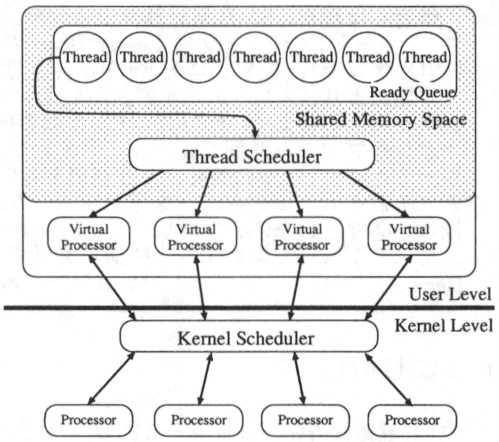

図 1: Lesser Bear のモデル

2.2 Lesser Bear の概要

筆者らは、今までのスレッド・ライブラリの利点を活かしつつ、SMP 型計算機を活用するための方法を提案し、並列に動作し、移植性にも優れたスレッド・ライブラリ Lesser Bear の設計・実現を行っている [6]。

従来のスレッド・ライブラリではスレッドを実行する仮想プロセッサが 1 つであるために、スレッドの並列実行を行うことができなかった。Lesser Bear ではスレッドを並列に動作させるために、複数の UNIX プロセスを仮想プロセッサとして生成する。スレッドの並列実行が可能なライブラリである LinuxThreads[3] や PPL[8] でも同様に、複数の仮想プロセッサを生成している。これにより、同一システム内で、並列に動作する仮想プロセッサが生成できる。しかしこのままでは、すべての仮想プロセッサが任意のスレッドを処理することができない。

またスレッド・ライブラリを実現する上で、

スレッドの中断・再開ができる機能を備えることが必要である。そのためライブラリには、実行中のレジスタ情報などが含まれるスレッド・コンテクストを保存する空間を提供する必要がある。

Lesser Bear では、仮想プロセッサが実行するスレッド・コンテクストを、すべての仮想プロセッサから見える空間に保存する。仮想プロセッサ間でスレッド・コンテクストを共有することで、任意のスレッドが実行できる。

Lesser Bear を利用したアプリケーションがSMP型計算機上で動作することにより、各UNIXプロセスが同時に複数のプロセッサに割当てられ、プロセスが並列に動く。これによってスレッドが並列に実行される。また、すべての仮想プロセッサから見える共有メモリ空間は、スレッド数が動的に変化しても領域の拡張を必要としないだけの大きさを、初期化時にあらかじめ確保しておく。

実現したLesser Bearの機能的な特徴として、次のものがあげられる[6]。

- 移植性
- インターバル割り込みによるコンテクスト切替え
- 標準的なユーザ・インタフェース
- 巨大な共有メモリ領域
- 仮想プロセッサ間での相互排除

以下では特に、移植性と巨大な共有メモリ空間について述べる。

2.2.1 移植性

ライブラリは、特定のアーキテクチャやOSに依存しないことが望ましい。そのためにLesser Bearは、すべてC言語で記述し、かつ、UNIXが標準的に提供する機能のみを用いて実現した。コンテクスト切替えは、`setjmp`関数と`longjmp`関数を用いて行うこととした。

Lesser Bearは、UNIXプロセスの並列実行ができるOSであれば、スレッドを並列に動作できる。また、次節で述べるようなメモリマップト・ファイルの機能を必要とする。

表1: 動作確認した計算機システム

OS	types	feature
SunOS 4.1.4	BSD UNIX	Uni-processor
SunOS 5.5.1	SVR4 UNIX	SMP
SunOS 5.6	SVR4 UNIX	SMP
FreeBSD 2.2.8	BSD UNIX	Uni-processor
FreeBSD 3.0	BSD UNIX	SMP
Linux 2.0	SVR4 UNIX	SMP
IRIX 6.4.1	SVR4 UNIX	SMP

Lesser Bearは現在、表1に記されたOS上での動作を確認している。実現したライブラリでOSに依存せざるをえない部分は、C言語のコードにして2、3行しかない。また、新しいアーキテクチャやオペレーティング・システムに対しても、容易に移植が可能と考えられる。

