情報処理学会論文誌　プログラミング　Vol. 4　No. 1　13–26 (Mar. 2011)

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

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，これを両立させ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，という特徴を持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，それにともなって生じる，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．たとえば，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

†1 東京大学
The University of Tokyo

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 節で示される状況をふまえ，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フェースを備えること

3 番目の要件は 1.1 節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持つものである．

(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　データインテンシブな計算

データインテンシブな計算とは，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 章では，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 章では，MassiveThreads の実装について述べる．4 章では，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 章で，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)]，Java fork/join Framework[6)]，OpenMP task[7)]，Intel Thread Building Blocks[8)] 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

⋆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持つものを一括してスレッドと呼称する．

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 コア数分だけカーネルレベルのスレッドを起動し，これらがユーザレベルのスレッドの管理を行う．ユーザ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カーネル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．また，カーネルレベルのスレッドはプロセスとして実現されるためスレッド間の公平性を第 1 にスケジューリングが行われるが，ユーザレベルのスレッド処理系は公平性にとらわれない，アプリケーションの特性に応じた柔軟なスケジューリングポリシを行うことができる．

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，ユーザレベルで I/O を多重化することが広く行われている．Capriccio[9]，GNU Pth[10]，StateThreads[11] 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フェースを提供しながら，実際にはユーザレベルで 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 節とは逆に 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．

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 StackThreads/MP Java Fork/Join OpenMP Task Intel TBB	○	×	×
Capriccio GNU Pth	×	○	△
StateThreads	×	○	×
NPTL	×	×	○

- 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フェースを持ち，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．

3.　設計と実装

3.1　指　　針

MassiveThreads の実装の最終的な目的は，1.2 節であげた，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．しかし，この中で 3 番目の，既存の処理系との互換性という特徴に関しては，比較的簡単に，しかも残り 2 つの特徴と完全に直交する形で達成可能である．

そこで MassiveThreads の実装にあたっては，まず 2.1 節にあげた処理系の手法を参考にして軽量なスレッド管理を達成した処理系を実装し，それに 2.2 節にあげた処理系を参考にして効率的な I/O 処理を導入する，というアプローチをとった．

しかし，効率的な I/O を実現している既存の処理系はいずれも複数 CPU に対応していない．したがって，I/O 処理の手法を多くの CPU コア上でスケーラブルに動作し，かつスレッドの軽量性を損なわないように拡張することが MassiveThreads の実装上の課題と

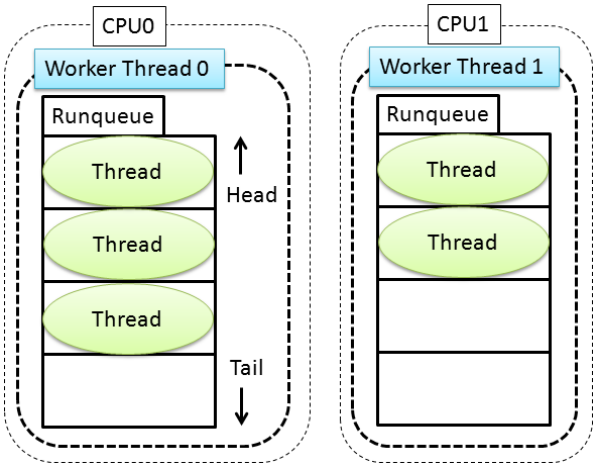

図 1　MassiveThreads の構成
Fig. 1　Organization of MassiveThreads.

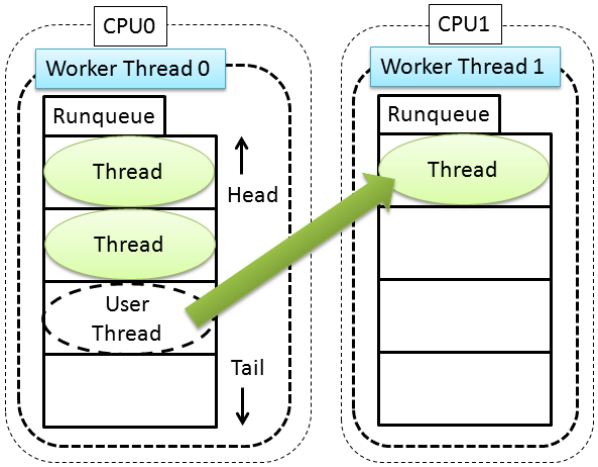

図 2　ワークスチール
Fig. 2　Work stealing.

