修　　士　　論　　文

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の設計と実装

Design and Implementation of Multithread Framework that Manages both Efficient I/O and Lightweight Thread

指導教員　**田浦 健次朗 准教授**

東京大学情報理工学系研究科
電子情報学専攻

氏 名　**48-096415 中島 潤**

提 出 日　**平成 23 年 2 月 9 日**

概 要

次世代の並列計算機の性能を十分に引き出すためには，そのあらゆる階層について，並列性を十分に活用することを意識してアプリケーションを記述することが求められる．そのうえ，既存の並列処理系の多くは動的な負荷分散を提供していないため，適合格子細分化法などのそれが必要となる手法を用いたアプリケーションを記述する際には，負荷分散のための処理も記述しなくてはならない．これは開発者にとって大きな負担である．

そこで我々は，アプリケーションの並列性に応じてスレッドを生成・破棄し，それらが自由に通信を行う，という直感的な形で高性能な並列分散アプリケーションを記述するためのマルチスレッド処理系，MassiveThreads を提案している．これは，(1) 軽量なスレッド管理 (2) マルチコア上でのスケーラブルな負荷分散 (3) 大量のスレッドによる I/O の効率的な実行 (4) 既存のマルチスレッド処理系からスムーズに移行できるインターフェース，の 4 つの特徴をもつマルチスレッド処理系である．

本論文では，MassiveThreads の設計と実装について述べる．MassiveThreads は，既存のマルチスレッド処理系が軽量なスレッド管理や効率的な I/O のために利用している手法をベースとして，それらを既存のコードとも高い親和性をもち，なおかつマルチコア上でスケーラブルに動作するよう拡張して組み合わせることで，上に述べた 4 つの特徴を達成している．

次にマイクロベンチマークによって，MassiveThreads が既存のマルチスレッド処理系と比較しても遜色ない軽量性，負荷分散能力，高いネットワーク I/O のスループットを兼ね備えていることを確認した．さらに，適合格子細分化法による移流計算を MassiveThreads を用いて並列化した結果，動的な負荷分散によって静的な領域分割よりも高い実行性能と良好な負荷分散が得られ，実アプリケーションにおける MassiveThreads の有効性を確認することができた．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

1.1.1　並列計算環境の大規模化

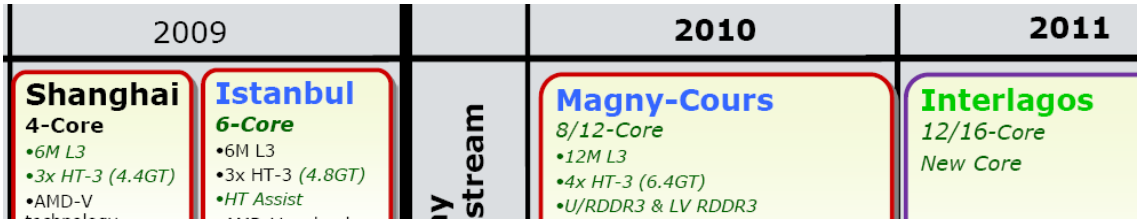

図 1.1: AMD の CPU ロードマップ (抜粋)

近年の並列計算環境の性能向上は，主に並列性を高めることによって行われている．例えば CPU コアにおいては Intel AVX[1] のような，より多くのデータに対して同時に演算が可能な命令が定義され，1 つの CPU の中には AMD のロードマップ (図 1.1) が示すようにより多くの CPU コアが実装される．Cray XMT[2] のような，1 つの CPU コアに多く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，スループット志向の計算機も次世代アーキテクチャとして注目されている．

さらに 1 つの計算ノードは多くの CPU を内部にもち，高速インターコネクトにより接続されたより多くの計算ノードが 1 つの並列計算機を構成している．例えば，理研が開発中の次世代スパコン「京」においては，8 コア CPU をもつ計算ノード 8 万台を 6 次元のメッシュ/トーラス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．

International Exascale Software Project によって発行されている，エクサスケールに対応したソフトウェア開発のためのロードマップ [3] によれば，このような並列性の向上が進んだ結果，2018 年頃に並列計算機はエクサスケールを達成し，そのシステムは全体で 100 億程度の並列性をもつとされている．

このような計算機の性能を十分に発揮するために，アプリケーションを記述する開発者は SIMD 命令による複数データの同時処理からネットワークを介した複数ノードによる協調処理に至るまで，あらゆる階層においてそれらがもつ並列性を活用できるよう，注意深くアプリケーションを記述する必要がある．

1.1.2 動的な負荷分散を必要とするアプリケーション

より大規模な計算を高精度に，かつ高速に行うことに対する要請を満たすため，近年の科学技術計算においては，流体シミュレーションに対する適合格子細分化法 [4] や重力多体計算に対する高速多重極展開法 [5] といった，部分領域ごとの計算の手順をその性質に応じて動的に変更することで，計算量の増加を抑えつつより高精度な解を求める手法が利用されている．これらを用いた並列アプリケーションは，計算が進むにしたがって部分領域の計算負荷が変動する．したがって並列実行による性能向上のためにはそれに従って動的に負荷分散を行うことが必要である．

しかし，並列アプリケーションを記述するために広く用いられている既存の処理系は動的に負荷分散を行う機構を提供していない．そのため，このような手法を用いるアプリケーションを 効率的に並列化するためには，アルゴリズムの本質とは関係ない，動的に計算負荷を分散させるためのコードを記述しなければならない．

1.2 MassiveThreads の提案

しかし，これらすべてを考慮しつつアプリケーションを記述することは開発者にとって大きな負担であると考えられる．そこで，我々は計算ノード上の CPU コアによる並列性を抽象化することで，次世代並列計算環境上で動作する並列アプリケーションを高生産に記述することを可能とする基盤ソフトウェア，MassiveThreads を提案している (図 1.2)．これは以下に示す 4 つの特徴をもつマルチスレッド処理系である．

1. 動的に大量のスレッドを生成・破棄できるように，それらの操作を小さなオーバーヘッドで行えること．
2. 多くの CPU コアを備えた計算機上でスケーラブルに負荷分散を行えること．
3. 大量のスレッドからの I/O 呼び出しを多重化し，効率的に処理できること．
4. 既存のマルチスレッド処理系からスムーズに，可能ならば再コンパイルや移植を行わずに移行できること．

MassiveThreads の (1)(2) の性質により，アプリケーションの開発者は上に示したようなアプリケーションに必要とされる動的な負荷分散を，アルゴリズムの並列性に応じた数のスレッドを動的に生成することで達成することができる．さらにノード間通信によって複数のノードを協調動作させる際にも，(3) の性質により，多くのスレッドから通信を行うことによる性能低下を被ることなく，各スレッドが自由に通信を行うような直感的な記述を用いることができる．

4 番目の要件は性能上の要件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 を利用することが可能になる．さらに，これを満たすために既存のマルチスレッド処理系と類似したインターフェースを用いることにより，より小さい学習コストで直接 MassiveThreads を利用するコードを記述できるようになると考えられる．

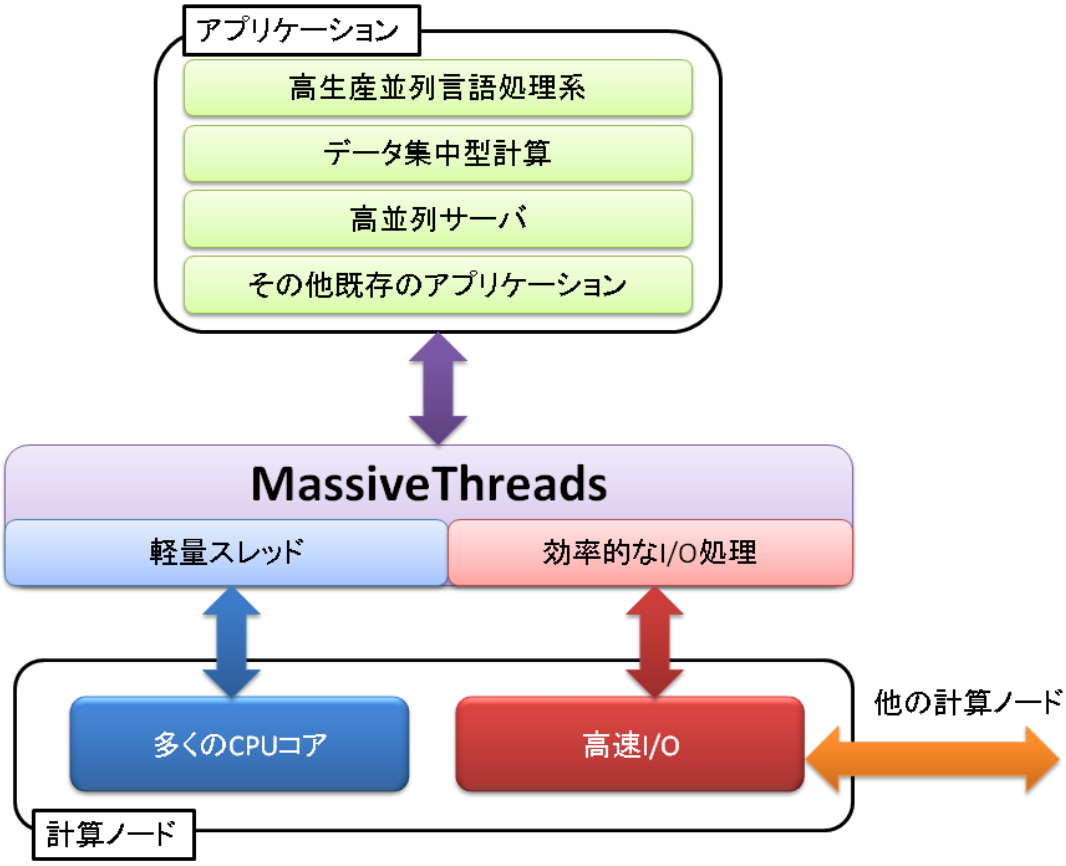

図 1.2: 並列計算環境における MassiveThreads の位置づけ

1.3 本研究の貢献

特に MassiveThreads によって生産性を向上させることができる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 高生産並列言語のランタイム

動的に並列度が変化するようなアプリケーションについても効率的に並列処理を行うために，近年開発されている並列分散アプリケーションを高生産に記述することを目標とする言語の多くは，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生成し，非同期にコード片を実行する機能を定義している．

さらに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生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アプリケーション

画像処理や自然言語処理といった，大容量データに対する大規模な処理を行うアプリケーションをデータインテンシブなアプリケーションという．このようなアプリケーションでは，データを計算機間に分散するための大量のデータ転送が発生するため，効率的にネットワーク I/O を処理することと，多くのスレッドを生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，MassiveThreads の要件のうちの一部を達成している既存のマルチスレッド処理系について説明し，それらと MassiveThreads の目標についての比較を行う．3 章では，MassiveThreads の設計に用いている要素技術として，マルチスレッド処理系を実現するための基本的な手法と，前

