# 高効率な I/O と軽量性を両立するマルチスレッド処理系

中島 潤[†1] 田浦 健次朗[†1]

今後のコンピュータは，現行のものと比較してより多くの CPU コアと，より高速なネットワークを備える方向に発展していくことが予想されている．そのため，次世代のコンピュータハードウェアを活用するためには，複数の CPU コア間にスレッドを適切に割り振って並列実行することと，I/O 処理を効率的に行うことの両方に留意してソフトウェアを設計することが要求される．既存のマルチスレッド処理系にはこの 2 つの要求のうち片方を満たしているものは存在しているが，これを両立した処理系は存在していない．

本稿では，次世代コンピュータ上で高性能なアプリケーションを記述する基盤ソフトウェアとして，この 2 つの要求に加え，既存のアプリケーションに対して，再コンパイルや移植の必要なしに適用することができる，という特徴をもつマルチスレッド処理系 MassiveThreads を提案する．MassiveThreads は，上述の要求を部分的に満たしている既存の処理系において用いられている実装手法を積極的に採用することによって，スレッドの管理の軽量化，マルチコア環境におけるスケーラビリティ，I/O 処理の効率化の 3 つの要件を達成する．評価の結果，小さなオーバーヘッドで多くのスレッドを管理でき，複数の CPU コア間の負荷分散を良好にとれること，OS 標準のマルチスレッド処理系よりも効率的に I/O の処理ができることを確認した．

# Multithread Framework that Manages both Efficient I/O and Lightweight Thread Management

Jun Nakashima[†1] and Kenjiro Taura[†1]

It is estimated that next-generation computer hardware has more CPU cores and faster network. Thus in designing software, both handling I/O efficiently and scalable execution of multiple threads are essential to utilize future computer hardware. Some existing multithread frameworks meet a part of the requirements, but none of them meets both.

In this paper, we propose MassiveThreads as a middleware for writing high-performance applications in next-generation hardware. MassiveThreads not only meets two requirements described above, but also has an interface to apply it to existing software without recompile or porting to improve the performance. MassiveThreads uses many techniques which existing multithread frameworks use and achieves lightweight thread management, scalability in multicore environment, and efficient I/O handling. Microbenchmarks confirmed that MassiveThreads can manage many threads with little overhead, balance loads well in multicore environment, and handle I/O more efficiently than multithread framework provided by the operating system.

## 1. はじめに

### 1.1 次世代コンピュータの性能を引き出すための課題

本節では，コンピュータの進歩の方向性について行われている予測と，それに伴って生じる，コンピュータの性能を引き出すためにソフトウェアが解決しなければならない課題について説明する．

#### 1.1.1 ノードあたりの CPU コア数，スレッド数の増大

近年のコンピュータは，複数の CPU コアを搭載し，それらを並列動作させることで性能を向上させている．今後もその傾向は進み，コンピュータにはますます多くの CPU コアが実装されるようになると考えられる．

さらに，次世代のコンピュータにおいては，1 つの CPU コアに 10〜100 個程度のスレッドを割り振り，細粒度にスレッドを切り替えることで，メモリやネットワークのレイテンシを隠蔽する手法が用いられると予測されている[1)]．

したがって，計算ノード単体の性能を引き出すために，ソフトウェアは多くのスレッドを軽量に管理し，複数の CPU コアに分散させて並列実行する必要がある．

#### 1.1.2 並列計算機のノード数の増加，I/O の高速化

近年の大規模並列計算機のほとんどは，多くの計算ノードを高速ネットワークで接続する構成になっている．例えば，理研が開発中の次世代スパコンにおいては，8 万台のノードをネットワークで接続し，全体で 10 ペタフロップスの性能を達成することが予定されている．これらのノード間の協調はネットワーク I/O を介して行われるため，ノード間協調のオーバーヘッドを最低限に抑えるためには I/O を効率的に処理することが必要である．さらに，今後 10 年間で，ネットワークの転送レートは数桁増加するという予測が発表されている[1)]．

そのため，I/O を効率的に処理することの重要性は今後ますます高まっていくと考えられ

†1 東京大学
The University of Tokyo

る．ソフトウェアはI/Oを処理する部分がボトルネックにならないよう，I/Oの効率化についても配慮して設計する必要がある．

### 1.2 本研究の目標

1.1で示される状況を踏まえ，我々は，次世代のコンピュータハードウェア上で高性能なアプリケーションを記述するためのソフトウェア基盤として，以下に示す3つの要求を満足するマルチスレッド処理系，MassiveThreadsを提案する．

(1) 多くのスレッドを軽量に管理できること．さらに，小さいオーバーヘッドで大量のスレッドを作成・削除できること
(2) 大量のスレッドからのI/Oを効率的に実行できること
(3) 既存のアプリケーションに対して，再コンパイルや移植の必要なく適用できるインターフェースを備えること

3番目の要件は1.1で示される状況とは直接関係しないが，これによって，既存のアプリケーションや処理系に対して容易にMassiveThreadsを適用し，軽量なスレッドや効率的なI/O処理のメリットを受けることが可能になる．