2.2.2 巨大な共有メモリ空間

各仮想プロセッサがスレッドを任意に実行するために、Lesser Bearでは図1で提案したように、巨大な共有メモリ空間(たとえば1Gバイト程度の共有メモリ空間)を用意し、Ready Queueなどのデータ構造を含め、すべてのスレッド・コンテクストを共有メモリ空間内に保存する。スレッド・コンテクストには、レジスタ情報やスレッドが個別に保持するスタック領域も含まれる。数多くのスレッド・コンテクストを保存するためには、巨大な共有メモリ空間が必要である。Lesser Bearでは、メモリマップト・ファイルを利用して巨大な共有メモリ空間を実現した。

このような方法を採用することで、移植性に優れ、スレッドの並列実行が可能なスレッド・ライブラリLesser Bearは設計・実現されている。

3 スケジューリング機構の設計

本章では、従来のLesser Bear上で起こる問題点について述べ、この問題を解決するスケジューリング機構の提案を行う。

3.1 fork-join 型並列プログラム

fork-join 型とは、図2のようにアプリケーション全体に対して、並列実行と逐次実行を頻繁に繰り返すアプリケーションのタイプである。このようなアプリケーションでは、並列実行部で複数のスレッドやプロセスを利用することにより並列処理が可能となる。

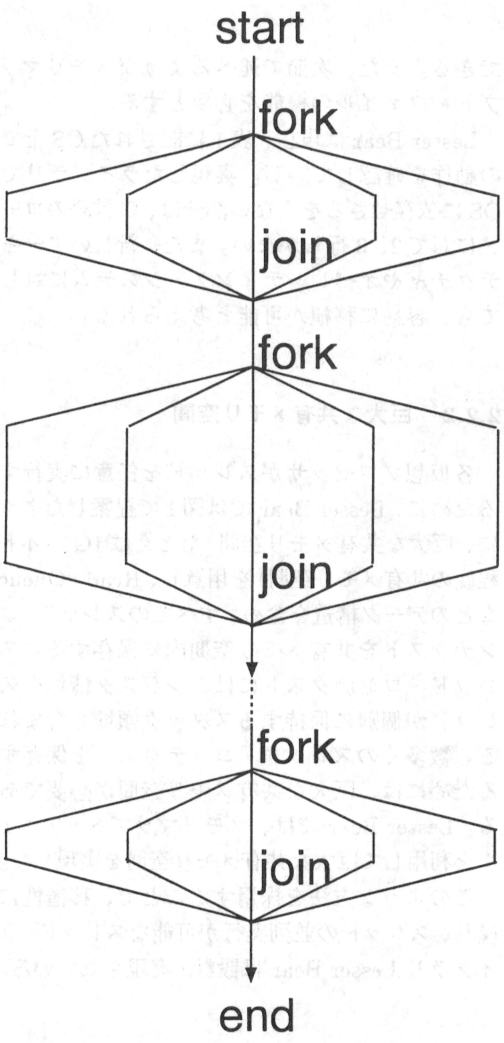

図 2: fork-join 型並列アプリケーションのモデル

fork-join 型並列プログラムでは生成、消滅や同期操作を頻繁に利用するため、実行するアプリケーションに対するオーバヘッドになる。fork-join 操作が頻発するアプリケーションの場合、アプリケーションを実行するシステムによって実行時間が異なる。

従来の Lesser Bear では次節で述べる理由のため、fork-join 操作が頻発するアプリケーションに適したスレッド・ライブラリとは言えない。fork-join 型アプリケーションは代表的な並列アプリケーションであるため、Lesser Bear においてもこのようなを効率よく実行できることが必要とされる。

3.2 スケジューラの逐次化

Lesser Bear では、Ready Queue を含めたすべてのデータ構造が共有メモリ空間に存在するため、それぞれの仮想プロセッサが任意のスレッドを並列に実行できる。そのため従来の Lesser Bear では共有メモリ空間をクリティカル・セクションとして扱い、常に仮想プロセッサ間における排他的な制御を必要としていた。

ある仮想プロセッサがクリティカル・セクションに入りロックを獲得した場合、ロックを獲得しようと試みる他の仮想プロセッサは休眠してしまう。この結果、コンテクスト切替えやスレッドのスケジューリングといったスレッド操作は、仮想プロセッサ間で逐次化をおこす。(図3)。SMP 型計算機におけるこのような現象は、CPU 数が多くなるごとにより頻繁に生じることが考えられる。例えば図4のように、ある仮想プロセッサでスレッドのコンテクスト切替えが生じた場合、他の仮想プロセッサがコンテクスト切替えを試みようとしても仮想プロセッサが休眠状態となってしまう。その結果従来の Lesser Bear では、すべてのスレッド操作が逐次化されてしまう。