章で説明した既存研究がそれぞれの目標を達成するために利用している手法を説明し，次の 4 章では，MassiveThreads が解決すべき課題を明らかにしたうえで，その設計と実装について述べる．5 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，その結果について述べる．6 章では，適合格子細分化法による移流計算という実アプリケーションを MassiveThreads によって並列化し，性能評価を行なった結果とその考察を述べる．最後に 7 章で，本稿の結論および今後の課題について述べる．

第2章　関連研究

2.1　軽量スレッドによる動的負荷分散を実現している処理系

N-Queen 問題などの動的に並列度が変化するアプリケーションを，計算領域を静的に分割するような単純な手法で効率的に並列化することは困難である．計算領域を静的に分割するだけでは負荷分散が取れず，負荷分散を取るのに十分なスレッドを生成するとそのオーバーヘッドが無視できないためである．

このようなアプリケーションの並列化を効率的に行うことを目的として，Multilisp[6]，Cilk[7]，StackThreads/MP[8]，Java Fork/Join Framework[9]，OpenMP Task[10]，Intel Thread Building Blocks[11]，PFunc[12] などの多くの処理系が提案されてきた．

これらはいずれも「十分に軽量で，かつ多くの CPU コア上でスケーラブルに負荷を分散できるスレッド[1]をアプリケーションの並列性にしたがって動的に生成・破棄することによって，動的な負荷分散を行う」という思想に基づいている．

しかし，これらの処理系は 1 台の計算ノード内ですべての処理が完結することを前提として設計されている．そのため，例えば各スレッドが I/O を用いて他の計算ノードと協調し，分散処理を行うような動作は想定されていない．もし仮にそのような処理を行った場合，たとえ他のスレッドが実行可能であったとしても I/O が完了するまでその CPU コアでは何もスレッドが実行されなくなるため，CPU コアの利用効率が著しく低下する．

2.2　I/O 多重化により高効率な I/O を実現している処理系

高並列サーバのような，多くの接続に対する I/O の効率的な処理が特に重要なアプリケーションを実装する際には，複数の I/O の処理を同時に行う I/O 多重化が用いられる．これは接続 1 つにつき 1 つのスレッドを割り振り，それぞれのスレッドが独立に I/O を行うことでプログラムの意味論的には容易に達成可能である．

しかし，OS が提供するスレッド処理系には (1) リソースの制約から多くのスレッドを生成するのは困難である (2) 多くのスレッドが I/O 呼び出しを頻繁に繰り返すため，スレッド切り替えが頻繁に発生することになるが，OS 標準のスレッド処理系は切り替えのオーバーヘッドが大きいためそれが性能に与える影響が大きい，という 2 つの問題点があるため，単に 1 スレッドに 1 つの接続を割り振っただけでは効率的な I/O 多重化を達成できない場合が多い．そこで実際には，I/O の完了を

[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．

待機するための API とノンブロッキング I/O を用い，イベント駆動型プログラミングによって I/O を多重化する手法が主流となっている．

しかし，これは OS 標準のスレッド処理系の実装に由来する問題であり [13]，スレッドというプログラミングモデルそのものに問題があるわけではない．Capriccio [14]，StateThreads [15]，GNU Pth [16]，などといった処理系は，ユーザーレベルでマルチスレッド処理を実装することで大量のスレッドを生成可能にするとともにスレッド切り替えのオーバーヘッドを削減して OS 標準のスレッド処理系の問題を解決し，さらに処理系の実装としてノンブロッキング I/O によるイベント駆動プログラミングを用いて I/O の多重化を行うことで，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．

しかし，これらの処理系は 2.1 節に挙げた処理系とは逆に I/O を行う効率のみに着目しており，複数の CPU コアを利用することができないため，複数の CPU コアをもつ計算ノードによる並列処理に用いるには難がある．

2.3 並列プログラミング処理系

複数の計算ノードを用いる並列処理を記述するための処理系としては，他のプロセスとの通信をすべて明示的に記述する形で並列処理を記述する MPI[17] が現在最も広く用いられている．また，全プロセスが共有するグローバルなアドレス空間を提供し，ノード間の通信を抽象化することでより高生産にアプリケーションを記述できる処理系として，UPC[18]，Titanium[19]，Co-Array Fortran[20] などが提案されている．しかし，これらの処理系は動的な負荷分散を行うための機構を提供しない．そのため動的に並列度が変化するアプリケーションを実装する際にはそのためのコードを記述する必要がある．

X10[21] や Chapel[22] などの高生産並列言語は，グローバルなアドレス空間に加えて，並列処理の基本単位として軽量スレッドを定義している．ノード間の I/O による通信がグローバルなアドレス空間によって抽象化されていることを考えると，これらの言語は，軽量なスレッドと I/O 多重化による高効率な I/O の 2 つの要件を同時に達成している，ととらえることもできる．しかし，これらのランタイムの実装はまだまだ途上段階で，動的にスレッドを生成して負荷分散をとることができるような軽量性を備えているわけではない．

2.4 研究目標との比較

既存研究と比較して，MassiveThreads が目標とする特徴を明らかにするため，表 2.1 に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示す．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．

表 2.1: 既存の処理系が満たす特徴と MassiveThreads の目標の比較

	軽量性・動的負荷分散	I/O 性能	互換性
MassiveThreads	◯	◯	◯
Multilisp Cilk StackThreads/MP Java Fork/Join OpenMP Task Intel TBB PFunc	◯	×	×
Capriccio GNU Pth	×	◯	△
StateThreads	×	◯	×
MPI UPC Titanium Co-Array Fortran	×	◯	×
X10 Chapel	△	◯	×
NPTL	×	×	◯

第3章　要素技術

3.1　マルチスレッド処理系の実現手法

マルチスレッド処理系は，スレッドとプロセスの対応関係によって，1 対 1 モデル，N 対 1 モデル，M 対 N モデルの 3 種類に大別される．

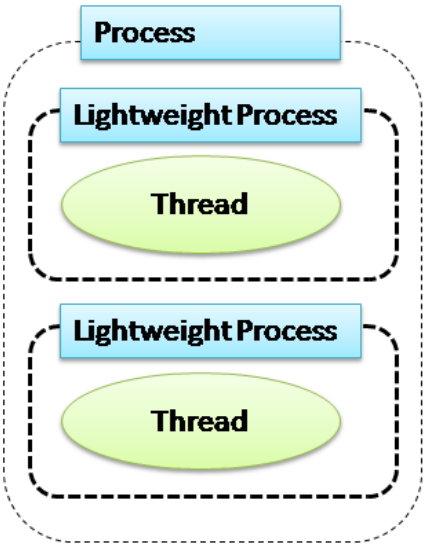

図 3.1: 1 対 1 モデル

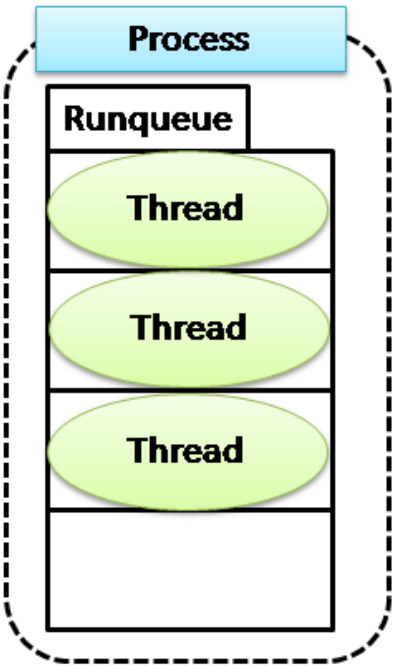

図 3.2: N 対 1 モデル

3.1.1　1 対 1 モデル

1 対 1 モデルでは，1 つのスレッドにつき 1 つの互いにメモリ空間を共有するプロセス (軽量プロセス) が割り当てられ，マルチスレッド処理は OS のカーネルに備わる，プロセスを管理する機能を

利用して実現される (図 3.1)．そのため，1 対 1 モデルのマルチスレッド処理系はカーネルレベルのマルチスレッド処理系とも呼ばれる．

1 対 1 モデルでは，プロセスとスレッドが固定的に割り付けられるため，スレッド間の公平性に配慮したスケジューリングやプリエンプティブなスレッドの切り替え，I/O がブロックした際のスレッド切り替えなどの，ユーザーレベルでは実現が難しい処理の多くを既存のプロセス管理の機構を利用して達成できるというメリットがある．そのため，近年の多くの OS において，標準的に提供されているマルチスレッド処理系はこの 1 対 1 モデルである．

しかし，スレッド生成・破棄やスレッド間の切り替えはカーネルによって，プロセスのそれに準ずる形で行われるため，オーバーヘッドが大きい．例えばスレッドを生成する際にはシステムコールの呼び出しと，その内部で多くのプロセス情報がコピーされることによるオーバーヘッドが生じる．また，スケジューリングのポリシーはあくまでプロセスのそれに準じたものに限定される．

3.1.2 N 対 1 モデル

N 対 1 モデルは，1 つのプロセスがユーザー空間でスレッドの切り替えを行うことにより，マルチスレッド処理を実現する (図 3.2)．ユーザー空間でスレッドのスケジューリングが行われることから，N 対 1 モデルは後述する M 対 N モデルとあわせてユーザーレベルスレッド処理系とも呼称される．

N 対 1 モデルにおいては，スレッド管理がユーザー空間で行われるため，システムコール呼び出しの必要がなく，操作する情報量も相対的に小さく済むためオーバーヘッドが小さい．また，スケジューリングがプロセスのスケジューリングと独立であるため，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を行うことができる．

しかし，プロセスが一つしか存在しないため複数の CPU コアを活用することが不可能である．また，I/O 操作がブロックした際のスレッド切り替え，プリエンプティブなスケジューリングなど，ユーザーレベルのスレッド処理系のみでは達成するのが難しい要件が存在する．

3.1.3 M 対 N モデル

M 対 N モデルは，N 対 1 モデルのプロセス数を CPU コア数に拡張し，各プロセスがユーザー空間でスレッドの切り替えを行うことで，複数プロセッサを活用できないという N 対 1 モデルの欠点を解消したものである (図 3.3)．これらのプロセスはメモリ空間を共有する必要があるため，一般的にはカーネルレベルのスレッドが利用される．このスレッドのことをワーカースレッドと呼ぶ．