本研究の目標は，MassiveThreadsの実装，評価を通じて，次世代のコンピュータハードウェアを活用することのできるソフトウェア基盤を構築することである．

### 1.3 想定するアプリケーション

MassiveThreadsが対象とするアプリケーションは，主に以下のいずれかの要求をもつものである．

(1) 多くのスレッドを軽量に管理する必要のあるアプリケーション
(2) 多くのスレッドからのI/Oを処理する必要があるアプリケーション

これらに該当する具体的なアプリケーションとして，以下のようなものが考えられる．

#### 1.3.1 次世代並列言語

X10[2]やChapel[3]といった，分散並列アプリケーションを高生産に記述することを目標としている言語は，並列化の最も基本的な単位として動的に軽量スレッドを作成し，非同期にコード片を実行する機能を定義している．

プログラムは複数ノードにまたがって実行されるため，これらの言語のランタイムには，スレッドを軽量かつ動的に作成する機能だけでなく，I/O経由で行われる，異なる計算ノードに存在するデータへのアクセスを効率的に処理することも求められる．これに対して本処理系を組み込むことで，軽量なスレッド管理やI/Oの効率性といった特徴を活用することができる．

#### 1.3.2 Data-intensive computing

Data-intensive computingとは，分散配置されているデータをネットワーク経由で取得し，それをもとに大規模シミュレーションなどの計算を行う処理である．

データが分散配置されておりネットワークI/Oによるデータ取得のレイテンシが大きいため，効率的にネットワークI/Oを処理することと，多くのスレッドを作成して並列処理を行い，レイテンシを隠蔽することの両立が性能向上には欠かせない．本処理系を基盤ソフトウェアとして用いることにより，これらは自動的に達成されるため，開発者はアプリケーションそのものの記述に注力することが可能になる．

#### 1.3.3 高並列サーバ

多くのクライアントからのネットワークI/Oを効率的に処理することが必要とされる，ファイルサーバのようなアプリケーションは，本処理系を利用することで1つの接続に1つのスレッドを割り振る形で簡潔に記述でき，しかも高いスループットを達成できる．既存のサーバアプリケーションについても，本処理系を適用することで再コンパイルや移植を行うことなくスループットを向上させることができる．

### 1.4 本稿の構成

2節では，既存の軽量なスレッド管理を実現している処理系や，I/Oを効率的に処理できる処理系について述べ，それらとMassiveThreadsの目標について比較を行う．3節では，MassiveThreadsの実装について述べる．4節では，MassiveThreadsの軽量性とI/Oの効率性を確かめるためのマイクロベンチマークと，その結果についての考察を述べる．最後に5節で，本稿の結論および今後の課題について述べる．

## 2. 関 連 研 究

### 2.1 軽量なスレッド管理を実現している処理系

Cilk[4]，StackThreads/MP[5]，OpenMP task[6]，Intel Thread Building Blocks[7]などといった処理系は，開発者に並列性を最大限に発揮するのに十分なだけ多くのスレッド[⋆1]を作成するようにプログラムを記述させ，それを処理系が適切にCPUコアに割り振って並列実行する，という思想で設計されてきた処理系である．

これらの処理系は，CPUコア数分だけOSレベルのスレッドを起動し，これらがユーザー

⋆1 各々の処理系においてはスレッドという呼称はなされず，タスクなどと呼ばれていることが多いが，ここでは「並列に実行される処理の流れ」というセマンティクスをもつものを一括してスレッドと呼称する．

レベルのスレッドの管理を行う．ユーザーレベルでのスレッド管理はすべてメモリ操作に帰着するので，操作にシステムコールが必要な OS レベルのスレッドと比較して軽量なスレッド管理を実現することができる．

しかし，これらの実装はスレッドが I/O 処理などでブロックすることを想定していない．また，既存のプログラムに適用できるようなインターフェースを備えていないため，これらの処理系を利用するためにはそれ専用のソースコードを記述する必要がある．

### 2.2 効率的な I/O 処理を実現している処理系

多くの接続の I/O を効率的に処理するために，epoll などの I/O 待ちのための API とノンブロッキング I/O を用い，ユーザーレベルで I/O を多重化することが広く行われている．Capriccio[8], GNU Pth[9] , StateThreads[10] などといった処理系は，この手法を利用することで，大量のスレッドから発行された I/O の効率的な処理を実現したマルチスレッド処理系である．

これらの処理系は同期 I/O の処理をフックして内部的にはノンブロッキング I/O を利用して実行し，もしブロックするならば他のスレッドに制御を移す．さらに定期的に epoll などで I/O 待ちを行い，I/O が可能になった段階でブロックしたスレッドを再び実行可能にする．これによってユーザーには複数スレッドを用いて同期 I/O を行うようなインターフェースを提供しながら，実際にはユーザーレベルで I/O を多重化し，効率的な処理を実現することができる．

しかし，これらの処理系は 2.1 とは逆に I/O を行う効率のみに着目しており，複数 CPU を利用することができない．また，Capriccio や GNU Pth は OS レベルのマルチスレッド処理系と同じインターフェースを備えているが，処理系を利用するためにはアプリケーションの再コンパイルが必要である．