このような状態をできるだけ防ぐため、Lesser Bear 内部のデータ構造の改善として以下のような提案が考えられる。

- クリティカル・セクションをデータ構造ごとに分割し、ロック変数をそれぞれのデータ構造ごとに用意する。
- データ構造全体を各仮想プロセッサごとに

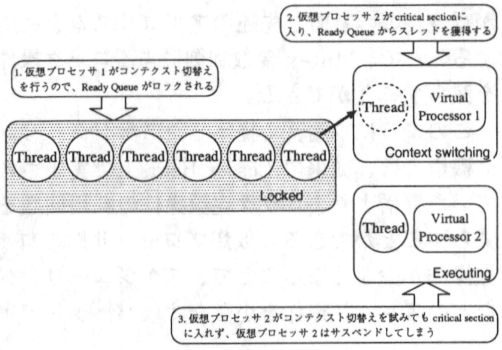

図 3: ブロックされた仮想プロセッサ

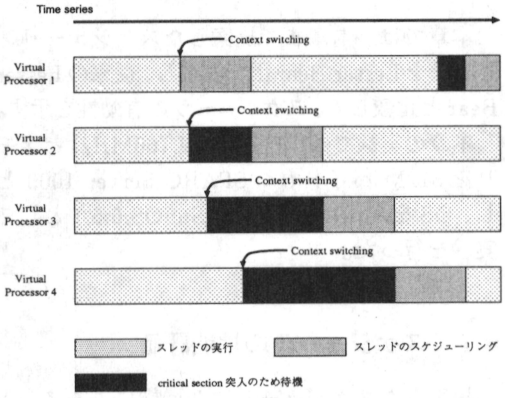

図 4: SMP 型計算機上で仮想プロセッサがブロックされる例

区切り、それぞれを仮想プロセッサに割り当てる。

前者は分割したクリティカル・セクションのために、複数のロック変数を必要とする。そのためライブラリの構造が複雑になりやすい。さらに、コンテクスト切替えや共有メモリ空間の割り当てなどの操作には、複数のロック変数が必要になるため、デッドロックが起きやすくなる。

後者はデータの一貫性を維持するために、仮想プロセッサ間の協調動作が必要になる。しかし、仮想プロセッサ自身が担当する空間に対しては保護に気をつける必要がないため、コンテクスト切替えやスケジューリングなどのスレッド操作にロック操作を必要としない。

本論文では、後者の案を採用する。この方法を実現する際に、すべてのスレッド・コンテクストはライブラリ内に存在する唯一の共有メモリ空間に保存する。それぞれの仮想プロセッサはスレッド・コンテクスト本体ではなく、割り当てられたスレッドの識別子を管理する。アイドル状態の仮想プロセッサは、ほかの仮想プロセッサが保持する実行可能状態のスレッドを獲得する。この間のデータ転送にのみ、仮想プロセッサ間の排他制御を利用する。

3.3 スケジューラの並列化

本論文では 1 つの巨大な共有メモリ空間を各仮想プロセッサごとに区切り、その空間ごとにそれぞれ 2 つのキュー (Protect Queue と Waiver Queue) を用意する。区切られた各空間は、内部に存在する 2 つのキューも含めて、仮想プロセッサごとで管理を行う。図 5 は仮想プロセッサごとで処理を行うスケジューラのモデルである。

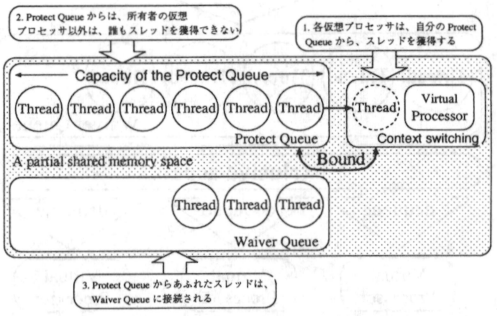

図 5: 新たに提案するスケジューリング機構のモデル

Protect Queue は，所有者の仮想プロセッサ以外からの挿入および削除を許さない。そのため Protect Queue では一切のロック操作を必要としない。スレッドのコンテクスト切替えが生じると、仮想プロセッサは自分自身が所有する Protect Queue から次に実行すべきスレッドを取り出す。仮想プロセッサごとのスルー・プットを均一にするため、それぞれの仮想プロセッサがもつ Protect Queue の容量は均一に設定される。仮想プロセッサが所有するスレッド数が