N 対 1 モデルと同様にスレッド管理のオーバーヘッドが小さく，自由にスケジューリング方針を定められ，しかも複数プロセッサを活用することができるが，I/O 操作がブロックした際のスレッド切り替え，スレッド間の公平性を達成するのに必要になるプリエンプティブなスケジューリングなどが困難な点も N 対 1 モデルと同様である．それに加えて，すべての CPU コアを活用するには適切にワーカースレッド間の負荷分散をとらなければならない．

2.1 節で紹介した，軽量スレッドを実現している既存の処理系はすべて M 対 N モデルを利用している．これは，動的にスレッドを生成して処理を行う際には可能な限りスレッド管理を軽量化する必

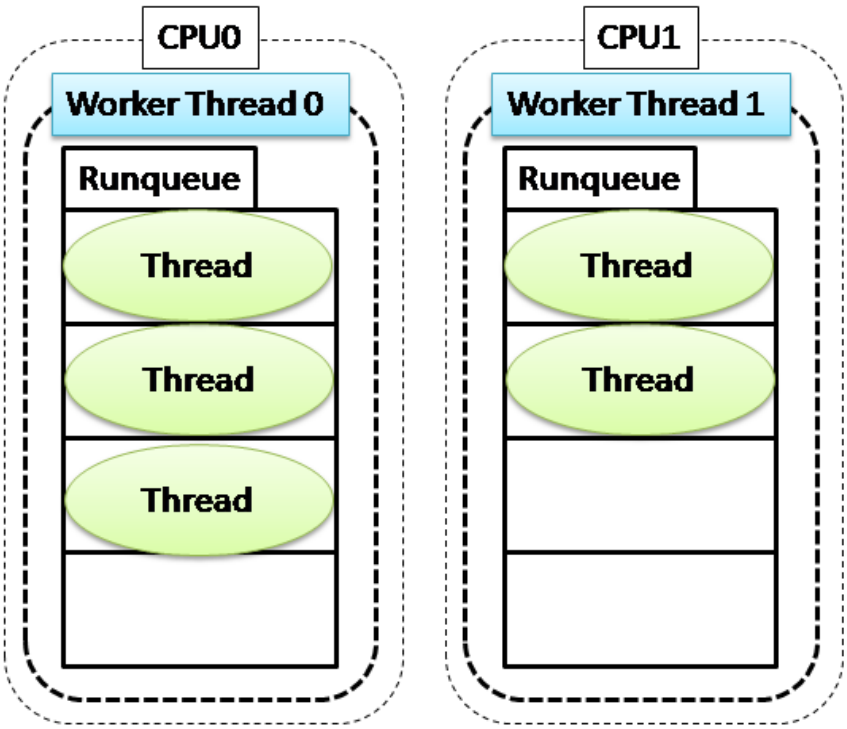
CPU0
Worker Thread 0
Runqueue
Thread
Thread
Thread
CPU1
Worker Thread 1
Runqueue
Thread
Thread

図 3.3: M 対 N モデル

要があり，しかも全てのスレッドが計算のみを行うので，公平性への配慮やプリエンプティブなスケジューリングが必要ないためである．

3.2 軽量スレッドの実現

多くの動的に並列度が変化するアプリケーションは，再帰的にスレッドを生成する形で並列化を行うことが可能である．そのため軽量スレッドを実現している既存の処理系の多くは，負荷分散に十分な数だけ再帰的にスレッドが生成されることを前提として，以下のような手法を用いた設計が行われている．

3.2.1 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が広く用いられている．これは以下の 2 つの方針にしたがって，スレッドの実行順を決定するものである．

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. スレッドを生成した直後，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，新しく生成したスレッドに制御を移す．

このスケジューリング手法では，再帰的にスレッドを生成するようなアプリケーションの処理は逐次に再帰関数を実行したのと同様の順番で実行され，ランキュー内に存在するスレッドの最大数は再帰の深さ程度に抑えられる．そのため，このスケジューリング手法は再帰的にスレッドを生成するようなアプリケーションを実行する際に良好な局所性が得られることが知られている [23].

このスケジューリング手法においては，実際に並列実行されるスレッドの数はアプリケーション全体で生成されるスレッドの数に対して少数である．そのため Multilisp や Clik の実装では，スレッドを生成するかわりに親スレッドの継続を保存し，並列実行の必要が生じた際にそれをもとにスレッドを生成することにより，実際にスレッドが生成される回数を削減し，スレッド管理のオーバーヘッドを減少させている．この手法を Lazy Task Creation[24] という．

3.2.2 ワークスチール

上に示したスケジューリング戦略では，各ワーカースレッド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，任意のワーカースレッドを一つ選択し，そのランキューから実行可能なスレッドを取り出して実行する (図 3.4)．これをワークスチールという．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，成功するまで他のワーカースレッドを選択して試行を繰り返す．

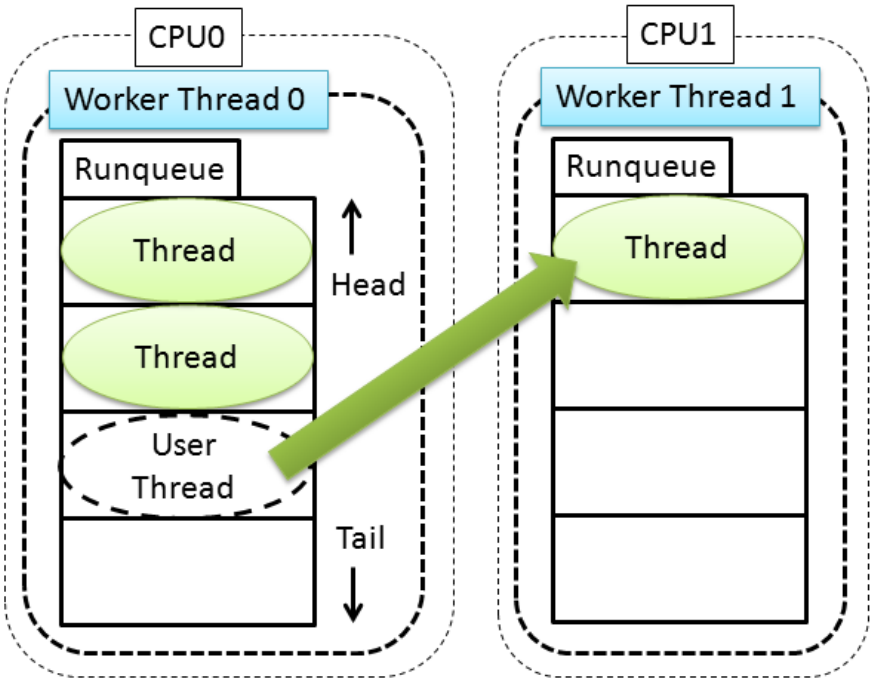
CPU0
Worker Thread 0
Runqueue
Thread
Thread
User
Thread
Head
Tail
CPU1
Worker Thread 1
Runqueue
Thread

図 3.4: ワークスチール

再帰的にスレッドが生成されるアプリケーションでは，ランキューの末尾に存在しているスレッドがのちのち多くのスレッドを再帰的に生成すると考えられる．そのため，ワークスチールの際にはランキューの末尾に存在するスレッドを取り出すことが一般的である．ワークスチールの対象スレッドはランダムに決めるのが望ましいとされている [25] が，近年は NUMA アーキテクチャの普及に伴い，データの局所性に配慮してスチール対象を決定する手法も提案されている [26].

3.2.3 ソースコード情報の利用

ユーザーレベルスレッド処理系におけるスレッド間の切り替えは，

1. 切り替え元のスレッドの構造体に，実行を継続するのに必要な情報を保存する
2. 実際に制御を切り替える
3. 切り替え先のスレッドの構造体から，継続に必要な情報を取り出す

という手順によって行われる．ここで「実行を継続するのに必要な情報」とは,「切り替え点より先で使用される可能性のあるローカル変数」に該当する．そこで，Cilk はソースコードに対してデータフロー解析を行ない，スレッド切り替えの際に保存するデータを最小限に抑えることによって，スレッド切り替えのオーバーヘッドを削減している．

3.3 ユーザーレベルによる I/O 多重化

N 対 1 モデルのマルチスレッド処理系では，スレッドがブロックするような I/O 操作を行なうと，プロセスそのものがブロックしてしまうため他の実行可能なスレッドに制御を移すことができない．しかし，これは I/O 操作がすぐに完了できない際にブロックする代わりに失敗し，固有のエラーコードを返す I/O 呼び出しであるノンブロッキング I/O と，I/O 操作可能に変化したファイルディスクリプタのみを通知することで，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 epoll などの I/O 状態の変化を通知するための機構を利用することによって解決することができる[1].

3.3.1 構成

I/O 多重化をサポートしたマルチスレッド処理系の構成は図 3.5 のようになる．

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 操作のタプルを要素としてもつリスト (I/O 待ちリスト．図には Blocked List と表記) を対応付ける．図には表記されていないが，各ワーカー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 インスタンスをもち，これによって I/O 可能になったファイルディスク

[1] ただし，ノンブロッキング I/O はパイプやソケットに対してしか有効ではないため，この手法が使えるのもパイプやソケットに限定される．

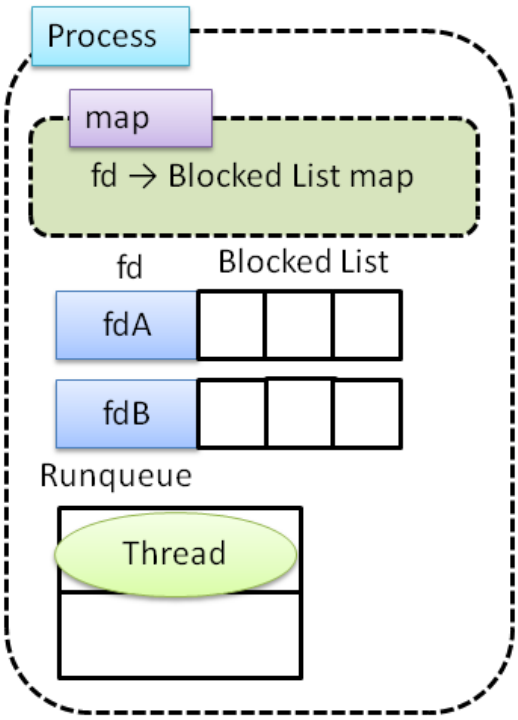

図 3.5: I/O 多重化を行うマルチスレッド処理系の構成

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，ファイルディスクリプタと I/O 待ちリストの組を格納する連想配列をもつ．