### 2.3 研究目標との比較

本研究が目標としている特徴と，既存の処理系がそれらのうちの何を実現しているのかを表にまとめたものを**表 1** に示す．

表の各列に略記されたマルチスレッド処理系の特徴の意味は以下のようになる．

- LIGHT. . . 大量のスレッドを軽量に管理できる
- IO. . . I/O を効率的に行うことができる
- PTHREAD. . . OS 標準のスレッド処理系と同様のインターフェースをもち，既存のプログラムに再コンパイルなしに適用できる

表の各行は処理系の名称を，列はその項目が実現できているかどうかを表している．比較対象として，Cilk，Capriccio，NPTL(Linux の OS 標準のマルチスレッド処理系) についても併記している．

**表 1** 各マルチスレッド処理系の満たす特徴
Table 1 Features of each multithread framework

| | LIGHT | IO | PTHREAD |
|---|---|---|---|
| MassiveThreads | ○ | ○ | ○ |
| Cilk | ○ | × | × |
| Capriccio | × | ○ | × |
| NPTL | × | × | ○ |

## 3. 設計と実装

MassiveThreads の実装の最終的な目的は，1.2 で挙げた，軽量なスレッド管理，効率的な I/O 処理，既存の処理系との互換性という 3 つの特徴を満たすことである．

各々の特徴は，関連研究として挙げた処理系によって達成されているので，スレッドのスケジューリングや負荷分散といった，スレッド管理に関わる部分は 2.1 に挙げた処理系を，ブロッキング I/O の取り扱いなどの効率的な I/O を実現するための部分は 2.2 に挙げた処理系を参考にして実装を行なった．

以下に，MassiveThreads の具体的な設計と実装について説明する．

### 3.1 処理系の構成要素

本処理系は，ワーカースレッド，スケジューラ，ユーザースレッドの 3 つの要素から構成される (**図 1**)．以下に，それらの意味と役割，それらの関係性について説明する．

#### 3.1.1 ワーカースレッド

ワーカースレッドは各 CPU コアにただ 1 つ存在する OS レベルのスレッドである．各ワーカースレッドはスケジューラと実行可能なユーザースレッドを格納する deque [⋆1]であるランキューをもち，スケジューラかユーザースレッドのどちらかを実行している．

#### 3.1.2 スケジューラ

スケジューラはランキューの中にユーザースレッドが存在しない場合にのみ実行される，ユーザーレベルのスレッドである．スケジューラはワークスチールや I/O のチェックを行い，実行可能なユーザースレッドを獲得したらそのユーザースレッドに制御を移す．

⋆1 先頭と末尾の両方から挿入および取りだしができるキュー

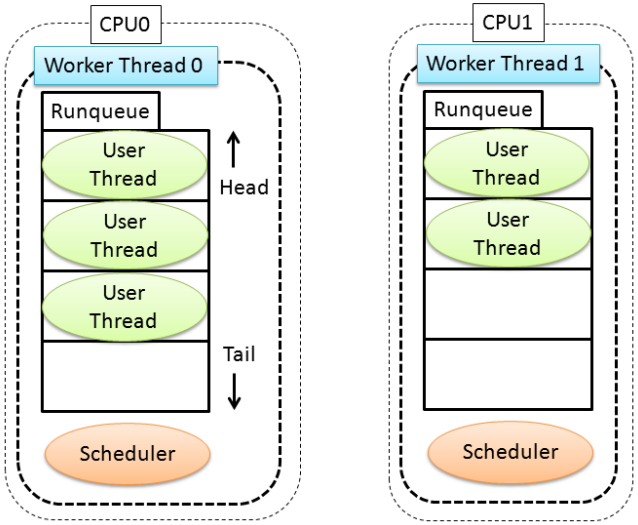


**図 1** 処理系の構成要素とそれらの関係
Fig. 1 Organization of MassiveThreads

#### 3.1.3 ユーザースレッド

ユーザースレッドはスケジュールの対象となる，ユーザーレベルのスレッドである．ユーザが「スレッド」として利用できるものはこのユーザースレッドに該当する．説明を簡単にするため，今後は「スレッド」と表記した場合ユーザースレッドを指すものとする．

### 3.2 スレッドの管理

スレッドのスケジューリングには，Work-First スケジューリング[11] とワークスチールを組み合わせた戦略を用いた．

#### 3.2.1 Work-First スケジューリング

Work-First スケジューリングとは，以下の 3 つの方針にしたがって，スレッドの実行順を決定するスケジューリング手法である．

(1) スレッドの実行は，ランキューの先頭にあるものを優先して行う．
(2) スレッドを作成した直後，現在のスレッドを停止させてランキューの先頭に追加し，新しく作成したスレッドに制御を移す．
(3) スレッドが自発的に制御を譲った場合は，そのスレッドをランキューの末尾に追加する．

