Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

メニーコア混在型並列計算機における スレッド管理方式

長嶺精彦[†1] 坂本龍一[†1] 佐藤未来子[†1] 下沢拓[†2]
吉永一美[†3] 辻田祐一[†3] 堀敦史[†4] 石川裕[†2]
並木美太郎[†1]

本稿では，エクサコンピュータの実現に向けて今後主流となるメニーコアアーキテクチャを備えるシステムを対象とした基盤ソフトウェアについて報告する．本研究では，メニーコア向け OS の軽量なスレッド管理方式と，メニーコア側の OS 内部処理の軽減のために I/O 処理を含む資源管理をマルチコア上の汎用 OS へ代行させる機構について述べる．Intel Corei7 上で試作したスレッド制御ライブラリを用い，スレッド生成・削除のオーバヘッド，マイクロベンチマークによる実行時間の計測を行い，1μs 程度でスレッド生成が可能であること，他のスレッド関連 API も汎用システムより軽量に実行可能であることを確認した．

Thread Management for Many-core and Multi-core architecture

Kiyohiko Nagamine[†1] Ryuichi Sakamoto[†1] Mikiko Sato[†1]
Taku Shimosawa[†2] Kazumi Yoshinaga[†3] Yuichi Tsujita[†3]
Atsushi Hori[†4] Hiroshi Ishikawa[†2] and Mitaro Namiki[†1]

This paper describes the light-weight thread management on many-core processor for the Exa-scale computing. The approaches in this study are (1)user-level thread management, (2)System calls delegation to multi-core system for I/O managment on many-core processor. The light-weight OS "MULiTh" is implemented on a hardware Intel Corei7. The OS's performance shows that pthread_create/join execute one micro second. The other POSIX Thread APIs of the OS can be also executed with light-weight as compared with the thread library on the Linux.

1. はじめに

近年，プロセスの微細化による単位面積あたりのコア数増加を背景として，スーパーコンピュータの性能は着実に向上している．TOP500 に掲載される上位システムの性能はペタフロップスを達成しており，エクサフロップス達成のための技術としてメニーコアプロセッサが注目されている．Intel 社は，Intel Architecture の汎用コアを搭載したメニーコアアーキテクチャ「Many Integrated Core(MIC)」[1] の商用化を発表し，NVIDIA 社も従来の GPGPU アーキテクチャに汎用コアを統合し，Streaming Multiprocessor(SM)数を増大することによりエクサフロップスを目指している [2].

本研究では，エクサコンピュータの実現に向けて，今後主流となるメニーコアアーキテクチャを備えるシステムを対象とし,基盤ソフトウェアの研究開発を行っている．従来のマルチコアプロセッサ(以下，マルチコア)と，Intel 社 MIC のような大量の汎用コアにより並列演算性能を追求するメニーコアプロセッサ(以下，メニーコア)を搭載する「メニーコア混在型並列計算機アーキテクチャ」を対象とし，アプリケーションプログラムから二種類のプロセッサを使い分けるプログラム実行基盤を提供することで，メニーコアとマルチコアの特性を活かした単一システムを実現することを目指している．並列演算における主要なオーバヘッドとなる I/O バウンドな処理や，複雑な制御系のコードはマルチコアで実行し，並列性能を追求する処理はメニーコアで実行することで，データインテンシブな大規模シミュレーションや，I/O 要求の頻度が高い高並列サーバなどのアプリケーションを効率的に処理する．すなわち，アプリケーションプログラムの特性によってシステム側でメニーコアとマルチコアを使い分けられるシステムを目標としている．このようなシステムのために重要となるのは，(1)メニーコア向けの軽量 OS における軽量なスレッド実行基盤，(2)マルチコア上の汎用 OS および軽量 OS が連携して行う I/O 処理の機構である．

本稿では，メニーコア向けの軽量 OS（以下，メニーコア OS）における軽量なスレッド実行基盤と，マルチコア上の汎用 OS およびメニーコア OS が連携して行う I/O 処理の機構について述べる．メニーコア OS では，ユーザレベルでメニーコア上のコア管理を行う専用のスレッドライブラリを用いることで，メニーコア上での軽量な実行基盤を実現する．また，メニーコア OS は可能な限り OS 内の処理を軽減するために I/O 管理を行わず，スレッドの実行管理に専念する．スレッドから要求される I/O 処理

†1 東京農工大学
Tokyo University of Agriculture and Technology
†2 東京大学
University of Tokyo
†3 近畿大学
Kinki University
†4 理化学研究所計算機科学研究機構
RIKEN Advanced Institute for Computational Science

Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

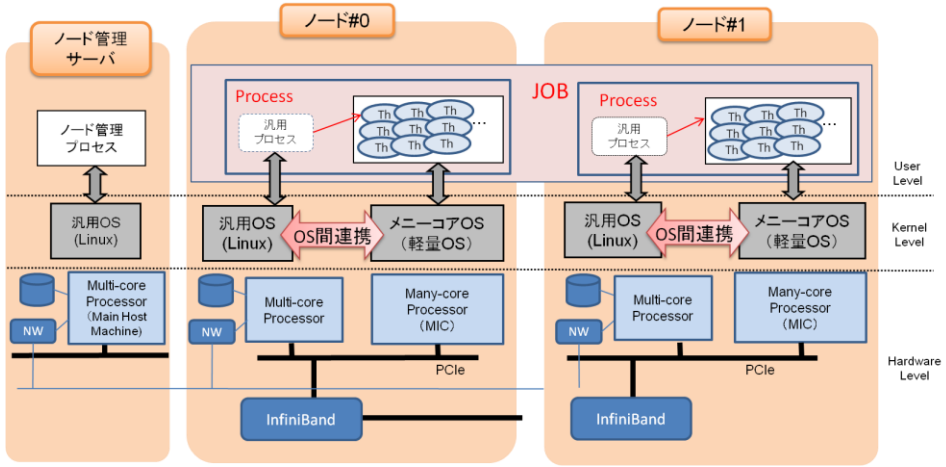

図 1 メニーコア混在型並列計算機アーキテクチャとプロセスモデル

に関しては，マルチコア上の汎用 OS に I/O 処理を代行させるための機構である「システムコールデリゲーション」を実現し，マルチコア上で I/O を処理する．本研究は，メニーコア OS において，極力軽量なスレッド管理を追求することで，システム全体として，次世代のエクサコンピュータに適用可能な並列プログラムの実行基盤を目指している．

本稿の構成は以下のとおりである，まず第 2 章において本研究で想定するシステム構成について述べる．第 3 章では，本研究で想定するメニーコア混在型並列計算機で定義するプロセスモデルについて述べる．第 4 章では，メニーコア OS 上で動作するスレッドの管理方式の設計について述べる．第 5 章では，メニーコア OS におけるスレッドライブラリの実装について述べる．第 6 章では，試作したスレッドライブラリの評価結果を示す．第 7 章において関連研究について述べ，第 8 章でまとめる．

2. システム構成

本章では，本研究で想定するシステムアーキテクチャについて示す．

2.1 ハードウェア構成

図 1 に，本研究で想定するメニーコア混在型並列計算機アーキテクチャを示す．一つのノードは汎用計算機としてマルチコア，並列演算専用計算機としてメニーコアが PCI 等のバスで接続されたシステムである．各ノード間を Infiniband 等の高速インタフェースで接続し，各ノードには高速なストレージデバイス，ネットワークインタフェースを備える．さらに，システム上の全ノードを管理するノード管理サーバを配置する．

マルチコアおよびメニーコアではそれぞれにメモリを備え，バスを介してお互いのメモリをアクセスする．また，本研究におけるメニーコアは，Intel 社の MIC のような数十単位の汎用コアを備えたホモジニアスなプロセッサを想定している．特徴として，従来のマルチコアプロセッサと比べてコアの単体性能は低く，コアごとに備えるキャッシュメモリやメインメモリの容量も少量であるが，コア間のキャッシュ共有が可能であり，コア数を増量して並列性能を強化している点があげられる．

2.2 OS アーキテクチャ

本研究では，汎用 OS と軽量 OS のハイブリッドな OS 構成とし，実行するプログラムの特性に応じてメニーコア混在型並列計算機に備えるマルチコアとメニーコアを使い分ける．具体的には，メニーコア OS は並列演算を担うスレッドの管理・制御に専念し，スレッドから発行される I/O 要求は，システムコールデリゲーションを通じて汎用 OS が代行して処理する．これにより，システム全体をユーザに対して単一システムとしてみせ，軽量な実行基盤を提供する．以下では，本 OS アーキテクチャの主な構成要素である汎用 OS，メニーコア OS，およびシステムコールデリゲーションの基盤となる OS 間連携機構の概要について示す．

2.2.1 汎用 OS

汎用 OS は，Linux 等の従来の汎用 OS と同様に，マルチコア上で実行するプロセス，メモリ資源，ファイルシステムやディスク・ネットワークなどの I/O 資源を管理する．また，メニーコアで実行するプログラムの実行開始．停止等の制御も担っている．

2.2.2 メニーコア OS

メニーコア OS は，メニーコア上で実行するプロセスやスレッド，メモリ資源を管理する．これらの管理機構を軽量にすることで並列演算に特化した実行基盤を提供する．メニーコア OS は汎用 OS からの指示に基づいてメニーコア向けプログラムを実行する．メニーコア OS におけるプロセスモデルに関しては，3 章において詳述する．また，メニーコアにおける軽量なコア資源管理を，4 章で述べる専用のスレッドライブラリで行う．