なる．

本章では，まず軽量性を実現するための MassiveThreads のスレッド管理について 3.2 節で述べ，次に 3.3 節でそれに対して導入した I/O 処理の手法を説明する．最後に 3.4 節で既存の処理系との互換性を達成するための手法について説明する．

3.2　スレッドの管理

3.2.1　構　　成

MassiveThreads はユーザレベルスレッド処理系として実装した．その構成を**図 1** に示す．

各 CPU コアはカーネルレベルのスレッドをただ 1 つ持ち，これをワーカスレッドという．各ワーカスレッドは実行可能なユーザスレッドを格納するための deque [⋆1]であるランキューを持ち，ランキューの中のスレッドをスケジューリング戦略に従って順次実行することにより，マルチスレッド処理を実現する．

3.2.2　スケジューリング戦略

スレッドのスケジューリング戦略としては，ランキューの先頭にあるスレッドを優先して実行する，LIFO なスケジューリングを採用した．これは以下の 3 つの方針に従って，スレッドの実行順を決定する．

(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．

3.2.3　ワークスチール

LIFO なスケジューリング戦略を採用しているため，各ワーカスレッドは明示的にスレッドを他のワーカ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持たないワーカスレッドは，他のワーカ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スレッドをランダムに選択して繰り返す．

3.2.4　ランキューの実装

ランキューは，Java Fork/Join Framework において利用されているアルゴリズムを実装した．このランキューは，以下のような特徴を持っている．

⋆1 先頭と末尾の両方から挿入および取り出しができるキュー

(1)　先頭に追加する操作（push），先頭から取り出す操作（pop），および末尾に追加する操作（put）は，1 つのワーカスレッドしか行うことができない．
(2)　末尾から取り出す操作（take）のみは，全ワーカ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．

スケジューリングポリシとして LIFO なスケジューリングを採用しているため，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ヘッドを低減させるのに有効である．

3.2.5　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，1 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．

3.2.6　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，それをフリーリストに格納しておき，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，複数スレッド分の領域を 1 度に確保してそれを分割し，すぐに必要となる 1 スレッド以外の資源を除いてはフリーリストに格納するようにしている．

3.3　ブロッキング I/O の処理

3.3.1　構　　成

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 処理部の構成を**図 3** に示す．

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．図には Blocked List と表記）を対応付け，これを複数のワーカスレッドに分散配置する．図には表記されていないが，各ワーカスレッドは自身が受け持つファイルディスクリプタの状態をチェックするための epoll [⋆1] インスタンスを持ち，これによって I/O 可能になったファイルディスクリプタをチェックし，中断していたスレッドを再開させる．また，ファイルディスクリプタから I/O 待ちリストを検索するため，本処理系は 1 つのファイルディスクリプタと I/O 待ちリストのタプルを格納する連想配列を持ち，全ワーカスレッドはこれを共有している．

3.3.2　I/O 処理の手順

MassiveThreads は同期 I/O 関連の関数をフックし，以下のような手順で同期 I/O の効率的な処理を実現する．

3.3.2.1　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が作成された際は，fcntl [⋆2] 関数を用い，同期 I/O をつねにノン

⋆1 Linux で提供されている，ファイルディスクリプタの状態を監視するための機構．select や poll といった同様の機構と比較して，スケーラブルに多くのファイルディスクリプタを監視することが可能である．
⋆2 ファイルディスクリプタの状態を取得，設定する関数

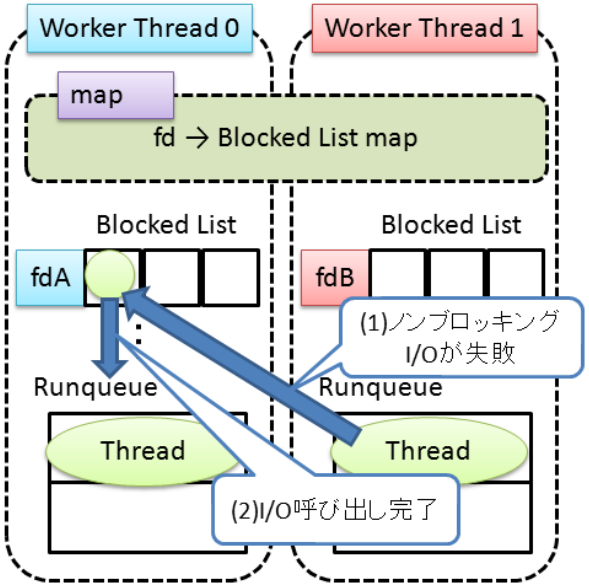

図 3　I/O の処理部分の構成
Fig. 3　Organization of handling I/O part.