#### 3.2.2 ワークスチール

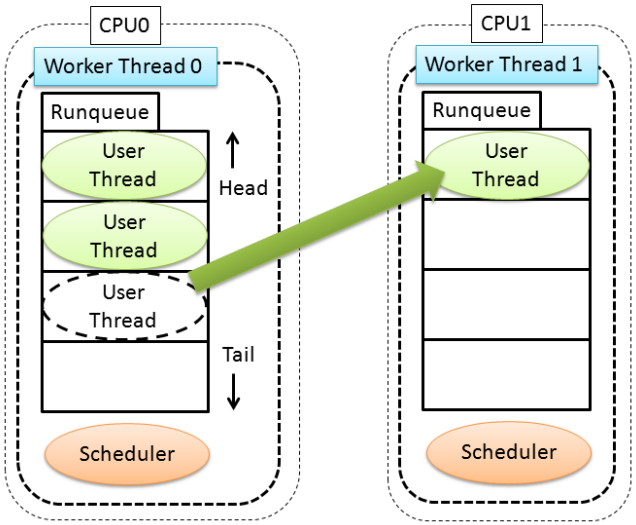


**図 2** ワークスチール
Fig. 2 Work stealing

Work-First スケジューリングにおいては，明示的にスレッドを他のワーカースレッドに割り振るような処理は行わない．それを補い CPU コア間の負荷分散をとるため，実行可能なスレッドをもたないワーカースレッドは，他のワーカースレッドをランダムに選択し，そのランキューの末尾から実行可能なスレッドを取り出して実行する (**図 2**)．もし対象のワーカースレッドが実行可能なスレッドを所有していないのであれば，ワークスチールが成功するまで他のワーカースレッドをランダムに選択して繰り返す．

#### 3.2.3 ランキューの実装

ランキューは，THE Protocol[12] を利用した deque を実装した．これは，Cilk や Java Fork/Join のランキューとしても用いられている，

THE Protocol は，以下のような特徴をもっている．

(1) 先頭に追加する操作 (push)，先頭から取りだす操作 (pop)，および末尾に追加する操作 (put) は，1 つのワーカースレッドしか行うことができない．

(2) 末尾から取りだす操作 (take) のみは，全ワーカースレッドが行うことができる．

(3) push は排他制御なしに行うことができる．pop も deque がほとんど空である場合を除いて，排他制御なしで行われる．

Work-First スケジューリングにおいては，take はワークスチーリングの際に，put はスレッドが自発的に制御を譲る際にしか発生しないのに対して，push や pop はスレッドの作成・削除のたびに呼び出されるので，その頻度は非常に大きくなる．したがって，その排他制御を省略することは処理系のオーバーヘッドを低減するのに有効である．

#### 3.2.4 コンテキストスイッチの実装

コンテキストスイッチのための関数は，レジスタの退避・復元を行うコードをアセンブラで記述することで実装した．さらにスレッド処理系の実装に特化した機能として，コンテキストを切り替えた後，切り替わった先のコンテキストに制御を移す前に，切り替わる前のコンテキストで指定した任意の関数を実行することのできる API を実装した．これによって「自スレッドを何らかのデータ構造に格納して待機状態にし，他のスレッドに制御を移す」操作を高速化することができる．

具体例として，スレッドが他のスレッドに制御を移す際の際の処理を説明する．この処理はランキューの末尾に自スレッドを追加し，ランキューの先頭にあるスレッドにコンテキストを切り替える操作であるが，ここで問題になるのはどのタイミングで自スレッドをランキューに入れるかである．一見すると切り替えの直前に追加を行えばいいように思えるが，ランキューへの追加が終わった後，自スレッドがコンテキストの切り替えを完了させる前にワークスチールが発生し，他のワーカースレッドで自スレッドが実行されるとコンテキストが破綻してしまう．

それを回避するためには，コンテキストが完全に切り替わってからスレッドをランキューに追加すればよい．単純なコンテキストスイッチのみを用いてこれを実現するためには，一度スケジューラにコンテキストを切り替え，それまで実行していたスレッドをランキューの末尾に追加したのち，ランキューの先頭からスレッドを取り出し，それに切り替えを行う，という 2 回のコンテキストスイッチが必要になる．

しかし，前述の API を利用し，コンテキストが切り替わったあとに，引数で指定されたスレッドをランキューに追加する関数を呼び出し，その引数に自スレッドを指定することによって 1 回のコンテキストスイッチでこれを実現できる．

#### 3.2.5 メモリ確保・解放の高速化

スレッドの作成・削除の際にはスレッドを管理する構造体や，スタック領域のメモリが確保・解放される．毎回 OS のメモリアロケータを呼び出してメモリの確保・解放を行うのはオーバーヘッドが大きいため，スレッドが破棄され，動的に確保されたメモリ領域が不要になった際には，それをプールしておき，後でメモリが必要になったときに再利用するものとした．

さらに，スレッドが所有する領域の空間的局所性を高め，複数スレッドが頻繁に切り替わる際のキャッシュヒット率を高めるため，OS のメモリアロケータを呼ぶ必要が生じた際には，複数スレッド分を一度に確保し，すぐに必要となる 1 スレッド以外の資源についてはフリーリストに格納しておくこととした．

### 3.3 ブロッキング I/O の処理

#### 3.3.1 構　　成

3.1 に加えて，ブロッキング I/O をフックするために MassiveThreads に追加する I/O 処理部の構成図を**図 3** に示す．

各ワーカースレッドは I/O 待ちを行うための，epoll インスタンスをもち，アイドル時に I/O のポーリングを行う．さらに，各ファイルディスクリプタには I/O を待っているスレッドと，そのスレッドが行おうとした I/O 操作のリスト (I/O 待ちリスト) を対応付ける．ファイルディスクリプタと I/O 待ちリストのタプルは連想配列に格納され，ファイルディスクリプタをキーとして I/O 待ちリストを取得することができるようになっている．