3.3.2 I/O 多重化の処理

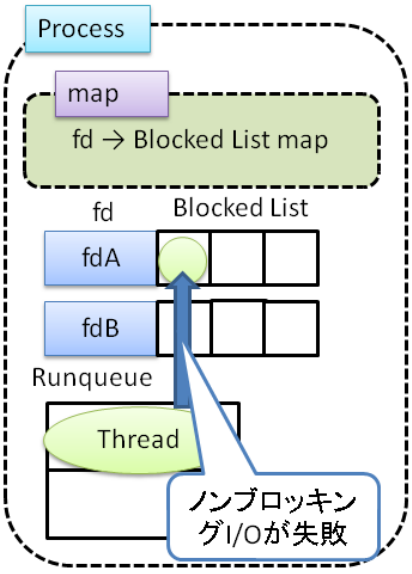

図 3.6: ノンブロッキング I/O 失敗時の処理

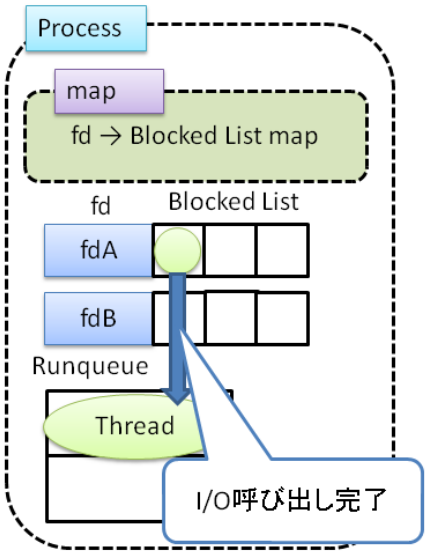

図 3.7: I/O チェック時の処理

I/O の多重化は，I/O に関係する関数をフックして以下のような処理を実行することで行われる．

3.3.2.1 ファイルディスクリプタ作成時

ファイルディスクリプタが作成された際は，I/O 呼び出しを常にノンブロッキング I/O として行うように設定した後で，それを 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．

3.3.2.2 I/O 呼び出し

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 呼び出しが失敗した場合はエラーコードをチェックし，失敗の原因が I/O のブロックだっ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 3.6).

3.3.2.3 I/O のチェック

定期的に 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．それが完了したならばそのスレッドをランキューに追加する (図 3.7).

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ー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わかりやすい.

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ースレッドに渡し，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.

第4章　MassiveThreadsの設計と実装

4.1　設計上の課題

3 章で説明した要素技術を用いて MassiveThreads の特徴である，

1. 動的に大量のスレッドを生成・破棄できるように，それらの操作を小さなオーバーヘッドで行えること．
2. 多くの CPU コアを備えた計算機上でスケーラブルに負荷分散を行えること．
3. 大量のスレッドからの I/O 呼び出しを効率的に処理できること．
4. 既存のマルチスレッド処理系からスムーズに，可能ならば再コンパイルや移植を行わずに移行できること．

を達成するためには，以下の課題を解決しなければならない．

4.1.1　既存コードと親和性の高い軽量スレッドの実現

既存の軽量スレッドを実現している処理系の多くでは，内部的にローカル変数やスタック領域に対して特殊な管理を行っている．既存のコードからのスムーズな移行をサポートするためには，これを改良し，OS 標準のスレッド処理系と同一の，リニアなスタック領域を提供する必要がある．

4.1.2　I/O 多重化のマルチコアへの拡張

3.3 節で説明した I/O 多重化手法はいずれも複数 CPU に対応していない．したがって，I/O 多重化の手法を多くの CPU コア上でスケーラブルに動作するように拡張する必要がある．また，目標が軽量性と I/O 多重化の両立であるため，I/O 多重化によってスレッドの軽量性が損なわれないように配慮して設計を行う必要がある．

4.1.3 I/O チェックを非同期に行う

高効率な I/O を実現している既存の処理系は，ほとんどのスレッドが I/O を行う場合に，I/O の多重化によってネットワークのスループットを最大限に活用することを目的として設計されている．そのため，プリエンプティブな形で I/O のチェックを行うことはない．

しかし，MassiveThreads のように計算を行うスレッドと I/O を行うスレッドが混在している場合は，何らかの形で実行中のスレッドと独立に I/O チェックを行わなければ，計算を行うスレッドが長時間 CPU を占有し，ブロックした I/O の処理が遅延してしまうことが起こりうる．

4.2 スレッドの管理

4.2.1 構成とスケジューリング戦略

MassiveThreads の基本的な構成は 3.1.3 節で説明した，M 対 N モデルのスレッド処理系である．各 CPU コアはワーカースレッドと実行可能なユーザースレッドを格納するためのランキューをもち，各ワーカースレッドがランキューの中のスレッドをスケジューリング戦略にしたが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．再帰的に大量のスレッドが生成・破棄される際にも軽量なスレッド管理を達成するために，スレッドのスケジューリングには，3.2 節で説明した LIFO なスケジューリングに，スレッドが自発的に制御を譲る際のポリシーを追加したものを用いた．したがって，スケジューリングは以下の 3 つの戦略に基づいて行われる．

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. スレッドを生成した直後，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，新しく生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．

また，ワーカースレッド間の負荷分散にはスチール対象をランダムに決定するワークスチーリングを用いた．

4.2.2 スレッドのデータ構造

スレッドはその情報を管理する構造体と，スタック領域の組によって表現するものとした．OS 標準のスレッドと同一の，リニアなスタック領域を提供する必要があるため，あるスレッドはそれが実行開始してから終了するまで，そのスレッドに割り当てられたスタック領域の中でのみ動作し，他のスタックに切り替わることはない．

スレッドを管理する構造体はスレッドの戻り値を保存する必要があるため合流処理が完了するまで再利用できないのに対して，スタックはスレッドが終了した直後から再利用可能である．そこで，スレッドの情報を管理する構造体とスタックの解放はそれぞれが不要になったタイミングで独立に

行うこととした．スタック領域はスレッドを管理する構造体と比較してはるかにサイズが大きいため，これによってスタック領域の再利用の効率を高め，メモリ確保のオーバーヘッドを減少させることができる．

4.2.3 ランキューの実装

ランキューの先頭に対する操作はスレッドの生成・破棄の度に行われるため，軽量なスレッド管理を達成するためには，特にこれのオーバーヘッドを削減することが望ましい．これを達成するため，ランキューの実装としては Java Fork/Join Framework において利用されているアルゴリズムを利用した．このランキューは，以下のような特徴をもっている．

1. 先頭に追加する操作，先頭から取りだす操作，および末尾に追加する操作は，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のみ，全ワーカースレッドが行うことができる．
3. 先頭への追加は排他制御やアトミックな命令を使わずに行うことができる．先頭からの取り出しもランキューがほとんど空である場合を除いて，排他制御やアトミック命令なしで行われる．

排他制御やアトミックな命令は特にオーバーヘッドが大きいため，これらを用いずにランキュー先頭への操作が行えることは特にオーバーヘッド削減に有効である．

4.2.4 コンテキストスイッチの実装

Linux の標準 C ライブラリにおいては，ユーザーレベルでコンテキスト切り替えを行うための API が提供されている．しかし，これはコンテキストの切り替え時にシステムコールを呼び出してシグナルマスクの退避・復元を行う．スレッドがシグナルを利用しないならばシグナルマスクの退避・復元は不要であるうえ，そのオーバーヘッドは数百サイクルにもなる．軽量なスレッド管理を達成するためには，このオーバーヘッドは到底許容できるものではない．

また，スレッド処理系の実装においては，コンテキスト切り替えの前後で継続した処理を行うデザインパターンが頻発するが，単なるコンテキスト切り替えのみではこれを小さなオーバーヘッドで実装することも困難である．

例えばあるスレッドが他のスレッドに制御を自発的に譲る際には,「ランキューの先頭にあるスレッドに対して制御を切り替え，その後以前実行されていたスレッドをランキューの末尾に格納する」という処理が必要だが，単純なコンテキストスイッチのみでこれを実装することは難しい．一旦別のコンテキストにスイッチし，そこでランキューにスレッドを格納してから改めてランキュー先頭のスレッドにスイッチする，という手法を用いれば不可能ではないが，2 回のコンテキストスイッチが必要で，オーバーヘッドが大きくなる．

そこでコンテキストスイッチのための関数は，必要最小限のレジスタの退避・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．このコンテキストスイッチ関数は，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 を備えている．これによって，前述のようなデザインパターンの実装をダミーのコンテキストを用いず，1 回のコンテキストスイッチのみで行うことが可能になった．

4.2.5 メモリ確保・解放の高速化

スレッドの生成・破棄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．

実際にはスレッドを管理する構造体とスタック領域は別々に管理されるため，ワーカースレッドはそれぞれのためのフリーリストを 1 つずつもっている．

さらに，スレッドを管理する構造体や，スタック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一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ものとした．

4.3 I/O 多重化

4.3.1 構成

I/O 多重化については，3.3 節で説明された手法をベースとし，それをマルチコア上で動作するように拡張を行なった．具体的な I/O 多重化部の構成を図 4.1 に示す．

各ファイルディスクリプタに対応付けられている I/O 待ちリストは複数のワーカースレッドに分散配置し，各ワーカースレッドに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 インスタンスと，epoll が I/O 可能に変化したファイルディスクリプタしか通知しないことに起因するデッドロックを回避する必要ために使われる予備リストを 1 つもたせる．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，全ワーカースレッドで共有する．

4.3.2 I/O 多重化の処理

4.3.2.1 ファイルディスクリプタ作成時

ファイルディスクリプタが作成された際は，fcntl [1]関数を用い，ノンブロッキング I/O を行うようにファイルディスクリプタを設定し，ワーカースレッドの epoll インスタンスと連想配列に対する登録を行う．ただし，特定のワーカー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ースレッドはランダムに決定する．