ブロッキング I/O として行うように設定した後で，それをワーカスレッドの 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．

特定のワーカスレッドにファイルディスクリプタが集中することを防ぐため，ファイルディスクリプタを登録するワーカスレッドはランダムに決定される．

3.3.2.2　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．I/O 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 待ちリストを連想配列から検索し，それに自スレッドと I/O 呼び出しのパラメータを登録して他のスレッドに制御を移す（図 3 (1)）．

3.3.2.3　I/O のチェック

各ワーカスレッドは実行可能なスレッドがなくなった際，epoll 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 I/O 待ちリストに登録されている I/O 呼び出しを行う．もし I/O 呼び出しが完了したならばそのスレッドを自身のランキューに追加する（図 3 (2)）．もともとスレッドが実行されていたランキューではなく自身のランキューに追加することにより，I/O 呼び出しを行うスレッドが I/O チェックを行うワーカスレッドに集中し，スレッドがワーカスレッド間を移動する頻度を削減することが期待できる．

I/O 待ちリストに I/O 呼び出しのパラメータを登録し，I/O のチェックを行う段階で I/O 呼び出しの完了確認まで行うのは，スレッド間のコンテキストスイッチや，I/O 待ちリストを操作するオーバヘッドを最低限に抑えるためである．その効果は 1 つの I/O 待ちリストに多くのスレッドが登録される状況を想定すると分かりやすい．

I/O 待ちリストに I/O 呼び出しのパラメータが含まれない場合，再開可能な可能性のあるスレッド，すなわち I/O 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戻すこととなる．しかし，すべてのスレッドをランキューに戻したとしても，ほとんどのスレッドが再び I/O 待ちでブロックし，I/O 待ちリストに戻ってきてしまう，すなわちスレッドを再開したのが無駄になってしまう見込みが強い．

しかし，I/O のパラメータを I/O 待ちリストに入れてチェックを行うワーカスレッドに渡し，実際に I/O まで行ってそれ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 I/O 待ちリストに入ってしまう事態を避けることができる．

3.3.3　実装上の課題と対策

3.3.3.1　スケーラブルな I/O 待ちリスト検索の実装

ファイルディスクリプタから I/O 待ちリストを検索するための連想配列は全ワーカスレッドから共有されているうえに，I/O 呼び出しがブロックするたびに呼び出されるため，複数ワーカスレッドからの同時アクセスに対してスケーラブルである必要がある．

しかし，I/O チェックの際に I/O を行うスレッドがそのファイルディスクリプタを担当するワーカスレッドに集中する傾向を考えると，同じファイルディスクリプタへのアクセスが同時に行われることは少ないことが予想される．したがって，異なるファイルディスクリプタに対する同時アクセスが可能ならば実用上問題ないと考えられる．

そこで，連想配列の実装にはチェイン法によるハッシュテーブルを用いた．キーとしてはファイルディスクリプタの番号の下位 16 ビットを用いている．これによって，異なるキーを持つファイルディスクリプタに対する同時アクセスは互いに干渉することなく行われる．

3.3.3.2　いつ I/O チェックを行うか

I/O チェックの中で行われる epoll 関数のオーバヘッドは小さくないため，I/O チェックを過剰な頻度で行うことは性能の低下につながる．しかし，I/O チェックの頻度が少なすぎると実行可能なスレッド数が減少し，これも性能低下の原因となりうる．

暫定的にではあるが，MassiveThreads では実行可能なスレッドがワーカスレッド内に存在しなくなり，ワークスチーリングを行う前に I/O チェックを行う戦略を採用している．これは以下の 2 つの観察による．

- ワークスチーリングが行われる頻度はあまり高くないため，オーバヘッドの大きい操作を行ってもそれほど性能が低下しないことが予想される．
- 多くのスレッドが I/O 呼び出しによってブロックした場合，ある程度時間が経過しているため，最初にブロックしたスレッドの I/O 呼び出しが可能になっている可能性が高いと考えられる．

ただし，この戦略は明らかに不十分なものである．たとえば，あるワーカスレッドに CPU を占有し続けるスレッドが存在した場合，そのワーカスレッドに存在するファイルディスクリプタの I/O チェックは永遠に行われない．したがって，そのファイルディスクリプタに対する I/O 呼び出しがブロックしたスレッドも永遠に再開されなくなってしまう．

適切な頻度で，しかも前述のようなデッドロックを起こさないような I/O チェックの手法を検討することは今後の課題である．

3.3.3.3　エッジトリガの導入

I/O チェックで利用している epoll には「呼び出した時点で I/O 可能だったファイルディスクリプタに対応するデータを返す」レベルトリガと「以前は I/O 不可能だったが，呼び出した時点までに I/O 可能に変化したファイルディスクリプタに対応するデータを返す」エッジトリガという 2 つの動作モードがある．