Protect Queue の容量を越えた場合、あふれたスレッドは Waiver Queue へ挿入される。

Waiver Queue は、所有者の仮想プロセッサからの挿入のみ受け付け、任意の仮想プロセッサからの削除を許す。

Waiver Queue からの削除では、削除を行う仮想プロセッサ間でロック操作を行う。したがって、複数の仮想プロセッサ間で、同時に Waiver Queue からの削除を行うことがない。しかし Waiver Queue への挿入はただ 1 つの仮想プロセッサだけが行う。Waiver Queue への挿入がただ 1 つの仮想プロセッサが行い、Waiver Queue からの削除をただ 1 つの仮想プロセッサが行う場合では、ロック操作が必要ないことが知られている。そのため、Waiver Queue への挿入を行う仮想プロセッサはロック操作を必要としない (図 6)。

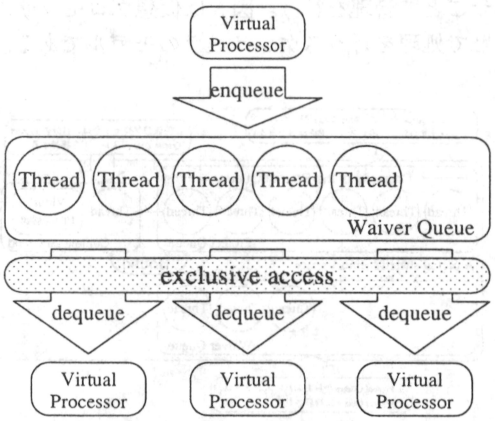

図 6: Waiver Queue への挿入動作と削除動作

仮想プロセッサはキューへの挿入は頻繁に実行するため、本論文のように小さなオーバヘッドで行えることで、システム全体の機能を高めることにつながる。また、Waiver Queue からの削除は、アイドルな仮想プロセッサが実行するため、多少のオーバヘッドがかかってもシステムに大きな影響を与えないと考えられる。

Protect Queue と Waiver Queue のようにキューを分割する手法は、Mutex 変数制御にも応用できる。Mutex 変数の獲得は任意の仮想プロセッサが可能であるが、解放はその時点で Mutex 変数を獲得している仮想プロセッサのみが行う。この特徴と先述のアルゴリズムを利用することで、Mutex 変数制御によるロック操作を減らすことができる。

このように本論文で提案したスケジューリング機構では、従来の Lesser Bear 上で実現されていた機構より、ロック変数操作の利用回数を減らすことができる。仮想プロセッサ間のロック操作が少なくなることで、スケジューリングなどのスレッド操作を小さなオーバヘッドで実現できる。

4 評価

本章では、前章まで提案したスケジューリング機構を Lesser Bear 上で実現し、従来の Lesser Bear と比較して、スケジューラの有効性を示す。

本章の実験では、8 台の CPU(60MHz) とメモリを 512M バイトもつ SPARC Server 1000 とし、OS を SMP に対応した SunOS 5.5.1 として実験を行った。

4.1 スケジューラの性能評価

提案したスケジューリング機構によるスレッド操作の軽量性を評価するために、提案したスケジューリング機構を実現した Lesser Bear と従来の機構を比較した。

まず、スケジューリングに要する時間を計測した。計測を純粋に行うため、ここでは仮想プロセッサ数を 1 に設定する。

表 2: スケジューリングに要するコストの比較

	従来の設計	本論文の設計
スケジューリングに要するコスト (μsec)	144.4	83.2

表 2 は、1 回のスケジューリングに要する時間をスケジューラ機構ごとで計測したものである。この実験では、2 つのスレッドを生成し、コ

ンテクスト切替えを互いに繰り返すアプリケーションを利用した。

この結果から、これまでのスケジューリング機構にはロック操作によるコストが含まれると考えられる。実験を行った環境では、セマフォによるロック操作、アンロック操作ふくめて、およそ65マイクロ秒だけ必要になる。2つとも、ラウンドロビンで動作するため、表2に示す従来のスケジューリング機構による結果は、ちょうどロック機構によるコストが含まれたことになる。