2.2.3 OS 間連携機構

本研究では，既存研究の異種 OS 間連携機構 [3]を利用し，汎用 OS とメニーコア OS の OS 間で，以下に示す機能を用いて，アプリケーションプログラムの実行制御を分担する．

- 汎用 OS からメニーコア OS の起動

Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

- メニーコア OS 用プログラムのロードおよび実行
- メニーコア OS 用プログラムにおけるシステムコールデリゲーション

「システムコールデリゲーション」とは，メニーコア OS で実行中のプログラム中で実行する UNIX API やノード間通信の API などを，メニーコア OS 内部でフックし，汎用 OS 側で処理を代行するための OS 間連携機構である．これにより，汎用 OS が管理する I/O 資源やネットワーク通信をメニーコア OS 上のアプリケーションプログラムから利用可能とする．システムコールデリゲーションの詳細については 4.3.3 節で述べる．

3. プロセスモデル

本研究で想定するメニーコア混在型並列計算機において，実行するアプリケーションプログラムに相当する単位を JOB と定義する．1 つの JOB は複数の Process と呼ぶ実行単位から構成される．各 Process は図 2 で示したノード 1 つに対して 1 つ割り当てる．すなわち，実行に必要なノード数を N とすると，JOB=Process×N となる．1 つのアプリケーションは複数ノードで処理を分散する．またノードに割り当てられる Process は，汎用 OS 側の汎用プロセスと，メニーコア OS 上のプロセスやスレッドといった実行実体を構成要素としている．汎用プロセスは，メニーコア上で実行するプログラムの制御を行い，メニーコア上に実行プログラムをロードすることでメニーコア上の実行実体を生成する．

4. スレッド管理方式

本研究のスレッド管理方式は，管理機構を単純化するために(1)軽量なスレッドライブラリによるスレッド制御，(2)システムコールデリゲーションによる I/O 処理の代行の 2 点について追求することで軽量な実行基盤の提供を目指している．本章では，この 2 点を実現するためのメニーコア上で動作するスレッド管理方式について述べる．

以下，4.1 節でメニーコア上での並列性能を実現するスレッドモデルおよびその設計方針について述べ，4.2 節でメニーコア OS において提供する API について述べ，4.3 節でスレッド管理の設計について述べる．

4.1 設計方針

本研究ではスレッド管理の軽量化へのアプローチとして，スレッド管理機構を極力単純化するという方針をとる． この方針に適したスレッドモデルとして，ユーザレベルスレッドモデルを採用する．ユーザレベルからカーネルレベルへ遷移を必要とせず，関数呼び出し程度のオーバヘッドでスレッド制御が実現できる点や，ユーザが扱うアプリケーションの特性に応じて，スケジューリングポリシを変更可能な点が上記の方針に適している．

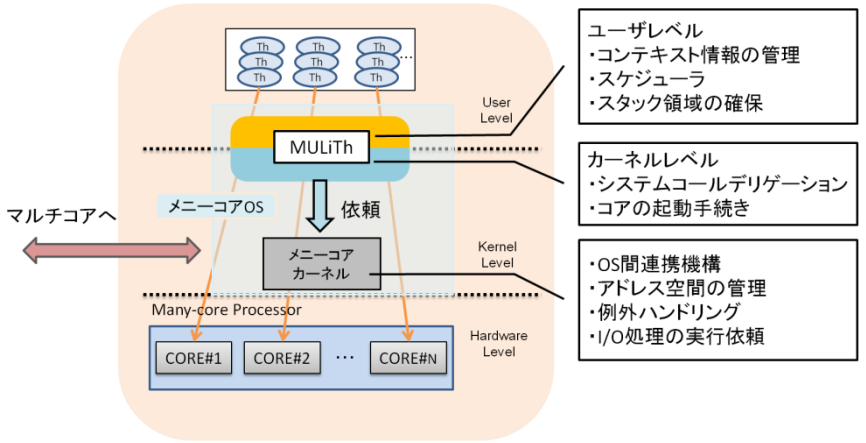

図 2 MULiTh とメニーコアカーネルの機能分担

図 2 にメニーコア OS の構成と機能を示す．本研究では過去に他アーキテクチャ向けに開発されたスレッドライブラリ MULiTh[4](Userlevel Thread Library for Multithreaded Architecture)と，カーネルレベルの「メニーコアカーネル」を搭載して，スレッドの実行制御を行う．MULiTh の特長として，(1)ユーザレベルスレッドモデルでの軽量なスレッド管理が可能なこと，(2)プログラミングインタフェースは POSIX Thread(PThread)に準拠しており，スレッド制御は既存のソフトウェアとソースコードレベルでの互換性を提供していることが挙げられる．MULiTh では，スレッドのコンテキスト情報の管理，制御，スタックの割り当てなどをユーザレベルで行い，マルチコア OS へのシステムコールデリゲーションや，コア起動手続きなどをカーネルレベルで行う．一方，メニーコアカーネルは，主にアドレス空間の管理，OS 間通信機構，例外ハンドリングなどを担う．