これまでのはレベルトリガで epoll を利用することを前提としてきた．レベルトリガを利用する場合は，「I/O 可能だが，スレッドをブロックさせておらず，I/O 待ちリストが空なので実は I/O 待ちリストを探索する必要のないファイルディスクリプタ」も I/O 可能であると見なされるが，エッジトリガの場合はこれを除外することができ，より I/O チェックのオーバヘッドを削減することができる．

しかし，単に epoll の動作モードをエッジトリガに変更するだけでは以下のようなケースでデッドロックが発生してしまう．

(1)　2 つのワーカスレッド（W1，W2）が存在し，W1 があるファイルディスクリプタ（fdX）を担当している．

(2)　W2 上のスレッドが fdX に対して I/O 呼び出しを行い，それがブロックする．

(3)　fdX が I/O 可能に変化し，さらに W1 の I/O チェックが発生する．

(4)　W2 が fdX の I/O 待ちリストにスレッドを登録する．

(5)　fdX は実は I/O 可能なのだが，I/O 可能に変化することはないため，永遠に I/O チェックで検出されない．したがって fdX を待っているスレッドは永遠に再開されない．

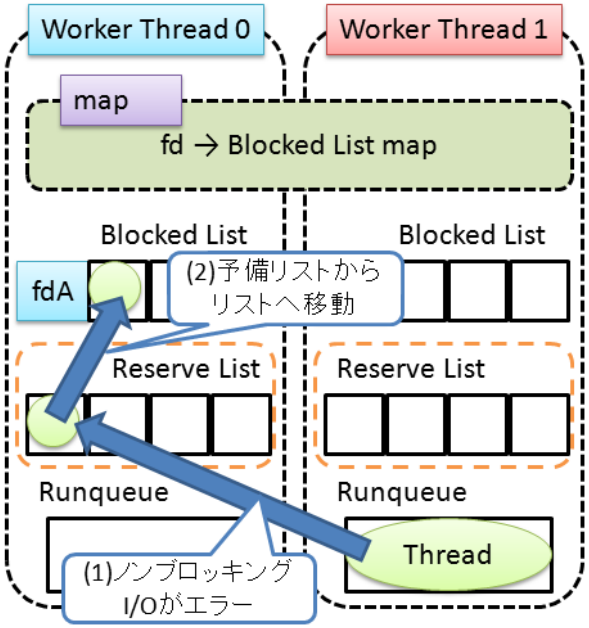

図 4　エッジトリガの epoll を利用するための予備リストの導入
Fig. 4　Reserve list for edge-triggered epoll.

これを回避するため，I/O 処理の構成に各ワーカスレッドにスレッドと I/O 呼び出しのパラメータを格納する通知リストを追加した（**図 4**）．各ワーカスレッドは I/O 待ちリストにスレッドを直接追加する代わりに，そのファイルディスクリプタを担当するワーカスレッドの予備リストにスレッドと I/O 呼び出しのパラメータを格納する（図 4 (1)）．I/O チェックの際に各ワーカスレッドは予備リストをチェックしてスレッドの I/O 呼び出しを再試行

し，もし I/O 呼び出しが成功したならばそのスレッドはランキューに戻す．ブロックしたならば I/O 待ちリストに登録される（図 4 (2)）．

3.4　既存のアプリケーションとのインタフェース

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 章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（2.5 GHz）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　オーバヘッドの評価

オーバヘッドの評価は，以下のような手順によってフィボナッチ数を計算するベンチマークによって行った．

(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 n を返して即座に終了する．
(2)　第 n 項を計算するスレッドは第 $n-1$ 項と第 $n-2$ 項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．

フィボナッチ数列の第 40 項を計算させた際の，各処理系のオーバヘッドを**図 5** に示す．横軸は使用した CPU コア数，縦軸は各処理系の 1 スレッドあたりのオーバヘッドである．各処理系のオーバヘッドは実行時間と，上記の処理を再帰によって実装したプログラムの実行時間から，以下の式によって計算した．

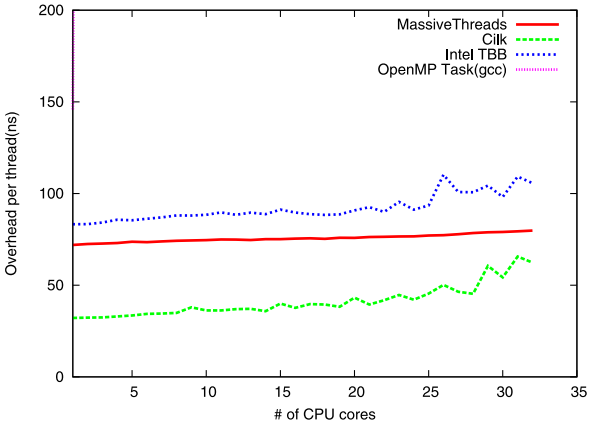

図 5　fib(40) を計算した際のオーバヘッド
Fig. 5　Overhead of fib(40) calculation.