#### 3.3.2 ファイルディスクリプタ作成時の処理

ファイルディスクリプタを作成する関数をフックし，ファイルディスクリプタが作成された際にはそれを fcntl 関数によりノンブロッキングに設定し，ワーカースレッドの epoll インスタンスに登録する．さらに，ファイルディスクリプタに対応した I/O 待ちリストを作成し，連想配列に登録する．

#### 3.3.3 同期 I/O 呼び出しのフック

ファイルディスクリプタが作成された時点でノンブロッキングに設定されているので，単に I/O 呼び出しを発行するだけでノンブロッキング I/O として実行される．呼び出しがブロックした場合はファイルディスクリプタに対応した I/O 待ちリストに自スレッドを登録して他のスレッドに制御を移す (図 3(1))．

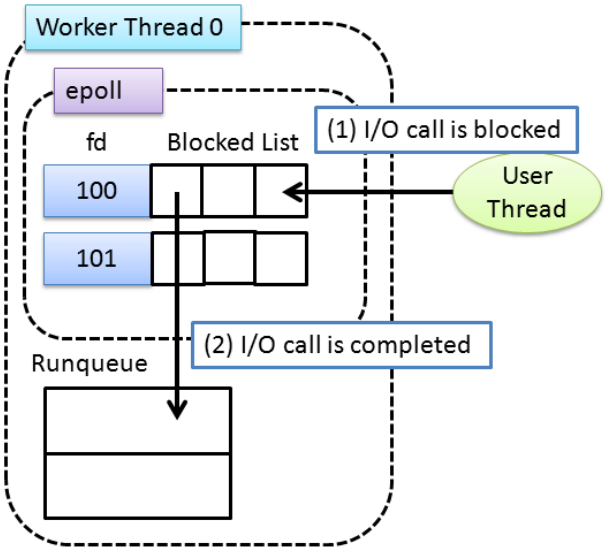


**図 3** I/O の処理部分の構成
Fig. 3 Organization of handling I/O part

#### 3.3.4 I/O のポーリング

スケジューラは実行可能なスレッドがなくなった際，epoll_wait によって，所有するファイルディスクリプタの I/O が可能であるかどうかを確認し，可能ならば待ちリストに登録されている I/O 操作を行う．もし I/O 操作が完了したならばそのスレッドを自分のワーカースレッドのランキューに追加する (図 3(2))．

#### 3.3.5 オーバーヘッドの削減

待ちリストに I/O のポーリングを行う段階で I/O 操作の完了確認まで行う目的は，スレッド操作のオーバーヘッドを最低限に抑えるためである．

1 つのファイルディスクリプタの待ちリストに多くのスレッドが登録されていた場合，待ちリストに登録されているすべてのスレッドをランキューに入れたとしても，ほとんどのスレッドが再び I/O 待ちでブロックし，待ちリストに戻ってきてしまう可能性が高い．しかし，I/O が完了したスレッドのみを再開するようにすれば，ランキューに入れたスレッドがまたすぐブロックしてしまう事態を避けることができる．

### 3.4 既存のアプリケーションとのインターフェース

Linux においては，LD_PRELOAD 環境変数に共有ライブラリのファイル名を指定することで，アプリケーションの実行前に共有ライブラリを読み込ませ，共有ライブラリから呼び出している関数をフックすることが可能である．これを利用して，本処理系のインターフェースとして OS 標準のスレッド処理系と同様のものを準備することで，既存のアプリケーションに対して，再コンパイルを行わずに本処理系を適用できるようにした．

## 4. 性能評価と考察

### 4.1 軽量性の評価

3 で説明した処理系のスレッド管理の軽量性を評価するため，マイクロベンチマークによって軽量なスレッド管理を実現する処理系である Cilk，Intel Threading Building Blocks，OpenMP Task との性能比較を行った．

コンパイラには GCC(4.4.1) を，評価に用いるプラットフォームには Opteron 2380(2.5GHz)4 コア 8 ソケットからなる，32 コアの計算機を用いた．

ベンチマークとしてはフィボナッチ数列の計算による処理系のオーバーヘッドの評価と，UTS Benchmark を用いた負荷分散の能力の評価の 2 種類を行った．

#### 4.1.1 オーバーヘッドの評価

オーバーヘッドの評価は，以下のような手順によってフィボナッチ数を計算することによって行った．

(1) 各スレッドは，開始時に引数として計算する項数 $n$ を受け取る．$n$ が 0 か 1 なら 0 を返して即座に終了する．
(2) fib(n) を計算するスレッドは fib(n) と fib(n-1) を計算するスレッドを作成する．
(3) さきほど作成した 2 つのスレッドが終了するのを待ち，完了したらそれらの結果の和を戻り値として返す．

大量のスレッドについて作成・削除が繰り返されるため，実行時間の大半を処理系のオーバーヘッドが占める．

フィボナッチ数列の第 40 項目を計算させた際の実行性能を**図 4** に示す．横軸は使用したコア数，縦軸は処理系のオーバーヘッドを，Cilk で記述した同じ内容のプログラムを実行した際のオーバーヘッドで正規化したものである．実装したスレッド処理系のオーバーヘッドは，最も大きな場合でも Cilk の 2.2 倍程度で，これは Intel TBB と比較しても小さい．