次に各仮想プロセッサ内で、スケジューラの起動回数を調べた。実験では、内部で約10分動作するスレッドを128個生成し、スレッドのタイムスライスを10ミリ秒に設定した。Lesser Bear では仮想プロセッサ間の排他制御にセマフォを利用している。UNIXが標準的に提供するセマフォは、セマフォ・キューに並ぶUNIXプロセスを休眠状態にする。ここでもしOS内に、実行可能状態であるUNIXプロセスが存在しない場合、計算機のCPUはアイドル状態となってしまう。そこで本実験では計算機がもつ各CPUをアイドルにさせないために、仮想プロセッサを実プロセッサ数の2倍である16個生成し実験を行った。

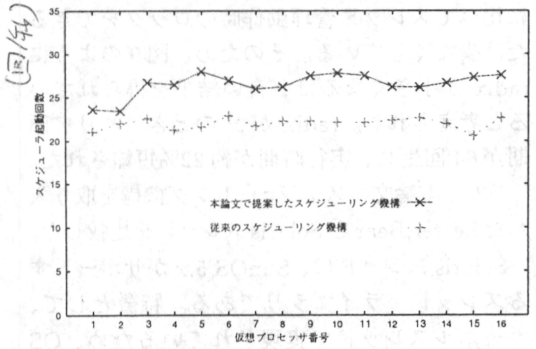

図7: スケジューラ起動回数

図7は横軸を仮想プロセッサ番号、縦軸をスケジューラの起動回数とし、各仮想プロセッサでスケジューラが起動した回数を表している。

図7から、本論文で実現したスケジューリング機構はより頻繁に起動していることが確認できる。この違いは、本論文で実現したスケジューラが仮想プロセッサ間でロックによる競合が少なくなるためであると推測できる。

4.2 アプリケーションによる実行時間比較

本節では、実際のアプリケーションを動作させて、並列化したスケジューリング機構の有効性を示す。本節で利用するアプリケーションとして、並列化したradixソートを利用した。

radixソートとは、各データをn進数で表し、各桁ごとのソートを繰り返すことにより全体のソートを行う方法である。効率化のために2の巾乗をnとすることが多い。各桁は高々n通りしかないので、その各値の個数を数え上げることによりソート後の位置を計算することができる。radixソートでは同じ値であるデータ同士の順序関係は保たれる。データを分散配置しておくことにより、radixソートを並列化することは容易であり、かつ、アルゴリズムとしては、十分なデータサイズがあればCPU台数の増加に対して非常にスケーラブルな性能向上を期待できる。

たとえば、radixを4としたとき、radixソートを並列化するためには、データを各スレッドに分散配置し次の手順でソートを行う(図8)。

1. 各スレッドごとで最下位桁が0、1、2、3である要素数を数える。
2. 1.の結果を集計し、総和を求める。
3. 1.の結果から、それぞれのスレッドが自身のスレッド番号より小さいスレッドまでの部分和を求める。
4. 上の2つの値から各値の転送開始アドレスを求める。
5. 各スレッドが持つデータを、求めた転送開始アドレスへ転送する。