$$\text{オーバヘッド} = \text{実行時間} - \frac{\text{再帰版の実行時間}}{\text{使用した CPU コア数}}$$

MassiveThreads のオーバヘッドは，1 コアの場合 72 ナノ秒，32 コアすべてを使う場合でも 80 ナノ秒程度であり，これは既存の処理系と比較しても遜色ない値である．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，このオーバヘッドは十分に小さいといえる．また，いずれの処理系も使用する CPU コア数が増加するに従ってオーバヘッドが増大しているが，MassiveThreads は既存の処理系と比較して増加量が小さくなっている．これはスレッドの管理を他の処理系と比較して，よりスケーラブルに行えていることを示している．

次に，CPU コア 1 つを使った際のスレッド管理に消費されたサイクル数の内訳を**表 2** に示す．比較対象として，評価を行った 4 種類の処理系の中で最もオーバヘッドの小さかった Cilk のオーバヘッドも併記した．

MassiveThreads は特にコンテキスト切替えのオーバヘッドが Cilk と比較して大きくなっていることが分かる．これは，Cilk のコンテキスト切替えがスレッド作成後に読み込まれるローカル変数（本ベンチマークでは 1 個の整数のみ）を退避させ，関数呼び出しによって切替え先に制御を移す，という手順で行われるのに対して，MassiveThreads においては関数呼び出し前後で保存される必要のあるレジスタすべてを退避させ，さらにスタックを切り

表 2　Cilk と MassiveThreads のオーバヘッドの内訳
Table 2　Summary of overheads of Cilk and MassiveThreads.

	MassiveThreads	Cilk
スタック確保・解放	17.89	14.80
ランキュー操作	8.90	2.09
コンテキスト切替え	22.21	2.03
合流処理	32.18	37.85
その他（関数呼び出しなど）	36.45	5.57
合計	117.63	62.23

替えたうえで切替え先に制御を移す，という手順で行われるため，処理の手順が比較的複雑で，さらに退避の対象となるデータ量が多いためである．

また，スレッド管理には直接該当しない，処理系の関数を呼び出す部分などのオーバヘッドも特に大きくなっている．これは，Cilk においてはスレッド管理を行うコードの大部分がソースコードに直接埋め込まれてコンパイルされるのでインライン化や関数呼び出しをまたいだ最適化を利用できるのに対し，MassiveThreads は共有ライブラリとして実装されているためインライン化が利用できないうえ，関数呼び出しが PLT [*1] を経由するため関数呼び出しそのもののオーバヘッドも通常の関数呼び出しより大きいことが原因であると推測される．

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 [12] を用いた．これは，一定の規則に従って生成される，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に実行性能を向上させるためには，処理系の軽量性だけでなく負荷分散能力も重要である．

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持つ．
(2)　各ノードは確率 q で，m 個の子ノードを持つ．
(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．このデータセットは $q = 0.124875$，$m = 8$ で，最大深さ 1,572，約 411 万ノードを持つ木構造を生成する．

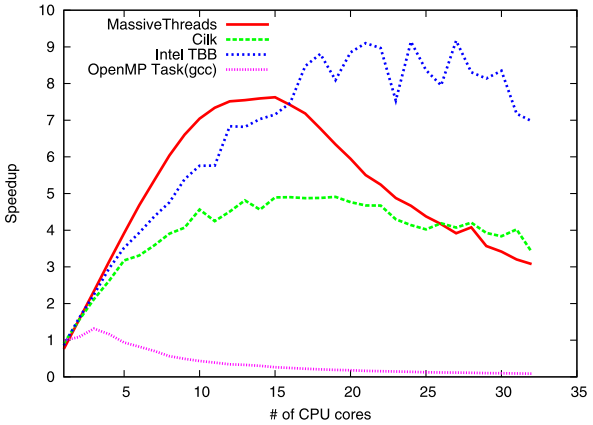

図 6　UTS Benchmark の結果
Fig. 6　The result of UTS Benchmark.