4.2 メニーコア OS 上のスレッドへ提供する API

本節では，メニーコア OS 上のスレッドに対して提供するスレッド関連 API と Linux 標準 API について述べる．

4.2.1 スレッド関連プログラミングインタフェース仕様

MULiTh が提供する Pthread インタフェースの一覧を表 1 に示す．Pthread インタフェースは， スレッドライブラリの規格として標準的なものであり，実行アプリケーショ

Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

表 1 MULiTh が提供する PThread I/F

分類名	関数名
スレッド管理系	pthread_create
	pthread_exit
	pthread_join
	pthread_attr_init
	pthread_attr_setdetachstate
	pthread_attr_getdetachstate
	pthread_yield
同期系	pthread_mutex_lock
	pthread_mutex_unlock
	pthread_mutex_init
	pthread_cond_wait
	pthread_cond_signal
	pthread_cond_broadcast
	pthread_cond_init
ローカルメモリ系	pthread_key_create
	pthread_key_delete
	pthread_key_getspecific
	pthread_key_setspecific

ンとのソースレベルでの互換性を提供する．このため，MULiTh 内部のスレッド管理を軽量にすることで，ユーザに対するプログラミングの容易さを損なわず，軽量な実行基盤を提供することが可能である． スレッド管理の設計に関しては 4.3 節で詳細を述べる．

4.2.2 Linux 標準プログラミングインタフェース仕様

本研究において，メニーコア OS 全体としては Linux 標準 API を可能な限りサポートする方針をとるが，詳細は検討中である． Linux API のうち，スレッドから利用する主な API は，read/write，socket などのファイル/デバイス，ネットワーク通信の I/O アクセスに関連する API とする．シグナルについては検討中である．I/O 関連の API の処理はシステムコールデリゲーションを利用して汎用 OS に処理の代行を依頼する．システムコールデリゲーションは，メニーコア OS 上のスレッドが発行するシステムコールをフックし，汎用 OS が管理する I/O 資源やネットワークのアクセスを実現する機構としてスレッドに提供する．この機構はカーネルレベルでは 2.2 節で述べた OS 間連携機構を利用する．システムコールデリゲーションの詳細は次節で述べる．

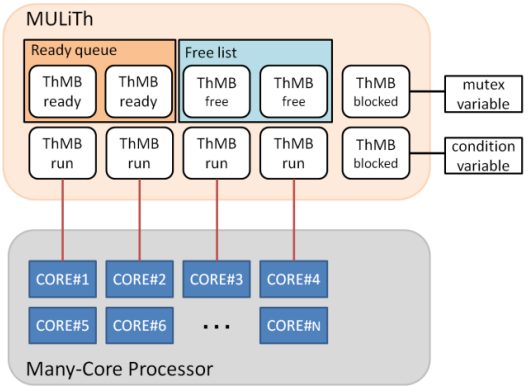

図 3 MULiTh によるスレッド管理の概念

4.3 スレッド管理の設計

本節では，メニーコアにおけるスレッド管理の設計について述べる．前述したように本研究では，スレッド管理の機構をできる限り単純化する方針としている．このため，MULiTh で管理するコアをあらかじめカーネルレベルで起動しておき，すべてユーザレベルでスレッドの管理・制御を行えるようにする．また，スレッド管理機構の複雑化につながる I/O 要求に対しては，システムコールデリゲーションを利用して汎用 OS に代行処理させる．以下では，これらの設計の詳細について述べる．

4.3.1 スレッド情報の管理

MULiTh は，図 3 に示すように生成されたスレッドの情報および必要なメモリ領域の管理を行う．スレッドの情報は，スレッド管理ブロック(ThMB)と呼ぶデータ領域中に存在する．スレッド管理ブロックは，当該スレッドのコンテキスト情報・属性・生成されたスレッドの状態を保持する．スレッドの属性とは，スレッドがデタッチされているかなどの，現在のスレッドの状況を示す．スレッドの状態は実行可能状態・待ち状態・ブロック状態のいずれであるかを示す．未使用の ThMB はフリーリストに登録され，実行可能状態となった ThMB はレディキューに接続する．これらの情報を保持するスレッド管理ブロックは，ユーザ空間上に配置される．また，スレッドに割り当てるスタック領域は MULiTh がユーザ空間上に静的に確保しておき，スレッド生成時に割り当てることで，スタック領域の割り当ての高速な実行が可能となる．