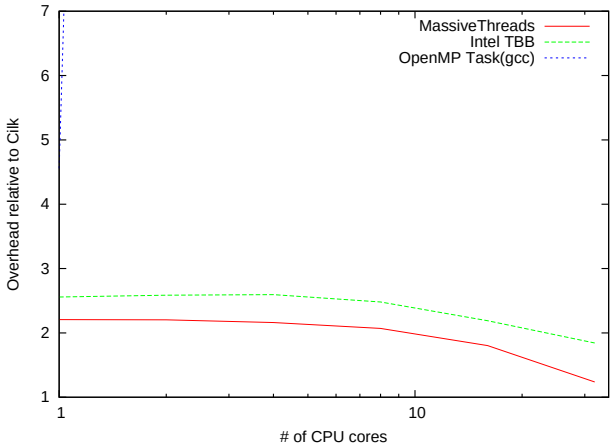


**図 4** fib(40) を計算した際のオーバーヘッド
Fig. 4 Overhead of fib(40) calculation

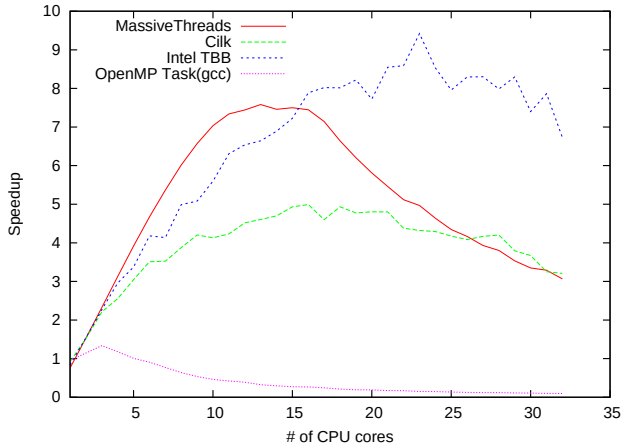


**図 5** UTS Benchmark の結果
Fig. 5 The result of UTS Benchmark

また，OpenMP Task のオーバーヘッドは他の処理系と比較して極めて大きかった．

Cilk と比較して最大 2.2 倍，というとオーバーヘッドが大きいように思えるかもしれないが，実際の実行時間に占めるオーバーヘッドは 23.3 秒で，この間に作成・削除されるスレッド数は 3.3 億個ほどにもなるため，1 スレッドあたりの作成・削除時間に換算すると約 71 ナノ秒となる．OS 標準のマルチスレッド処理系では，1 スレッドの作成・削除に 10 マイクロ秒単位の時間がかかることを考えると，このオーバーヘッドは十分に小さいといえる．

Cilk と比較してオーバーヘッドが大きくなった理由としては，Cilk は再帰的な並列処理に特化した言語仕様[⋆1]となっており，処理系もそれを活用して終了待ちの際に必要な同期を最低限に抑えているのに対して，MassiveThreads はより一般的な用途を想定して設計されているので，終了待ちの際の同期と，それに伴うオーバーヘッドが増加しているためであると考えている．

#### 4.1.2 負荷分散能力の評価

負荷分散能力の評価には，UTS Benchmark[13] を用いた．これは，一定の規則にしたがって生成される，不均一な木構造を探索し，そのノード数の合計を計算する際の性能を測定するベンチマークである．木の構造が不均一であるため，CPU コア数を増やした際の性能向上率は負荷分散能力に左右される．

木は以下のような規則で生成される．

(1) 木の各ノードは 20 バイトのディスクリプタをもつ
(2) 各ノードは確率 $q$ で，$m$ 個の子ノードをもつ．
(3) 子ノードのディスクリプタには，親ノードのディスクリプタと子のインデックスを結合したものをハッシュ関数にかけて得られた 20 バイトを用いる．

パラメータとしては，UTS Benchmark に添付されている T3 データセットを用いた．このデータセットのパラメータは $q = 0.124875$，$m = 8$ で，木の最大深さは 1572，ノード数は約 411 万である．

結果を**図 5** に示す．横軸は使用した CPU コア数，縦軸は逐次版と比較した際の性能向上率である．MassiveThreads は Cilk と比較してオーバーヘッドが大きいため，コア数が少ない場合には Cilk と比べて低い実行性能であるが，コア数が 4 を超えると Cilk の実行性能を上回るようになる．しかし，CPU コア数が 16 を超えると急激に性能が低下してしまう．この性能低下は Cilk にも同様にみられている．

Intel TBB はコア数が 16 個程度までは MassiveThreads と比べて低い実行性能だが，CPU

⋆1 例えば，作成されたスレッドが親の終了を待つような操作は Cilk では行えない．

コア数が多い領域でも性能低下が小さくなっている．また，OpenMP Task は CPU コア数が増えるとわずかに性能が向上するもののすぐ減少に転じ，ほとんど並列処理の効果が出ていない．

現段階の MassiveThreads と Cilk のスケジューリング手法はそれほど差があるわけではないため，MassiveThreads と Cilk との性能差が生じる原因はメモリバリアやロックといった，細かい実装の差異に起因するものであると推測される．