木のノード 1 つあたり 1 つのスレッドを作成する形で UTS Benchmark を並列化した際の実行性能を**図 6** に示す．横軸は使用した CPU コア数，縦軸は再帰版と比較した際の性能向上率である．

MassiveThreads は CPU コア数が 10 程度まではほぼ線形に性能向上し，他の処理系よりも高い実行性能を達成できているが，CPU コア数が 16 を超えると徐々に性能が低下していき，32 コアにおいては 3 倍程度にまで落ち込んでしまう．Intel TBB は CPU 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 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．

MassiveThreads が 32 コアを利用した場合，性能向上は MassiveThreads で並列化したものを 1 コアで実行した場合と比較して 4 倍程度であった．もし負荷の不均衡のみでこの性能低下が生じていると仮定すると，CPU 時間の 87.5%においてワーカスレッドはアイドル状態であることになる [*2]．しかし，実際にベンチマークを実行し，アイドル状態の CPU

⋆1 Procedure Linkage Table の略．共有ライブラリの動的ロードを実現するためのテーブルで，これを経由すると間接ジャンプ 1 回分のオーバヘッドを被る．

⋆2 アイドル状態の割合 = (CPU コア数 − 性能向上率) ÷ CPU コア数より導出される．

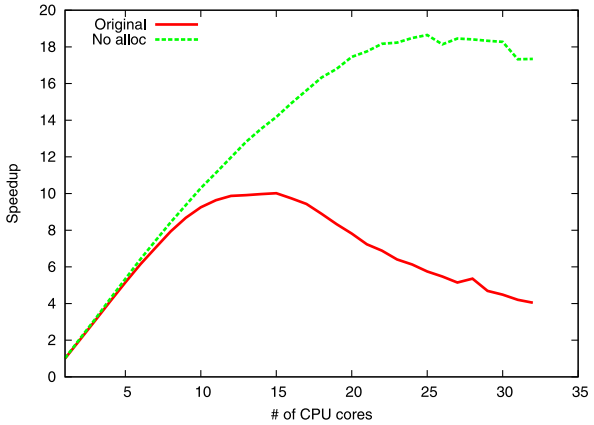

図 7　スタック確保とページフォルトのオーバヘッドを回避した場合の性能向上率
Fig. 7　Scalability without overheads of stack allocation and page faults.

時間を計測した結果は全体の 5%程度であった．したがって性能低下の主な原因は負荷の不均衡ではなく，CPU コア数が増加するに従って処理系のオーバヘッドが増大することに起因するものであると考えられる．

そこでオーバヘッドの所在を明らかにするため，32 コアでのスレッド管理に消費された時間の内訳を測定したところ，スレッド作成のオーバヘッドが 1 コアの場合 1 スレッドあたり約 140 ナノ秒だったのが，32 コアでは約 4,020 ナノ秒と大幅に増大しており，その原因は確保したスタックに最初にアクセスする際のページフォルトであることが分かった．現在の実装にはスレッドを管理する構造体とスタック領域を一組にし，同じタイミングで確保・解放を行っているため，終了待ちをしているスレッドのスタック領域が再利用されるタイミングがスレッド合流時まで遅延してしまい，メモリを確保する量が多くなるという問題点があるが，それもページフォルトのオーバヘッドが増大する原因の 1 つとなっている．

さらにそれを確認するため，計算前に必要なだけのスタック領域を確保し，さらに 1 度アクセスしておくことで，スタック確保とページフォルトのオーバヘッドを回避した際の実行性能を**図 7** に示す．横軸は使用した CPU コア数，縦軸は MassiveThreads を 1 コアで実行した場合を基準にした性能向上率である．Original は MassiveThreads を，No alloc はスタック確保のオーバヘッドを回避したものを示している．CPU コア数が多くなると性能向上は頭打ちになるが，性能が極端に低下するようなことはなくなっている．

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メッセージを返す．
(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)　全部の接続がた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，MassiveThreads を利用して同様の記述を行ったものの 2 種類についてベンチマークを行い，そのスループットを比較した．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っ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（2.33 GHz）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，10 Gbps の Ethernet で接続した．

結果を**図 8**，**図 9**，**図 10** 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．ほとんどの条件において，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．また，同時接続数が増やしていくに従ってスループットが低下していく傾向がいずれの処理系にもみられている．MassiveThreads においては，すべての接続が同時に通信する場合と，同時にメッセージを送受信できる接続数が全体の 8 分の 1 である場合にそれが顕著である．

スループット減少の原因としては，大きく分けて処理系のオーバヘッドと I/O 待ち時間の増加の 2 種類が考えられる．どちらが主要な原因であるかを特定するため，サーバ側の CPU 時間をプログラムの貢献している部分（スレッドの実行時間，成功したワークスチー

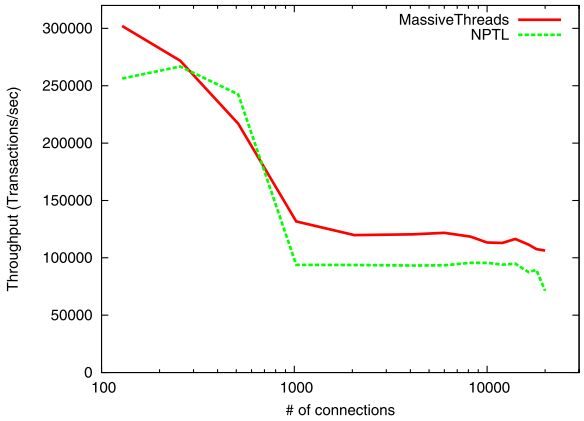

図 8　全部の接続がたえず通信する場合のスループット
Fig. 8　I/O throughput (All connections active).