4.3.2 スレッド割り当ての手順

MULiTh は実行可能なスレッドを格納するレディキューを 1 つ持ち，キューのスケジューリングポリシに従ってスレッドの実行順序を決定する．管理機構の軽量化のた

Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

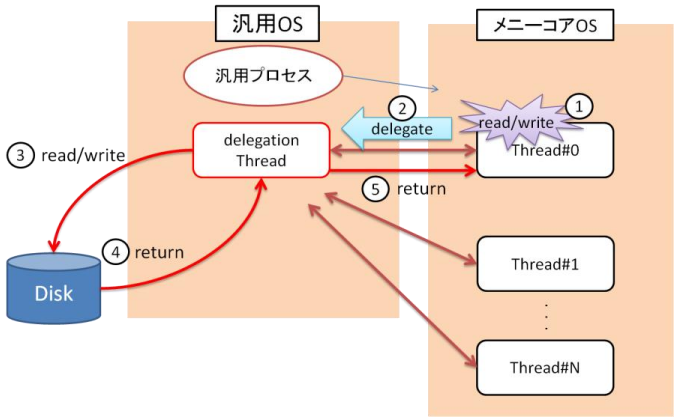

図 4 システムコールデリゲーションの概念図

め，スケジューリングポリシとしては FIFO を採用しているが，ユーザレベルで制御しているため，ユーザの必要に応じてスケジューリングポリシを変更することが可能である．スレッド割り当ては，レディキューから実行可能なスレッドを取り出し，このときに空いているコアが存在すれば，生成したスレッドをただちに割り当てることで，高速なスレッド実行を実現することが可能である．

本研究では，スレッド割り当ての軽量性を重視し，以下のようにコアの起動およびスレッドの割り当てを行う．MULiTh は，初期化時に指定した数だけコアを起動し，各コアにはポーリングを実行するダミースレッドを割り当てる．コアへのスレッド割り当てはスレッド生成時のフラグセンスにより実現する．省電力のために，pause 命令を用いて停止する．

4.3.3 システムコールデリゲーション

本節では，デリゲーションの概念および実行手順について述べる．システムコールデリゲーションは，メニーコア OS から発行される I/O 要求を汎用 OS が代行して処理を行う機構である．I/O 処理を代行することにより，メニーコア OS 上でのスレッド管理機構の軽量化を図ることが目的である．

メニーコア OS 上にスレッドを生成するとき，システムコールの代行処理を行うデリゲーションスレッドを汎用 OS 上に生成する．生成されたデリゲーションスレッドは，メニーコア OS 側のスレッドからの I/O 要求などを待ち，要求が発生した場合には，必要な処理を行い，結果を当該スレッドに返す． 生成するデリゲーションスレッド数に関しては， 一つのデリゲーションスレッドで全ての代行処理を担うと，I/O 要求が集中した際に性能ボトルネックとなる可能性がある．反対にメニーコア上の各スレッドに一対一で対応する形態は，通信機構の管理の複雑化につながる．このため，発生する I/O 要求の頻度に対して適切な数で対応する必要があると考えられるが，設計としては生成するデリゲーションスレッド数を固定せずに，ユーザが必要に応じて任意に設定するような方式を検討している．図 4 にシステムコールデリゲーションの概要を示す．次に，システムコールデリゲーションで必要となる情報，および具体的手順について述べる．