この手順では、総和を求める時点と、1つの手順が終了し桁数を進める時点にすべてのスレッド間で同期操作を必要とする。

Lesser Bearが準拠しているPthreadでは、全スレッド間のバリア同期をサポートしていない。本実験ではMutex変数と条件変数を利用して、

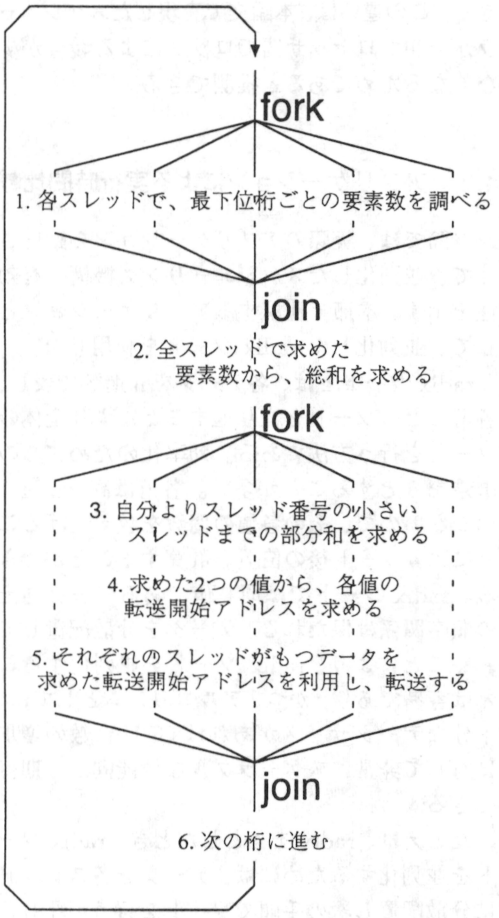

図 8: 並列 radix ソートプログラムのモデル

バリア同期を実現している。

まず Lesser Bear 上で、従来のスケジューリング機構と本論文で提案したスケジューリング機構の性能比較を行った。比較するデータ数を 2^{22}、生成するスレッド数を 2^8 として、radix を 2^1 から 2^8 に変化させた場合の結果を図 9 に示す。図 9 では、従来のスケジューリング機構による実行時間を基準とした相対比を示す。

radix が小さいほど、図 8 の fork 部が小さくなりバリア同期の回数が増える。そのため Mutex 変数制御、条件変数制御などのシステムによるスレッド管理動作の回数が増え、アプリケーション処理以外のオーバヘッドが心配される。特に従来のスケジューリング機構では Mutex 変数制御により、ほかの仮想プロセッサ上でスケジューリングができなくなる可能性もある。

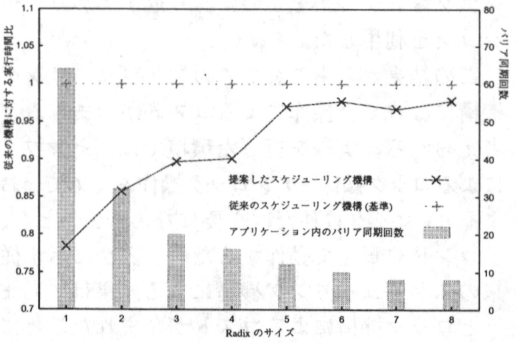

図 9: 従来のスケジューリング機構と新たに提案したスケジューリング機構の実行時間比較 (従来のものを基準とする)

本論文で提案したスケジューリング機構で radix ソートを実行するアプリケーションを動作させた場合、特に radix が小さくなるに従って、従来の機構に比べて実行時間が短縮される。これは、radix が小さくなるに従って、アプリケーションの実行時間に対するシステムのスレッド管理動作の割合が増えるためである。本論文で実現したスケジューリング機構では、これまでの Lesser Bear 上で実現されていた従来の機構に比べてスレッド管理動作時のロックをできるだけ少なくしている。そのため、図 9 のように radix が小さくなるほど良い結果を得られていると考えられる。radix が 2^1 のときにバリア同期が 64 回生じ、実行時間が約 22%短縮された。

次に、本論文のスケジューリング機構を取り入れた Lesser Bear と Solaris スレッドを比較した。

Solaris スレッドは、SunOS 5.x がサポートするスレッド・ライブラリである。特徴として、カーネルスレッドで実現されているため、OS の特徴に適したスレッドライブラリである。

しかし、Solaris スレッドはカーネルスレッドであるため、ライブラリ内で行われるスレッド操作にカーネルが介在してしまう。カーネルの動作は一般にオーバヘッドが大きいため、ライブラリが行うスレッド操作が頻繁に生じるアプリケーションでは、あまり良い結果が期待でき

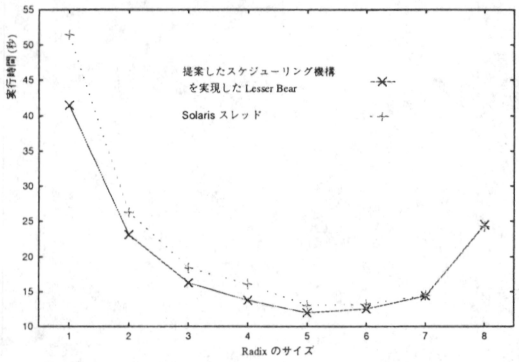

図 10: Solaris スレッドと Lesser Bear の実行時間比較

ない。

図 10 では、先ほどと同じ radix ソートのアプリケーションを利用し、Solaris スレッドを利用した場合と本論文で提案したスケジューリング機構を取り入れた Lesser Bear を利用した場合を比較した。

radix ソートを実行するアプリケーションは先ほどと同様に、データ数を 2^{22}、生成するスレッド数を 2^8 として、radix を 2^1 から 2^8 に変化させた。

radix が小さい時は、図 8 で示した fork-join 処理の回数が増えるため、アプリケーション内の逐次化部分が増えてしまう。逆に radix が大きい時は、図 8 で示した総和を求める処理が大きくなるため、やはり逐次化部分が増えてしまう。図 10 は、このようなアプリケーションがもつ特性から生じた結果である。