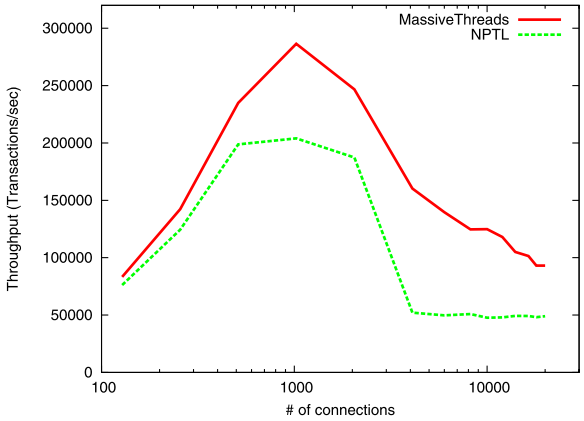

図 9　8 分の 1 の接続が同時に通信を行う場合のスループット
Fig. 9　I/O throughput (1 of 8 connections active).

ル，スレッドを実行可能にした I/O チェック）と進捗に貢献していない部分（失敗したワークスチール，何もしなかった I/O チェック）に分類して測定した．これらの分類はそれぞれ処理系のオーバヘッドと I/O 待ち時間に対応している[⋆1]．

結果をトランザクション 1 回あたりに正規化したものを**図 11**，**図 12**，**図 13** に示す．グラフには進捗に貢献している部分の CPU 時間から，I/O 呼び出しの時間を除去したものも併記した．

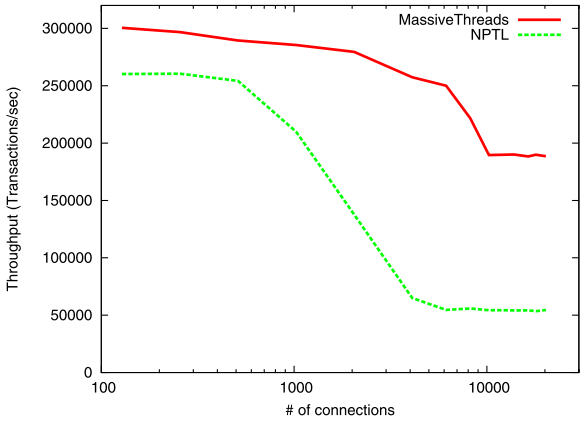

図 10　128 個の接続が同時に通信する場合のスループット
Fig. 10　I/O throughput (128 connections active).

すべての接続が同時に通信する場合，接続数が小さい範囲では進捗のある時間（=処理系のオーバヘッド）が CPU 時間の大半を占めているが，接続数が増加するに従って進捗のない時間（=I/O 待ちをしている時間）が大きく増加する．同時にメッセージを送受信できる接続数が全体の 8 分の 1 である場合も同様に，接続数が増加するに従って I/O 待ちの時間がいったん減少し，その後大きく増加する．したがって，これらの条件においてスループットが減少する主な原因は I/O 待ちの時間が増加することであると考えられる．また，これらの条件においては，処理系のオーバヘッドの大半を I/O 呼び出しが占め，それ以外の，アプリケーションと MassiveThreads そのものに由来すると考えられるオーバヘッドはそれと比較して小さく，しかも接続数が増加してもあまり大きくならない．

いっぽう，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には，接続数によらず処理系に由来するオーバヘッド，その中でも特に I/O 呼び出しが多くを占めており，I/O 待ちの割合は小さい．

⋆1 厳密にいえば前者にはアプリケーションが使う CPU 時間も含まれている．しかし本ベンチマークにおいては各スレッドはひたすら I/O 呼び出しを行うのみなので，アプリケーションそのものが実行されている時間は無視できる程度に短いと考えられる．

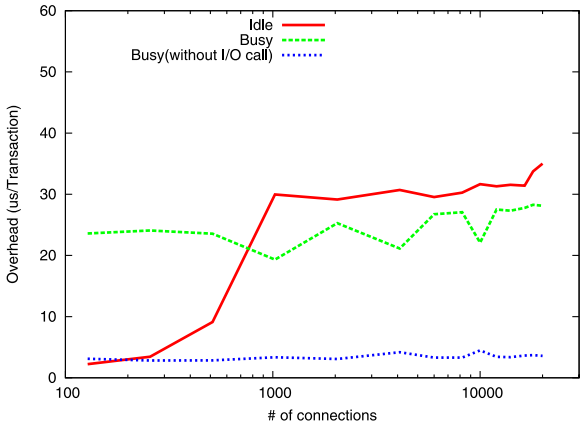

図 11　すべての接続が同時に通信する場合の CPU 時間の内訳
Fig. 11　CPU time summary (All connections active).