MassiveThreads と Intel TBB との性能差の原因はまだ判明していない．Intel TBB の実装がなぜ多くの CPU コアを性能低下なく利用できるか突き止め，それを実装の参考にすることは，MassiveThreads 内の性能低下の原因となっている部分を改善し，より多くの CPU コアを利用するために有用であると考えられる．

### 4.2 I/O 性能の評価

MassiveThreads の I/O 処理の効率性を評価するために，2 台の計算機を用いて以下のような手順で ping-pong ベンチマークを行い，OS 標準のスレッド処理系と性能を比較した．

(1) サーバ-クライアント間で複数の接続を確立する
(2) クライアントは各接続について 1 バイトのメッセージを送信し，サーバはそれを受信して同一の返事を返す
(3) (2) を 1 回のトランザクションとし，そのスループットを測定する

測定条件としては，

(1) 全部の接続が絶えずメッセージを送受信する場合，
(2) 同時にメッセージを送受信できる接続が全体の 8 分の 1 である場合
(3) 同時にメッセージを送受信できる接続が 128 個で固定されている場合

の 3 通りを用いた．(1) は計算ノードどうしが高速なネットワークで接続されている状況を，(2)，(3) はバンド幅が小さく，実際に通信を行えるスレッド数が限定されている状況を想定したものである．

上記の処理を OS 標準のマルチスレッド処理系を用い，接続 1 つにつき 1 つのスレッドを割り振る形で記述したものと，それに LD_PRELOAD を利用し，MassiveThreads を適用したものの 2 種類についてベンチマークを行ない，そのスループットを比較した．ただし，同時通信数の制限数は，クライアント側のスレッドの数を制限し，1 つのスレッドに複数の接続を割り振てラウンドロビンに送受信を行うことによって実現している．

コンパイラには GCC(4.4.1) を，評価に用いるプラットフォームには Xeon E5410(2.33GHz)4 コア 2 ソケットからなる，8 コアの計算ノード 2 台を用いた，計算ノードはスイッチを経由し，10Gbps の Ethernet で接続した．

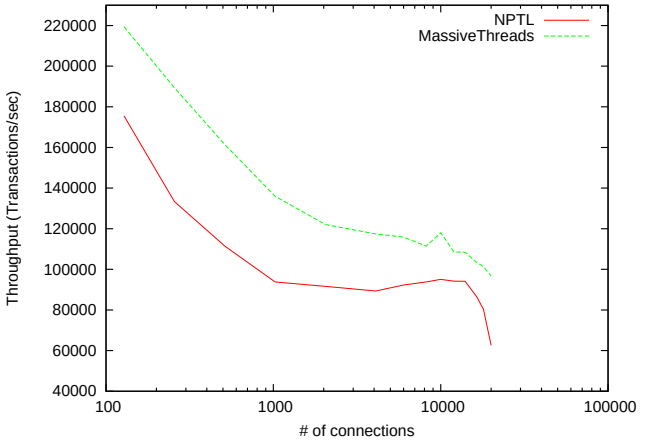


**図 6** 全部の接続が絶えず通信する場合のスループット
Fig. 6 I/O throughput (All connections active)

結果を**図 6**，**図 7**，**図 8** に示す．横軸は同時接続数，縦軸はトランザクションのスループットである．全ての条件において，MassiveThreads を利用することで，スループットの向上がみられている．

しかし，接続数に着目すると，同時接続数が増やしていくにしたがってスループットが低下していく傾向は MassiveThreads を適用しても変わらず，最終的にはスループットが最も高い場合の半分未満にまで落ち込んでしまう．まだ詳しい調査は行なっていないが，何が原因でこのような状態が発生しているのか特定し，実装に原因があるならば改善する必要がある．

## 5. おわりに

### 5.1 まとめ

本稿では，次世代のコンピュータの性能を引き出すための基盤ソフトウェアの実装を目的として，(1) 大量のスレッド管理 (2) 大量のスレッドによる I/O の効率的な実行 (3) 既存の処理系に適用できるインターフェースの 3 つを満足するスレッド処理系，MassiveThreads を提案した．