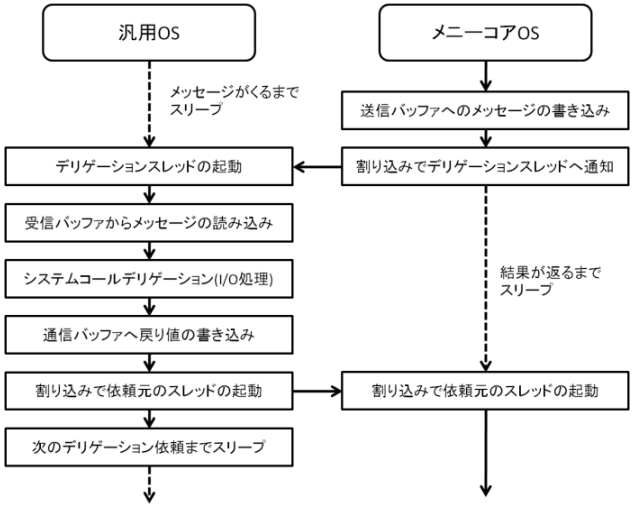

図 5 システムコールデリゲーションの実行手順

(1) 通信メッセージ

システムコールデリゲーション時にデリゲーションスレッドへ渡す情報は，以下の 3 種類である．

（a） メッセージタイプ
メッセージの種類を示す．これにより，代行処理する I/O 資源の判別を行う．

（b） 論理スレッド番号
MULiTh が管理するメニーコア OS 上の論理スレッド番号を示す．どのスレッドから依頼された処理であるかを判別する．

（c） システムコールのパラメータ
代行する I/O 処理に必要な引数などの情報である．

これらの情報はメッセージ形式で保持し，各 OS が持つ専用の通信バッファ領域に書き込み/読み込みを行うことで情報を受け渡す．

Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

(2) システムコールデリゲーションの実行手順

図 5 に示すシステムコールデリゲーションのシーケンスに関して，まずデリゲーションスレッドは，I/O 処理の依頼があるまで処理を行う必要がないため，メニーコア OS からの通知があるまでスリープする．メニーコア OS 上のスレッドから I/O 要求が発生すると，まず OS 間通信用のバッファ領域にメッセージを書く．次に，依頼を待つデリゲーションスレッドに対して，割り込みを利用してデリゲーションスレッドを起動する．依頼を発行したスレッドは結果が返るまでスリープする．起動したデリゲーションスレッドは，OS 間通信用バッファからメッセージを読み取り，メッセージの内容に応じて I/O 処理を行う．結果を再度 OS 間通信用バッファに書き込み，依頼元のスレッド論理番号に基づき，処理終了通知を行う．最後に依頼元スレッドは OS 間通信用バッファから結果を受け取り，処理を継続する．

4.3.4 スレッドに対する I/O ブロック

スレッドからの I/O アクセスに関しては，前述したシステムコールデリゲーションを用いる．このときに発生する，スレッドに対する I/O ブロックの方式には，以下の 2 通りの方法がある．

1. スレッドをブロックしてシステムコールデリゲーションを行う．I/O 処理の結果が戻ってくるまでコアにはスレッドを割り当てず，稼働しない．
2. スレッドをブロックせずにシステムコールデリゲーションを行う．システムコールデリゲーションの発行後，実行可能なスレッドがレディキューにあれば，実行する．

本研究では，まず前者の方式で実装を行うが，扱うスレッド数やキューの形態による I/O 処理性能を評価した上で今後も検討を続ける．

5. 実装

x86 マルチコアプロセッサのネイティブ環境において，64bit 対応のメニーコア OS の開発を進めており，これまでに基本的な例外処理，コアの起動および MULiTh の実装を行った．ただし，これらは全てカーネルレベルでの実装である．以下では，スレッド管理の実装について述べる．

5.1 コア管理

MULiTh 内部の処理では，各コアにスレッドを割り当てるため，コア番号によるコアの識別が必要となる．コア番号には，x86 アーキテクチャで各コアが備えた Local APIC と呼ばれる割り込みコントローラの ID を利用するが，起動するコアの順序は保証されない．MULiTh 内部では，コア番号 0 を基準として複数のコアを扱うため，起動したコア番号を論理化する必要がある．このため，コア ID を物理コア ID と論理コア ID に分け，両者の対応をとるコア ID 管理テーブルを用意する．MULiTh からは論理コア ID を利用することで MULiTh のスレッド管理機構との整合をとる．

5.2 コア間のメモリ共有

MULiTh では，スレッド管理のための情報をコア間で共有している．主に共有している情報として，(1)コア ID 管理テーブルなどのコア管理に関連する情報，(2)各コアで使用中のスタックおよびコンテキスト情報，(3)レディキューやフリーリストなどがある．今回の実装では，カーネル空間を 4MB のページサイズで管理し，アドレス変換情報をあらかじめ設定しておくことにより，ページフォルトによるメモリ管理の擾乱が発生しないようにしている．なお，本ページ管理情報の設定については，コア起動時に各コアにおいてシステム関連レジスタへ同一のアドレス変換情報を設定することで実現した．

5.3 排他制御の実装

MULiTh 内部には，レディキューへのアクセスなど，クリティカルセクションが存在する．このため，各クリティカルセクションに応じたロック変数と，ロック変数に対するアトミック操作が必要となる．アトミック操作は tas 命令を利用してスピンロックによる実装を行った．また，スピンロックはメモリアクセスが頻発し，他のスレッドの実行を阻害する可能性がある．このため，x86 アーキテクチャが用意する pause 命令を利用し，ロック獲得に失敗した場合は，スレッドの実行を一時停止するよう実装を行った．これにより，他のスレッドへの影響を抑えることが期待できる．ただし，pause 命令によるスピンロックでは CPU バスアクセスの競合を引き起こすため，今後は monitor/mwait 命令を用いたスピンロックの実装を検討している．

6. 試作システムの評価および考察

本研究で提案したスレッド管理方式がどの程度並列性能の向上を実現できるか確認するため，表 2 に示す環境で試作した MULiTh の主なスレッド関数のオーバヘッドの実行時間を計測した．また，コア数に応じた性能向上率を確かめるため，マイクロベンチマークとして，平均画素法による画像縮小プログラムの実行時間を計測し，評価を行った．結果を表 3 および図 6 に示す．また，比較データとして，Linux(2.6.32) の PThread(NPTL)上で同様の性能評価を行った結果を示す．スレッド関数の評価について，pthread_create/pthread_join はスレッドの生成・実行・削除の全てを同一のコアで行う場合と，生成時に空いているコアへ割り当てた場合の実行時間を示す．また，mutex_lock / unlock は，mutex のロック・解除を一万回繰り返したときの実行時間を示す．

Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

表 2 評価環境

項目	仕様
CPU コア	Intel Corei7 CPU 980
システムバス周波数	3.33GHz
キャッシュ	L2: 256KB, L3: 12MB

表 3 基本性能の計測結果(単位: μ sec)

計測項目	MULiTh	Linux
pthread_attr_init	0.072	0.143
pthread_attr_setdetachstate	0.077	0.142
pthread_create / join (同一コア)	1.004	19.42
(他コア)	0.792	10.52
pthread_mutex_lock / unlock : 1core	0.13	0.36
2core	186.23	357.42
4core	583.44	1024.27
6core	745.55	1424.88

表 3 から，評価を行った全てのスレッド関数で，MULiTh はよりオーバヘッドの少ないスレッド制御が可能である結果を示した．特に，pthread_create/ join では，スレッド生成時に空いているコアへただちにスレッドを割り当てることで，同一コアで全ての処理を行う場合と比べて高速な実行結果を得た．MULiTh では，コアの起動や一部の処理を除き，ライブラリ内部で直接スレッド制御を行うため，カーネルを介する汎用 OS(Linux)のスレッド制御と比較して軽量になっていると考えられる．したがって，基本的に同一のプログラム実行時は NPTL と比較して同等またはより高速な並列演算が可能である．

図 6 のベンチマーク測定結果は，本稿で示した軽量なスレッド管理機構全体の性能を示している．前章までに示した設計および実装における検討課題はいくつかあるが，コア数による性能向上率に関して MULiTh は NPTL と同等の性能結果を示した．

ただし，実装上メモリオーバヘッド・システムオーバヘッドのない状況での評価であるため，今後はメモリ管理方式について検討を進め，ユーザレベルでの動作および性能評価を行う．また，コア数に対するスケーラビリティを確かめるためには今回の評価環境では十分でないため，より実行時コア数を増加させた環境での性能評価を行う必要があると考えている．

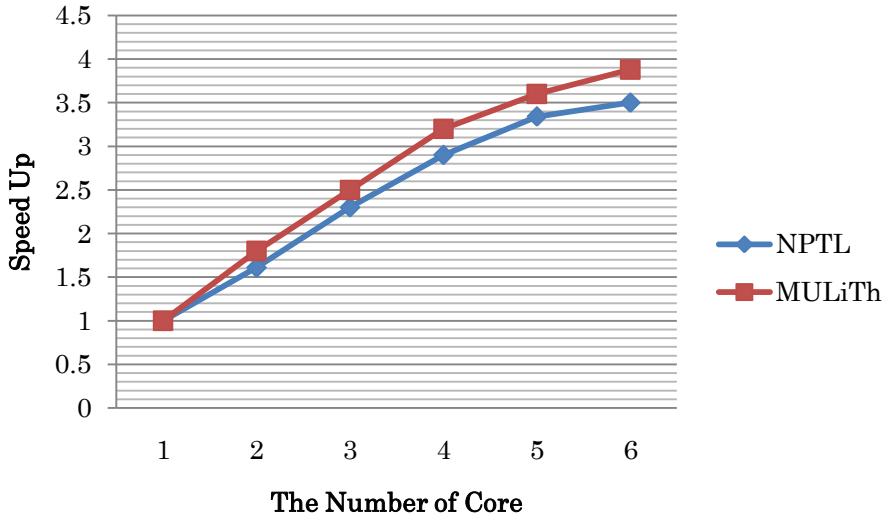

図 6 ベンチマーク実行時間の性能向上率

7. 関連研究

本研究のスレッド管理方式は，メニーコア OS 上での軽量な実行基盤を提供するスレッドライブラリに加え，汎用 OS による I/O 要求の代行によりスレッド管理機構の単純化を図っている．この 2 つの観点から既存研究と本研究の差分について述べる．

軽量なスレッド管理を用いた並列性能の追求に関しては，研究が盛んに行われている．Cilk[5]，OpenMP task[6]，QThread[7]，などは，並列性能を最大限発揮するためにコア数分のプロセスを生成し，各プロセス内部でユーザレベルのスレッド管理を行う．この設計方針は本研究と同じくするところである．しかしながら，これらのスレッド管理方式では，スレッドの I/O ブロックに対する．1 つの計算機内で処理が完結するため，I/O ブロックが発生した場合，コアの 1 つが I/O 処理のために占有され，並列性能のボトルネックとなる恐れがある．また，専用のプログラミングインタフェースを用いてプログラムを記述するため，プログラミングに対して習熟する必要がある．