[1]ファイルディスクリプタの状態を取得，設定する関数

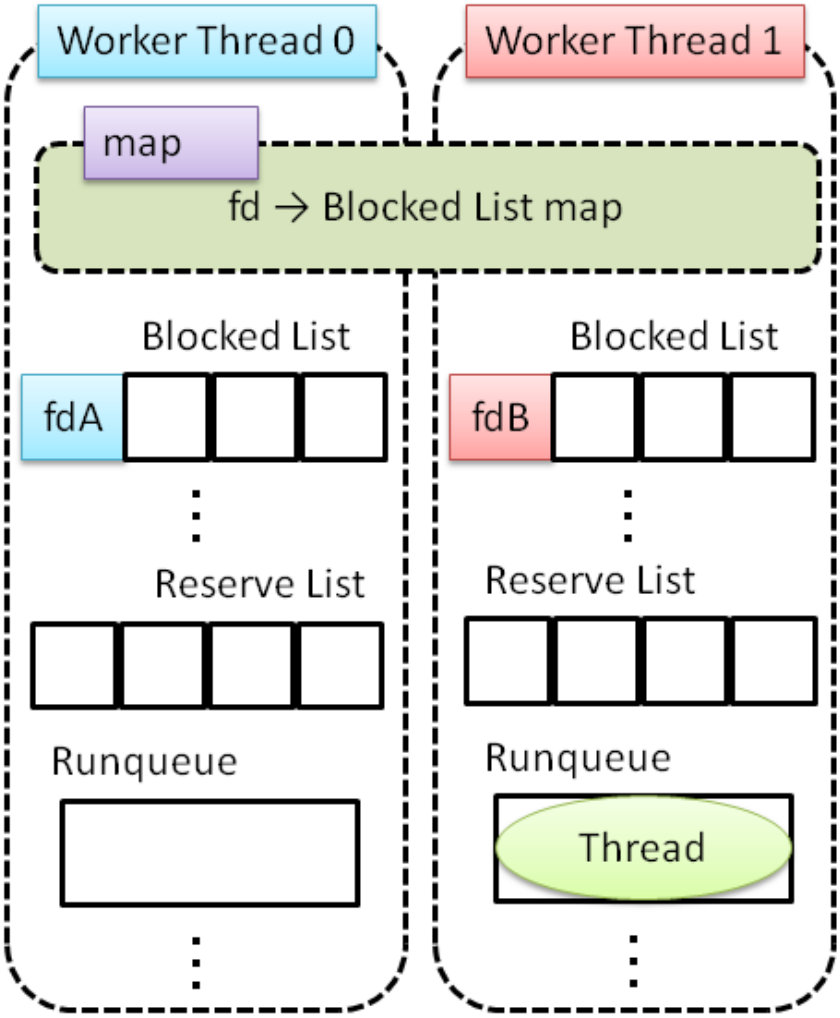

図 4.1: I/O 多重化部の構成

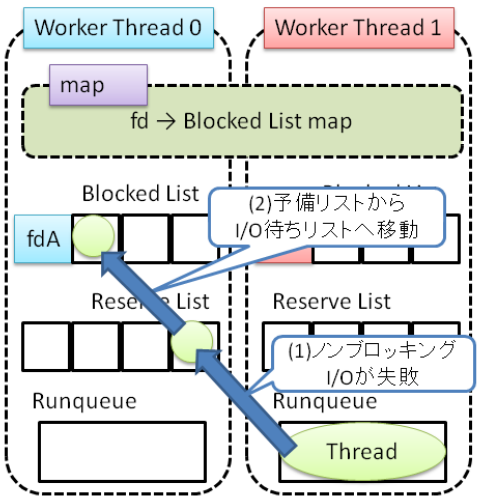

図 4.2: ノンブロッキング I/O 失敗時の処理

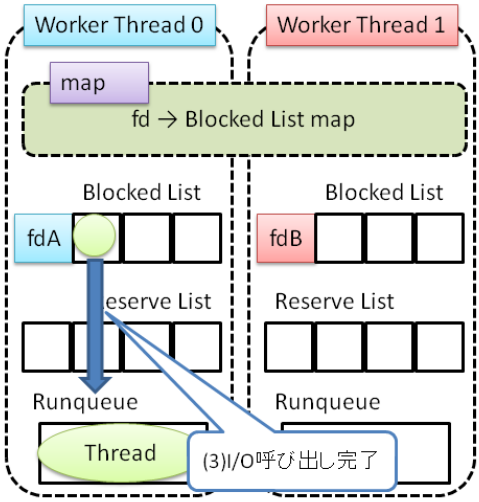

図 4.3: I/O チェック時の処理

4.3.2.2 I/O 呼び出し

I/O 呼び出しをフックし，ブロックして失敗したならば連想配列から I/O 待ちリストを検索し，そこにスレッドと 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．ただし，I/O 対象のファイルディスクリプタが他のワーカースレッドに登録されている場合は，直接 I/O 待ちリストに登録するのではなく，ファイルディスクリプタが登録されているワーカースレッドの予備リストに対してスレッドを追加する (図 4.2(1))．

4.3.2.3 I/O のチェック

I/O のチェックおよびスレッドの再開処理は各ワーカースレッドで独立に行う．このとき同時に予備リストの中のスレッドを取り出して I/O 可能であるかチェックし，I/O 可能でなければそれを対応するファイルディスクリプタの I/O 待ちリストに追加し (図 4.2(2))，I/O 可能であれば再開する．

また，スレッドを再開する際には，そのスレッドをもともとスレッドが実行されていたワーカースレッドではなく，自身のランキューに追加する (図 4.3)．これによって I/O 呼び出しを行うスレッドがそのファイルディスクリプタの I/O チェックを行うワーカースレッドに集中し，スレッドがワーカー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．

4.3.3 I/O 待ちリストのスケーラブルな検索

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連想配列に対しては I/O 呼び出しがブロックするたびに検索が発生するため，検索操作は複数ワーカースレッドからの同時アクセスに対してスケーラブルでなければならない．

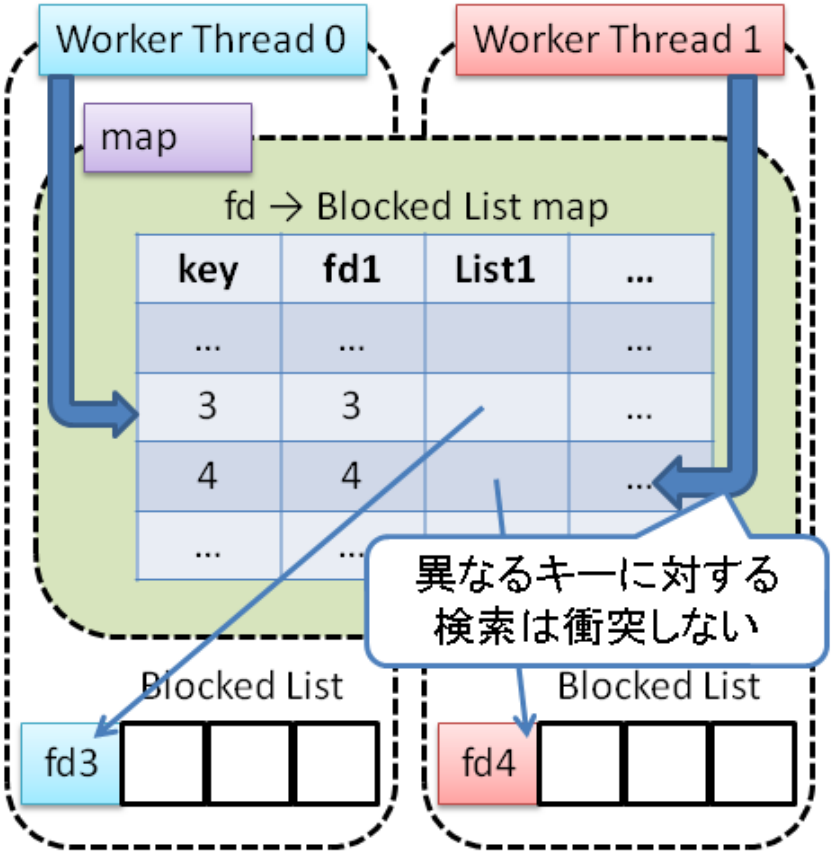

図 4.4: ハッシュテーブルによるスケーラブルな I/O 待ちリストの検索

しかし，I/O 呼び出し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ー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，異なるキーをも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 (図 4.4)．

4.3.4 I/O チェックのタイミング

I/O チェックのオーバー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 I/O 処理が遅延され実行可能なスレッド数が減少するため，これも性能低下の原因となりうる．そのため，MassiveThreads では，I/O チェックの方針として

- ワーカースレッドがアイドル状態である間の I/O チェック
- タイマーによる I/O チェック
- あるファイルディスクリプタが I/O 可能に変化した際の I/O チェック

の 3 つを提供しており，アプリケーションはこれらを組み合わせて利用することができる．

4.4 アプリケーションとのインターフェース

MassiveThreads は共有ライブラリとヘッダファイルの 2 種類のインターフェースを備えている．前者は pthread と同様のインターフェースをもち，Linux 環境においては LD_PRELOAD 環境変数を利用することで既存のアプリケーションに対しても移植や再コンパイルを行うことなく適用することが可能である．

ヘッダファイルのインターフェースは既存のアプリケーションにそのまま適用することはできないが，コンパイル時に処理系のコードをインライン化することができるため，より小さなオーバーヘッドで MassiveThreads を利用することができる．

第5章　マイクロベンチマーク

5.1　軽量性の評価

軽量性の評価は，以下のような手順によって再帰的にスレッドを生成することにより，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 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生成する．
3. 生成した 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．

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 8354(2.2GHz)8 ソケットからなる 32 コアの計算機を用い，スレッドを生成した場合の実行時間と再帰を用いて同様の計算を行なった場合の実行時間をもとにスレッドあたりのオーバーヘッドを以下の式で計算した．

$$\text{オーバーヘッド} = \left(\text{実行時間} - \frac{\text{再帰版の実行時間}}{\text{使用した CPU コア数}}\right) \div \text{作成されたスレッド数}$$

共有ライブラリ版と，ヘッダファイル版のオーバーヘッドを比較したものを図 5.1 に，ヘッダファイル版 MassiveThreads のオーバーヘッドを Cilk および Intel Threading Building Blocks と比較した結果を図 5.2 に示す．

ヘッダファイル版は，共有ライブラリ版と比較してオーバーヘッドが 15 ナノ秒程度削減されていることがわかる．また，ヘッダファイル版 MassiveThreads のオーバーヘッドは，1 コアの場合 55 ナノ秒，32 コアすべてを使う場合でも 60 ナノ秒程度，共有ライブラリ版は 1 コアの場合 71 ナノ秒，32 コアすべてを使う場合に 76 ナノ秒程度であり，いずれも既存の処理系と比較して遜色ない値である．OS 標準のマルチスレッド処理系では，1 スレッドの生成・破棄に 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．また，いずれの処理系も使用する CPU コア数が増加するにしたがってオーバー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 5.1 に示す．比較対象として，評価を行なった 4 種類の処理系の中で最もオーバーヘッドの小さかった Cilk のオーバーヘッドも併記した．