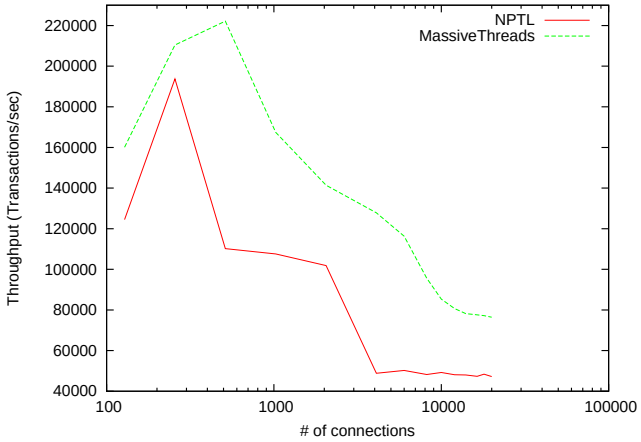


**図 7** 8 分の 1 の接続が同時に通信を行う場合のスループット
Fig. 7 I/O throughput (1 of 8 connections active)

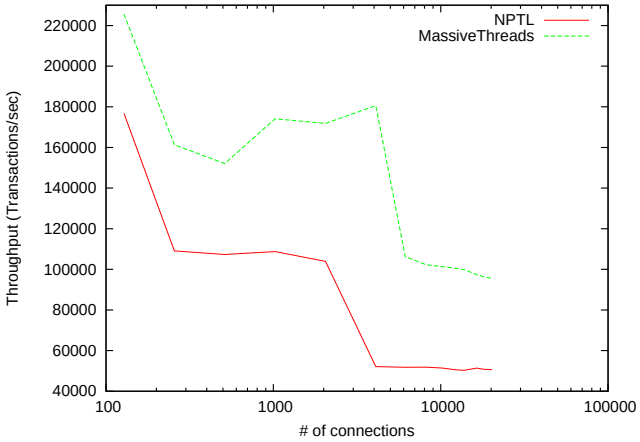


**図 8** 128 個の接続が同時に通信する場合のスループット
Fig. 8 I/O throughput (128 connections active)

現在の実装は，多くのスレッドを管理できる既存の処理系の技術と I/O 処理を効率的に処理できる既存の処理系の技術を組み合わせることで，既存のプログラムへ適用可能な形で多くのスレッドの管理と I/O の効率的な実行を両立している．

さらに，処理系のオーバーヘッド，負荷分散能力，および I/O 性能の評価を行った結果，既存の処理系と比較しても遜色のない小さいオーバーヘッド，良好な負荷分散，および OS 標準のマルチスレッド処理系と比較して高いネットワーク I/O のスループットが得られることを確認した．

### 5.2 今後の課題

#### 5.2.1 スケジューリングの改善

現段階の実装においては，ワーカースレッド内に実行可能なスレッドが存在する場合はそれを優先し，ランキュー内に実行可能なスレッドが存在しなくなった段階で I/O のチェックを行うものとしている．このスケジューリングは，ほとんどのスレッドが I/O 呼び出しを行い，かつ I/O 処理のレイテンシよりもスループットが重要視される場合や，ほとんどのスレッドが計算を行う場合には確かに有効であると考えられる．

しかし，本研究で対象とするアプリケーションはそのようなものに限定されない．例えば 1.3 で例として挙げた Data-intensive なアプリケーションにおいては，分散データに I/O を行うスレッドと計算を行うスレッドが混在する．また，並列言語のランタイムにおいてネットワーク I/O 経由で複数ノード間の同期を行う際は I/O 処理のレイテンシが重要になる．

したがって，これらのケースにおいてもアプリケーションの性能向上を達成するために，上で挙げたような場合においてもハードウェアの性能を活用することのできる，スケジューリング手法を検討・導入する必要がある．

#### 5.2.2 マルチスレッド処理系として必要な機能の追加

現在の MassiveThreads は基本的な機能しか実装していないため，実用的なアプリケーションを記述することは困難である．MassiveThreads を利用してより実用的なアプリケーションを記述できるようにするために，排他制御やスレッド間同期といった，マルチスレッド処理系として必要な機能を追加実装する必要がある．

さらに，OS から提供されるマルチスレッド処理系との互換性を高めるためには，シグナルやプリエンプションといった機能への対応も検討する必要がある．

## 参 考 文 献

1) : The International Exascale Software Project Roadmap, University of Tennessee EECS Technical Report (2010).

2) Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing, *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, ACM, pp.519–538 (2005).
3) Callahan, D., Chamberlain, B.L. and Zima, H.P.: The Cascade High Productivity Language, *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04*, pp.52–60 (2004).
4) Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H. and Zhou, Y.: Cilk: an efficient multithreaded runtime system, *SIGPLAN Not.*, Vol.30, No.8, pp.207–216 (1995).
5) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: integrating futures into calling standards, *SIGPLAN Not.*, Vol.34, No.8, pp.60–71 (1999).
6) Ayguadé, 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: Proceedings of the 3rd international workshop on OpenMP*, Berlin, Heidelberg, Springer-Verlag, pp.1–12 (2008).
7) Pheatt, C.: Intel®threading building blocks, *J. Comput. Small Coll.*, Vol.23, No.4, pp.298–298 (2008).
8) von Behren, R., Condit, J., Zhou, F., Necula, G.C. and Brewer, E.: Capriccio: scalable threads for internet services, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM, pp.268–281 (2003).
9) : GNU Pth - the GNU Portable Threads. http://www.gnu.org/software/pth/.
10) : State Threads for Internet Applications. http://state-threads.sourceforge.net/docs/st.html.
11) Duran, A., Corbalan, J. and Ayguade, E.: Evaluation of OpenMP Task Scheduling Strategies, *OpenMP in a New Era of Parallelism*, Lecture Notes in Computer Science, Vol.5004, Springer Berlin / Heidelberg, pp.100–110 (2008).
12) Frigo, M., Leiserson, C.E. and Randall, K.H.: The implementation of the Cilk-5 multithreaded language, *SIGPLAN Not.*, Vol.33, No.5, pp.212–223 (1998).
13) Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J. and wen Tseng, C.: UTS: An Unbalanced Tree Search Benchmark.