二つ目の I/O 処理を効率的に行う方式としては，epoll などの I/O 処理を待つための API とノンブロッキング I/O を併用し， I/O の多重化をユーザレベルで行う方式があげられる．Cappriccio[8]，GNU Pth[9]などは，これらの手法を用いて I/O 処理の効率化を

Vol.2011-ARC-197 No.30
Vol.2011-HPC-132 No.30
2011/11/29

図っている．しかしながら，これらのスレッド管理方式は I/O 処理の高効率化のみを着目しており，複数のコアを効率的に利用してスレッドを実行することができない．

また，これら軽量なスレッド管理および高効率な I/O 処理の両立を目的とするスレッド管理方式として Massive Threads[10]が提案されている．複数の計算ノードにまたがってスレッドを管理・実行することを想定し，並列計算全体の性能を高めるような設計がなされている．しかしながら，他ノードへの動的な負荷分散のためにスレッドのマイグレーションなどが発生するため，複雑なスレッド管理が必要となり，管理機構を極力単純にして軽量化を狙う本研究とは設計方針が異なる．

8. まとめ

本稿では，メニーコア向けの OS について，軽量なスレッド制御を実現するためのスレッド管理方式について提案を行った．本研究におけるスレッド管理方式は，メニーコアカーネルと機能分担して，専用のスレッドライブラリがスレッドを管理することで，軽量な実行基盤を提供することを目標としている．さらに，汎用 OS に対してシステムコールデリゲーションによる代行依頼を発行することで，I/O 処理による複雑なスレッド管理を排除し，さらなる単純化を目指している．実際に，試作したスレッドライブラリに対して，スレッド制御関数のオーバヘッドおよびマイクロベンチマークの実行時間を計測したところ，既存のスレッドライブラリと比較して同等またはより軽量なスレッド制御が可能であることを確認した．

今後の課題として，まず本稿で示した設計についての課題を検討する．特に，I/O 処理に関わるスレッド管理の軽量化について重要となるシステムコールデリゲーションの設計および実装を行い，システム全体として並列性能の評価を行う．また，メニーコア OS 上のメモリ管理方式について，設計の検討を進め，メニーコア OS 上でプロセスを複数生成し，各プロセスで複数のコアの確保および MULiTh を用いたマルチスレッド環境の実現を目指す．

謝辞 本研究を進めるにあたり，東京大学情報基盤センター長の石川裕教授から有益なご意見をいただいた．なお，本研究は，科学技術振興機構「JST」の戦略的創造研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」によるものである．

参考文献

1) Kirk B. Skaugen: HPC Technology –Scale-Up & Scale-Out, ISC'10 Keynote(on-line), available from <http://lecture2go.uni-hamburg.de/konferenzen/-/k/10940>, (2010.05).
2) Dally, B.: GPU Computing To Exascale and Beyond (online), available from <http://www.nvidia.com/content/PDF/sc_2010/theater/Dally_SC10.pdf>, (accessed 2011-10-10).
3) 磯部泰徳，佐藤未来子，並木美太郎: ホモジニアスマルチコア CPU における異種 OS 間の連携機構の試作，情報処理学会「システムソフトウェアとオペレーティング・システム」第 111 回研究報告, Vol.2009-OS-111, No.17, pp.1-8 (2009,04)
4) 佐藤未来子，磯部泰徳，十山圭介，野尻徹，入江直彦，内山邦男，並木美太郎: 汎用ホモジニアスコアプロセッサにおける OS とスレッドライブラリ，情報処理学会第 20 回コンピュータシステムシンポジウム，ポスターセッション (2008)
5) Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Rnadall, K.H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, SIGPLAN Not., Vol.30, No.8, pp.207-216 (1995)
6) Ayguade, E., Copty, N., Duran, A., Hoeflinger, J.,Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P. and Zhang, G.: A Proposal for Task Parallelism in OpenMP, IWOMP '07: Proc. erd international workshop on OpenMP, pp.1-12, Berlin, Heidelberg, Springer-Verlag (2008).
7) Wheeler, K.B., Murphy,R.C. and Thain, D.: Qthreads : An API for programming with millions of lightweight threads, IPDPS, IEEE, pp.1-8 (2008).
8) von Behren, R., Condit, J., Zhou, F., Necula, G.C. and Brewer, E.: Capriccio: scalable threads for internet services, SOSP '03: Proc. 19th ACM symposium on Operating systems principles, New York, NY, USA, pp.268-281, ACM (2003).
9) Engelschall, R.S.: GNU Pth — the GNU Portable Threads. http://www.gnu.org/software/pth/
10) 中島潤，田浦健次朗: 高効率な I/O と軽量性を両立させるマルチスレッド処理系 情報処理学会論文誌 プログラミング Vol. 4 No. 1 13-26 (2011)