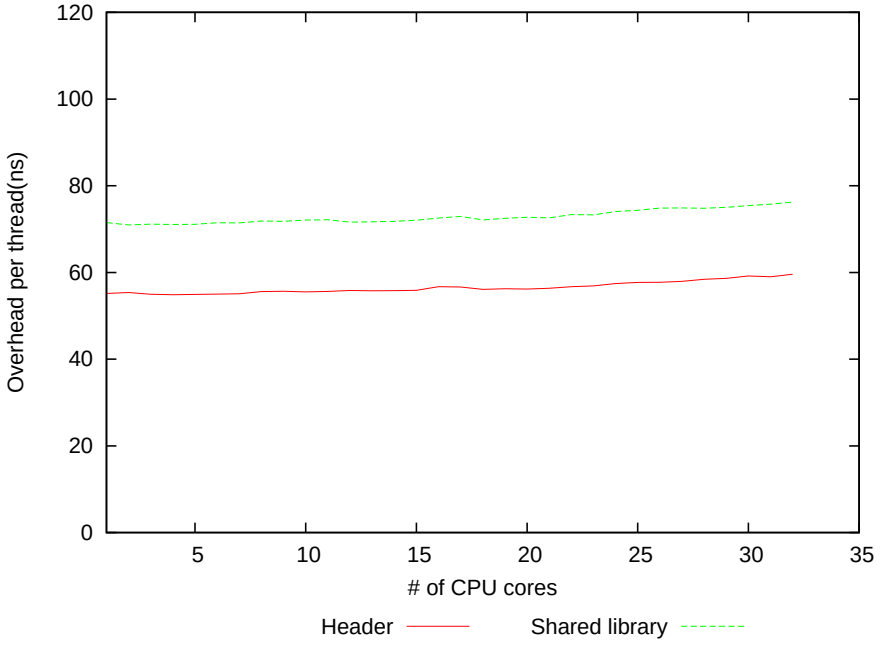

図 5.1: ヘッダファイル版インターフェースの効果

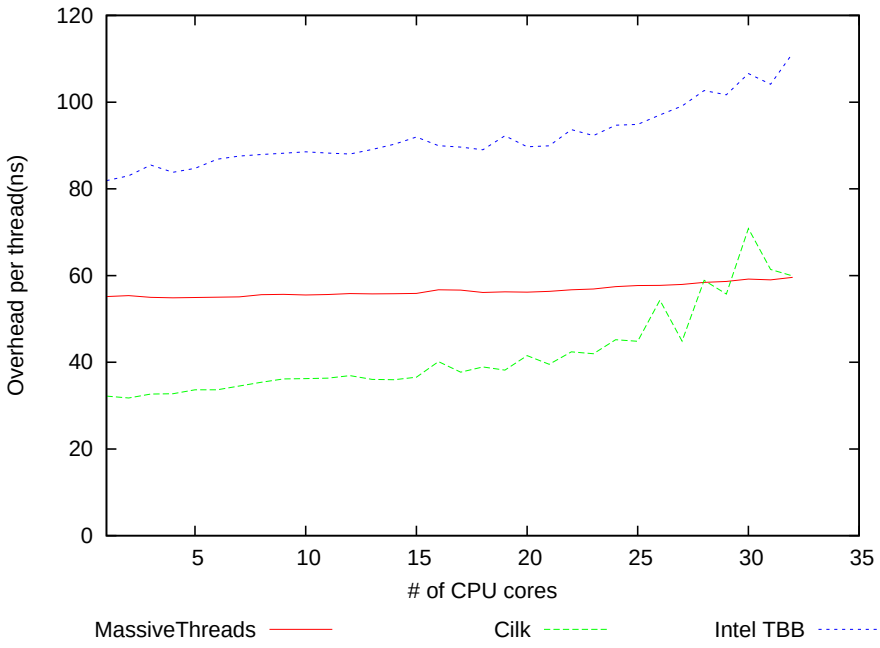

図 5.2: 既存の処理系との比較

表 5.1: Cilk と MassiveThreads のオーバーヘッドの内訳

	MassiveThreads(Shared lib)	MassiveThreads(Header)	Cilk
スタック確保・解放	11.81	12.08	14.80
ランキュー操作	23.74	24.49	2.09
コンテキスト切り替え	22.65	22.55	2.03
合流処理	34.67	32.79	37.85
その他 (関数呼び出しなど)	33.70	11.86	5.57
合計	126.57	103.76	62.23

MassiveThreads は特にコンテキスト切り替えと，ランキュー操作のオーバーヘッドが Cilk と比較して大きくなっていることがわかる．Cilk のコンテキスト切り替えが高速な理由は，スレッド生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) を退避し，関数呼出しによって切り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し，さらにスタックを切り替えたうえで切り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，Cilk のランキュー操作のオーバーヘッドが小さい理由は，Cilk はスレッドの親子関係が定まっており，あるスレッドが終了した後は必ずその親が実行されるため，ランキューから次に実行するスレッドを取り出す必要がなく，そのぶんのランキュー操作を省略できるためである．

MassiveThreads の共有ライブラリ版では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ーヘッドも特に大きくなっている．これは，MassiveThreads のヘッダファイル版や 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ー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考えられる．

5.2 負荷分散能力の評価

負荷分散能力の評価には，UTS Benchmark [27] を用いた．これは，ハッシュ関数をもとに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]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 回分のオーバーヘッドを被る．

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．このデータセットは $q = 0.124875$，$m = 8$ で，最大深さ 1572，約 411 万ノードをもつ木構造を生成する．

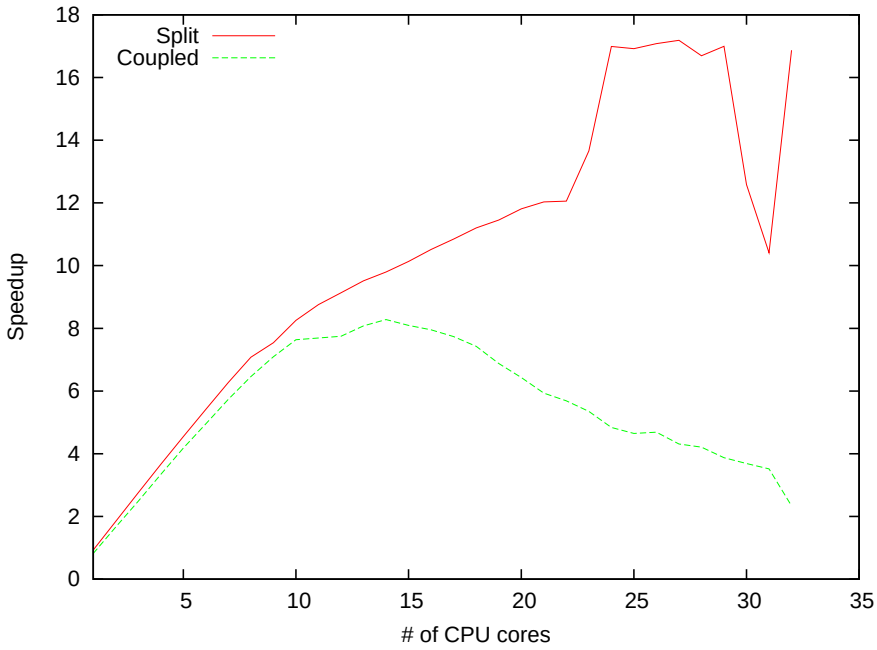

図 5.3: スレッド構造体とスタックのメモリ管理を分離した効果

軽量性の評価と同じ環境を用い，木のノード 1 つあたり 1 つのスレッドを生成する形で UTS Benchmark を並列化した際の実行性能を図 5.3 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．Split はスレッド構造体とスタックの解放のタイミングを分離した場合，Coupled はスレッド構造体とスタックをまとめて管理している場合の性能向上率を表している．MassiveThreads の性能向上率を Cilk，Intel Threading Building Blocks，および GCC の OpenMP Task と比較したものを図 5.4 に示す．

スレッド構造体とスタック領域をまとめて管理し，同じタイミングで解放を行った場合は，スタック領域の再利用が遅れることが原因で多くのスタック領域が確保され，その際のページフォルトのオーバーヘッドがボトルネックになって CPU コア数が多い領域で性能が大きく低下している．

それに対してスレッド構造体とスタック領域を別々に解放する場合は，スタック領域が再利用される頻度が増加したため，ページフォルトがボトルネックになることもなく，32 コアまでゆるやかに