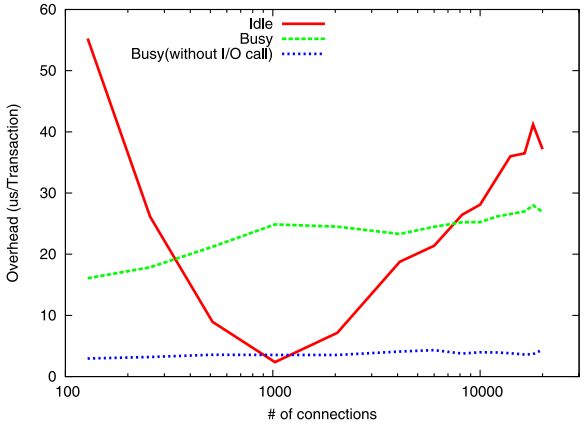

図 12　8 分の 1 の接続が同時に通信を行う場合の CPU 時間の内訳
Fig. 12　CPU time summary (1 of 8 connections active).

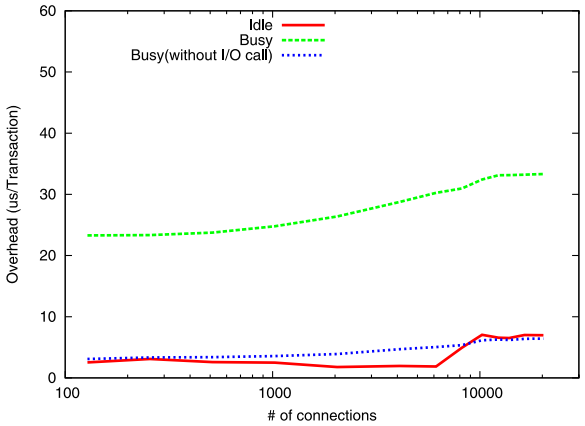

図 13　128 個の接続が同時に通信する場合の CPU 時間の内訳
Fig. 13　CPU time summary (128 connections active).

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．

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させている．

さらに，処理系のオーバヘッド，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，3.3.3.2 で言及したように，このスケジューリングポリシだけでは不十分であり，アプリケーションによってはデッドロックの可能性すらある．また，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．たとえば 1.3 節で例としてあげたデータインテンシブ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，上であ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．

謝辞　本研究の一部は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する新 IT 基盤研究プラットフォームの構築」の助成を得て行われた．

参　考　文　献

1) Dongarra, J., Beckman, P., Cappello, F., Lippert, T., Matsuoka, S., Messina, P., Aerts, P., Trefethen, A. and Valero, M.: 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: Proc. 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, pp.519–538, ACM (2005).
3) Callahan, D., Chamberlain, B.L. and Zima, H.P.: The Cascade High Productivity Language, *9th 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) Lea, D.: A Java fork/join framework, *JAVA '00: Proc. ACM 2000 conference on Java Grande*, New York, NY, USA, pp.36–43, ACM (2000).
7) 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: Proc. 3rd international workshop on OpenMP*, pp.1–12, Berlin, Heidelberg, Springer-Verlag (2008).
8) Pheatt, C.: Intel®threading building blocks, *J. Comput. Small Coll.*, Vol.23, No.4, pp.298–298 (2008).
9) 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).
10) Engelschall, R.S.: GNU Pth — the GNU Portable Threads. http://www.gnu.org/software/pth/
11) Unknown: State Threads for Internet Applications. http://state-threads.sourceforge.net/docs/st.html
12) Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J. and wen Tseng, C.: UTS: An Unbalanced Tree Search Benchmark.

(平成 22 年 7 月 5 日受付)
(平成 22 年 11 月 18 日採録)

中島　　潤（学生会員）
1986 年生．2009 年東京大学学士（工学）．同年より東京大学大学院情報理工学系研究科電子情報学専攻在学中．

田浦健次朗（正会員）

1969 年生．1997 年東京大学大学院理学博士（情報科学専攻）．1996 年より東京大学大学院理学系研究科情報科学専攻助手．2001 年より東京大学大学院情報理工学系研究科電子情報学専攻講師．2002 年より同助教授．2007 年より同准教授．日本ソフトウェア科学会，ACM，IEEE-CS 各会員．