図 10 のように radix が小さいくなるに従って、Solaris スレッドよりもよい結果が得られる。radix が小さい場合、Solaris スレッドではスレッド操作を頻繁に利用するため、Lesser Bear に比べて、スレッド操作のオーバヘッドが表れてしまう。一方、radix が大きい場合、オーバヘッドの影響が出にくいため、ほぼ同性能の結果を得ている。

これらの結果より本論文で提案したスケジューリング機構は、仮想プロセッサ間において並列にスケジューリングを可能とし、かつ、小さなオーバヘッドでスケジューリングなどのスレッド操作を可能にできることが確認された。

5 おわりに

本論文では、Lesser Bear 内部で並列にスケジューリングを行い、かつオーバヘッドを小さくしたスケジューリング機構の設計·実現を行った。

スケジューラの並列動作を実現するために本論文では、1 つの共有メモリ空間を各仮想プロセッサごとの空間に区切り、その空間ごとにそれぞれ 2 つのキュー (Protect Queue と Waiver Queue) を用意した。Protect Queue は担当する仮想プロセッサのみが挿入·削除を行うため、ロックを必要としない。Waiver Queue は担当する仮想プロセッサのみが挿入を行い、任意の仮想プロセッサが削除を行う。本論文では、キューへの挿入操作に一切のロック操作を必要としないアルゴリズムを利用する。このアルゴリズムにより、小さなオーバヘッドでスケジューリングを行う機構が実現できた。

実験では、スケジューリングによるオーバヘッドの低減化をしめし、より並列にスケジューラが起動できることを確認した。またアプリケーションには、fork-join 型アプリケーションである、並列 radix ソートを利用した。

並列 radix ソートによる実験の結果、従来の Lesser Bear や Solaris スレッドに比べて、本論文で提案したスケジューリング機構はスレッド操作プリミティブのオーバヘッドがより小さいことを確認した。

現在筆者らは、あらゆるスレッド処理について、できるだけロック操作を必要としない設計を目指し、特にキュー操作に注目し、より小さなオーバヘッドで行うプリミティブの設計を行っている。

参考文献

[1] 安倍 広多, 松浦 敏雄, 谷口 健一：BSD UNIX 上での移植性に優れた軽量プロセス機構の実現, 情報処理学会論文誌, Vol.36, No.2, pp.296–303 (1995).

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska and H. M. Levy : Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 53–79 (1992).

[3] X. Leroy : Linuxthreads – POSIX 1003.1c kernel threads for Linux, `http://pauillac.inria.fr/~xleroy/linuxthreads`.

[4] F. Mueller : A Library Implementation of POSIX Threads under UNIX, *Proceedings of the Winter 1993 USENIX Conference*, pp. 29–41 (1993).

[5] B. Nichols, D. Buttlar and J. P. Farrell : *Pthreads Programming*, O'Reilly & Associates (1996).

[6] 小熊 寿, 海江田 章裕, 森本 浩通, 田村 友彦, 鈴木 貢, 中山 泰一：SMP 型計算機を活用する軽量プロセス・ライブラリ, 情報処理学会論文誌, Vol. 39, No. 9, pp. 2718–2726 (1998).

[7] C. Provenzano : Pthreads version 1_70, `http://www.mit.edu:8001/people/proven/pthreads.html`.

[8] 坂本 力, 宮崎 輝, 桑山 雅行, 最所 圭三, 福田 晃：並列性と移植性をもつユーザレベルスレッドライブラリー PPL の設計および実装, 電子情報通信学会論文誌, Vol. J80–D–I, No. 1, pp. 42-49 (1997).

[9] 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の実現法, 電子情報通信学会論文誌, Vol. J73–D–I, No. 12, pp. 961-970 (1990).