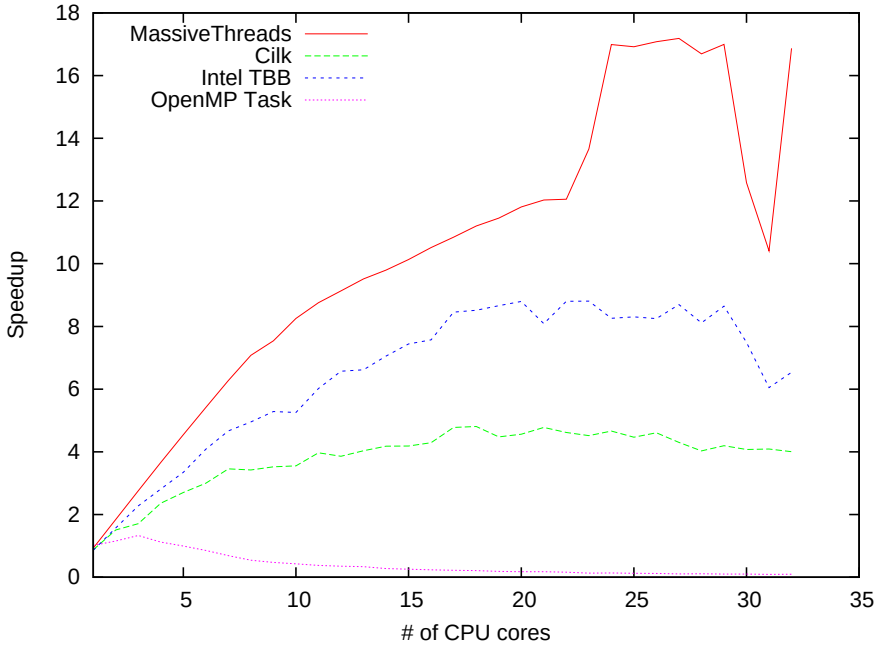

図 5.4: UTS Benchmark の結果

性能が向上し，逐次の 16 倍程度の性能向上が得られている．また，MassiveThreads は既存の処理系はと比較してより大きな性能向上率を示している．

5.3 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

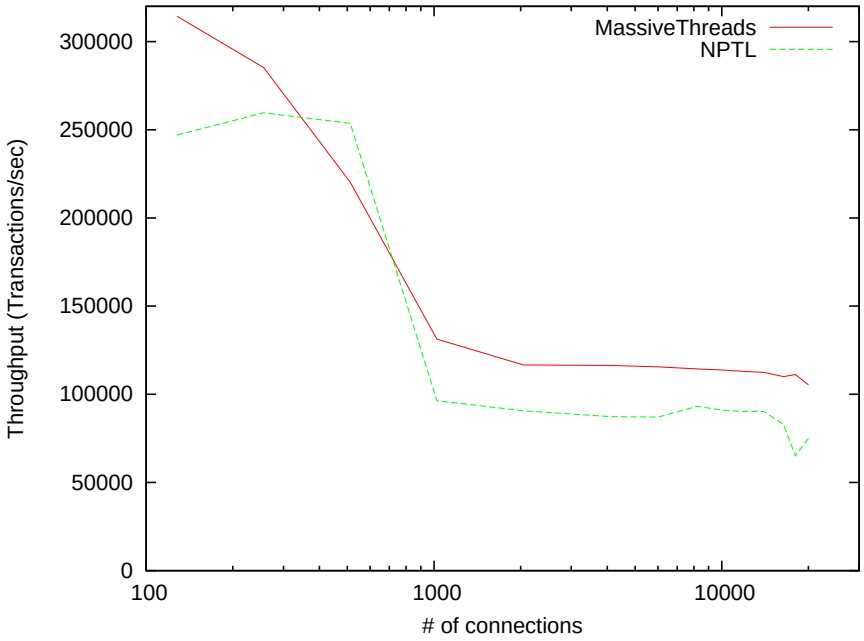

図 5.5: 全部の接続が絶えず通信する場合のスループット

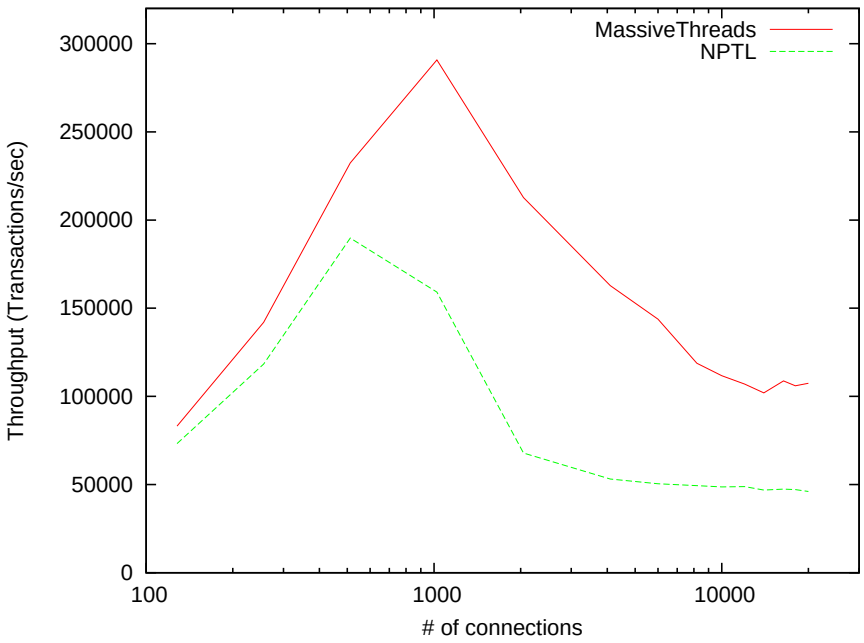

図 5.6: 8 分の 1 の接続が同時に通信を行う場合のスループット

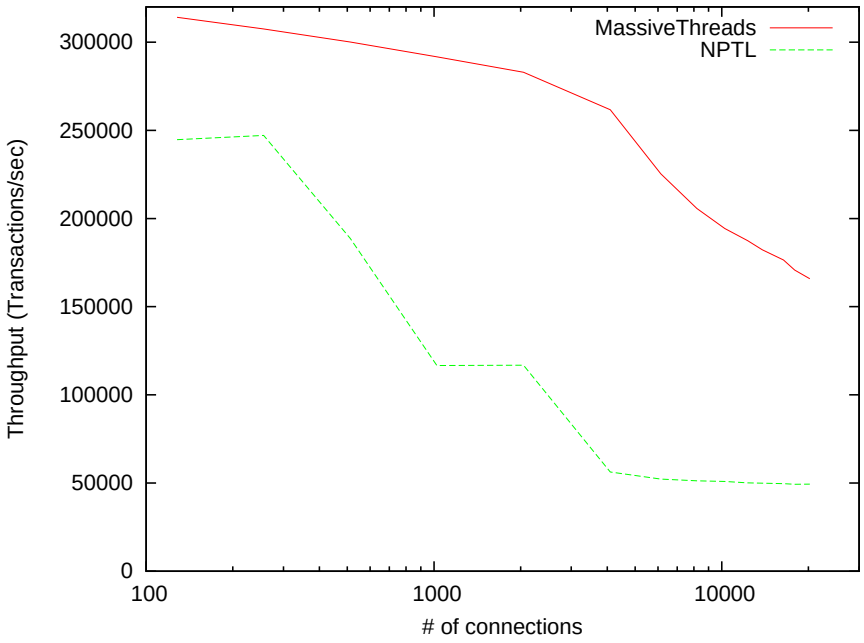

図 5.7: 128 個の接続が同時に通信する場合のスループット

の 3 通りを用いた．(1) はクラスタ内部などの，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) は分散環境や高並列サーバなどのバンド幅が小さく，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なったものの 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．同時通信数の制限は，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り，それらの中からランダムに 1 つの接続を選択して送受信を行うことを繰り返して実現している．

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．

結果を図 5.5，図 5.6，図 5.7 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したが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，全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ーヘッドと 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 時間をアプ

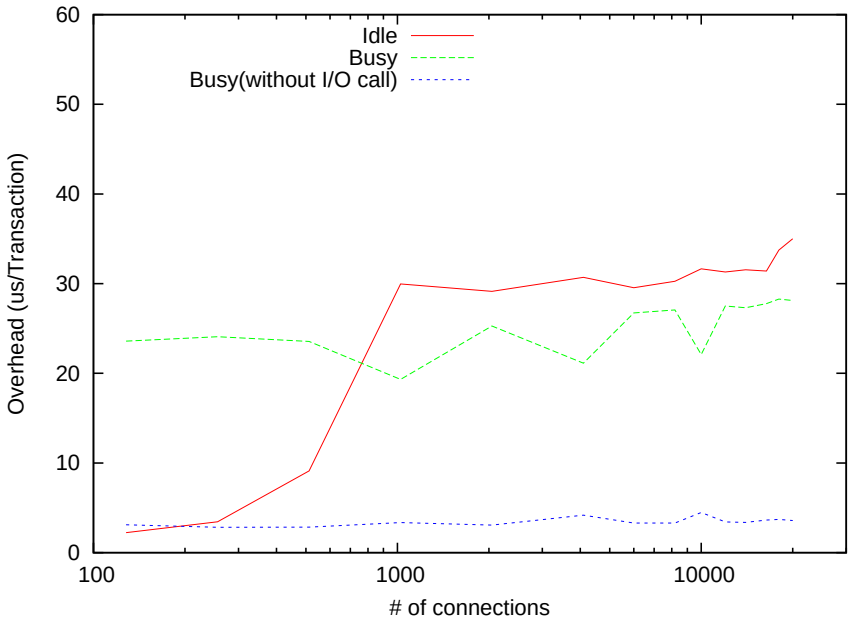

図 5.8: 全ての接続が同時に通信する場合の CPU 時間の内訳

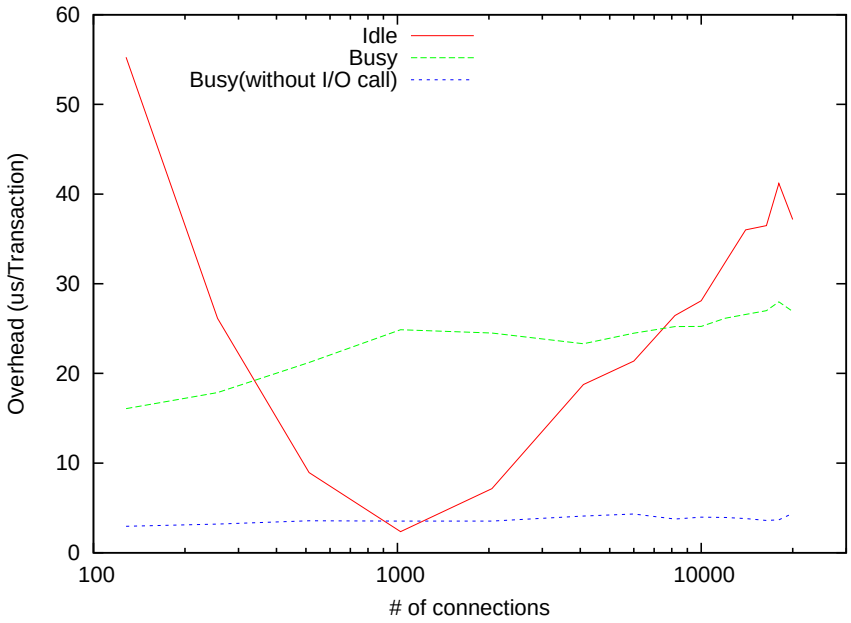

図 5.9: 8 分の 1 の接続が同時に通信を行う場合の CPU 時間の内訳

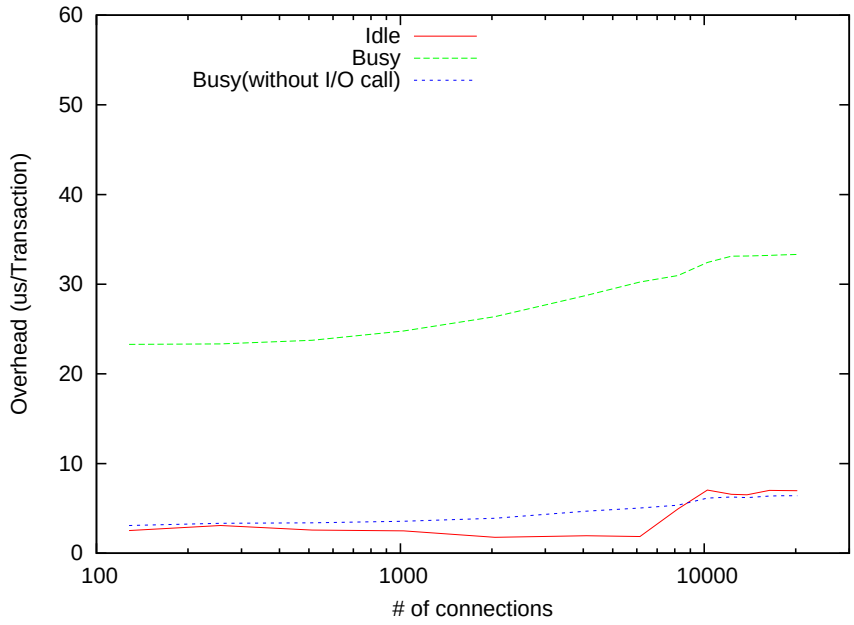

図 5.10: 128 個の接続が同時に通信する場合の CPU 時間の内訳

リケーションの進捗に貢献している部分 (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[2]．これらの分類はそれぞれ処理系のオーバーヘッドと I/O 待ち時間に対応している[3]．

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 5.8，図 5.9，図 5.10 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．

全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 (=処理系のオーバーヘッド) が CPU 時間の大半を占めているが，接続数が増加するにしたが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したがって 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，処理系のオーバーヘッドの大半を 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ー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

[2] この測定においては，同時通信数の制限をクライアント側がラウンドロビンに各接続に対して通信を行うことで実現しているため，図 5.5，図 5.6，図 5.7 の測定条件と実際にはわずかに異なる条件になっている．ただし，いずれの条件においても MassiveThreads のスループットの傾向に大差はないため，オーバーヘッドの傾向にも差異はあまりないものと考えられる．

[3] 厳密に言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．

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ーヘッド，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.

第6章　適合格子細分化法による性能評価

6.1　問題設定

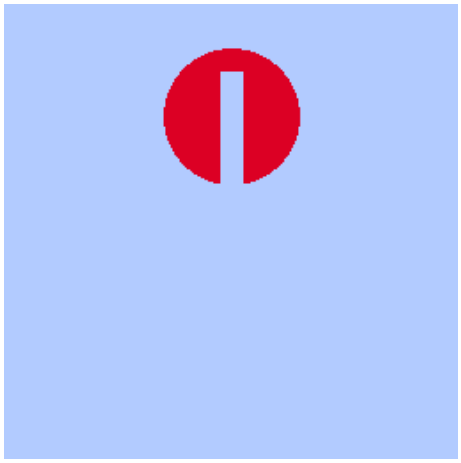

図 6.1: 初期状態

性能評価の対象には Zalesak 問題 [28] の移流計算を用いた．これは図 6.1 に示すようなコの字形状の初期濃度を角速度一定の流れのもとで回転させ，一回転したのちの誤差を評価するものである．

移流計算は時間を離散化し，各時間における格子点の濃度を周辺の格子点の濃度から補間することによって行われる．本実験では，計算手法として Cubic セミラグランジュ法を用いた．これは，格子点の周辺 25 点の中から流れの方向に応じて 16 点を選択し，それらを利用して 5 回の 3 次補間を行い，次の時間の濃度を求めるものである．

移流計算では次の時間の濃度を決定するのに周辺の濃度を利用するため，時間が経過するにしたがって周辺の格子との間に拡散が生じ，特に濃度変化が急峻な領域ではこれが誤差の要因となる．これはより細かい格子と細かいタイムステップを用いることで抑えることができるが，計算量や必要とするメモリがそれにしたがって大きく増加してしまう．

6.2 適合格子細分化法の導入

6.2.1 計算の手順

しかし，本問題では濃度の勾配が大きい領域が領域全体に占める割合が小さいため，適合格子細分化法を用い，濃度の勾配が大きい領域にのみ細かい格子を割り当てて計算を行うことで計算時間やメモリ消費の増加を抑えながら計算精度を向上させることができる．

適合格子細分化法における，時間を t_0 から $t_0 + \Delta t$ だけ進める処理は以下の手順を再帰的に繰り返すことによって行われる．細かい格子の時間を $\frac{\Delta t}{2}$ ずつ進めるのは，CFL 条件[1] を満たす必要があるためである．

1. 粗い格子のタイムステップを t_0 から $t_0 + \Delta t$ に進める
2. t_0 における粗い格子の値から細かい格子の境界条件を求める
3. 細かい格子のタイムステップを t_0 から $t_0 + \frac{\Delta t}{2}$ に進める
4. t_0 と $t_0 + \Delta_t$ における粗い格子の値から $t_0 + \frac{\Delta t}{2}$ における細かい格子の境界条件を求める
5. 細かい格子のタイムステップを $t_0 + \frac{\Delta t}{2}$ から $\frac{\Delta t}{2}$ に進める
6. 細かい格子の値から粗い格子の値を求める

6.2.2 細分化の指標

細分化を行う指標としてはレルナーの指針値 [29] を用いた．これは U_i を格子点 i における物理量としたとき，以下の式によって E を求め，これと閾値とを比較して細分化を行うか定めるものである．δ は任意の定数である．

$$E = \frac{|U_{i-1} - 2U_i + U_{i+1}|}{|U_{i-1} - U_i| + |U_i - U_{i+1}| + c}$$

$$c = \delta(|U_{i-1}| + 2|U_i| + |U_{i+1}|)$$

本実験では $\delta = 0.05$ として X 軸と Y 軸の 2 方向に対して下の式を計算し，大きい方の値が閾値 (本実験では 0.1 とした) を上回った際にその格子を細分化するものとした．

[1] 流体計算を安定に行うために必要な条件．格子の大きさを Δx，タイムステップを Δt，流速を v としたとき，$\frac{\Delta x}{\Delta t} > v$

6.3 並列化

適合格子細分化法を並列化するにあたっての大きな課題として，負荷分散がある．適合格子細分化法においては，各格子点の計算量が細分化の度合いに応じて大きく異なり，さらに細分化される領域が時間発展にしたがって動的に変化する．したがって，静的な領域分割で負荷分散を維持することはほぼ不可能で，何らかの手法を用いて動的に負荷分散をとる必要がある．本実験においては，MassiveThreads を使って並列化を行い，動的な負荷分散は処理系の機能を利用する．

MassiveThreads による並列化は，計算領域を 16×16 格子点を単位として細分化し，それぞれの領域に 1 スレッドを割り当てることによって行なった．適合格子細分化法の計算にはグローバルな同期をとる必要のある箇所が存在しているが，これは同期の直前で全スレッドの終了待ちを行い，その後スレッドを起動しなおすことによって実現している．

負荷分散の効果や適合格子細分化法の効果を確認するためにの比較対象としては，

- pthread を用い，領域を 1 次元に分割して並列化したもの
- 適合格子細分化法を使わずに最も細かい格子を並列計算したもの

の 2 種類を利用した．

6.4 実験結果と考察

6.4.1 実験環境と計算条件

実験環境としては，Opteron 8354(2.2GHz)8 ソケットからなる 32 コアの計算ノードと，Xeon X5680(3.3GHz)6 コア (Hyper-Threading により 12 スレッド) 2 ソケットからなる 12 コア (24 スレッド) の計算ノードの 2 種類を用いた．以下ではそれぞれをノード A，ノード B と呼称する．また，コンパイラには GCC(4.4.1) を用いた．

計算条件としては，最も粗い格子の大きさを 256×256 に，総タイムステップ数を 2560 に設定し，細分化は 2 レベルまで行なうものとした．

6.4.2 解の精度

表 6.1: 誤差の自乗平均平方根

粗い格子のみ	2 レベル AMR	細かい格子のみ
6.000×10^{-2}	4.114×10^{-2}	3.566×10^{-2}

粗い格子のみを用いて計算した場合，2 レベルの適合格子細分化法を用いた場合，最も細かい格子のみを用いて計算した場合の 3 種類の条件について，誤差の自乗平均平方根を表 6.1 に示す．適合格

図 6.2: 256×256 の格子における解

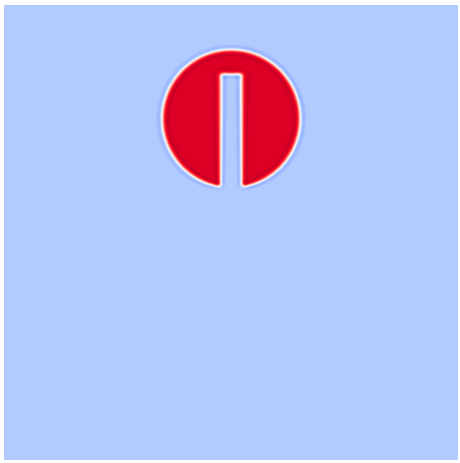

図 6.3: 1024×1024 の格子における解

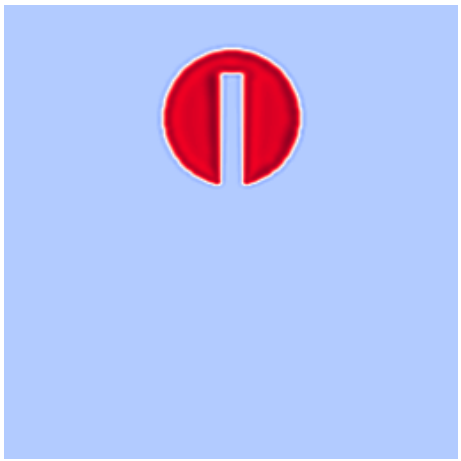

図 6.4: 256×256 の格子に 2 レベルの適合格子細分化法を適用した際の解

子細分化法を用いることで荒い格子を用いた場合と比較して誤差が減少し，最も細かい格子のみを利用したものに大きく近づいていることがわかる．

図 6.2 は 256×256 の格子を用いてこの問題を解いた解，図 6.3 は 1024×1024 の格子を用い，図 6.2 の 4 倍のタイムステップ数を利用して解いた解，図 6.4 は 256×256 の格子を用い，適合格子細分化法を用いて密度の勾配の大きい領域に細かい格子と細かいタイムステップを割り当てて解いた解である．図 6.4 のほうが図 6.3 と同程度にまで周辺への拡散が抑えられていることがわかる．

6.4.3 実行時間

表 6.2: 実行時間

	ノード A	ノード B
MassiveThreads	21.29	17.35
pthread	26.03	24.59
NoAMR(OpenMP)	34.31	53.98

ノード内のすべてのコア (含ハイパースレッディング) を使用した場合の各実装の実行時間を表 6.2 に示す．適合格子細分化法を用いることによって，細かい格子を計算するよりも計算量を削減できている．また，MassiveThreads の動的負荷分散を利用することで，静的に領域を分割する場合と比較してノード A においては 18%，ノード B においては 30%程度計算時間を削減することができた．

6.4.4 性能向上率

pthread 版を 1 コアのみ使用して実行した際の実行時間を基準にした際の性能向上率を図 6.5，図 6.6 に示す．いずれの計算環境においても，MassiveThreads を用いたものが pthread より大きな性能向上率を示している．ノード B においては，12 スレッド以上のワーカースレッドを使用するとハイパースレッディングによって 1 つの物理コアに 2 スレッドが割り当てられるようになるため，12 スレッドを境に性能向上の度合いが小さくなっている．

6.4.5 負荷分散の効率

使用するコア数を変化させた際の負荷分散の効率を図 6.7，図 6.8 に示す．負荷分散の効率は以下の式を用いて計算した．ただし N_C は使用された CPU コア数 (スレッド数) を，$n_u(t,i)$ はタイムステップ t において i 番目のコアが 3 次補間を実行した回数を表している．

$$\sum_t \max_i (n_u(t,i)) \div \left(\sum_t \sum_i (n_u(t,i)) \div N_c \right)$$

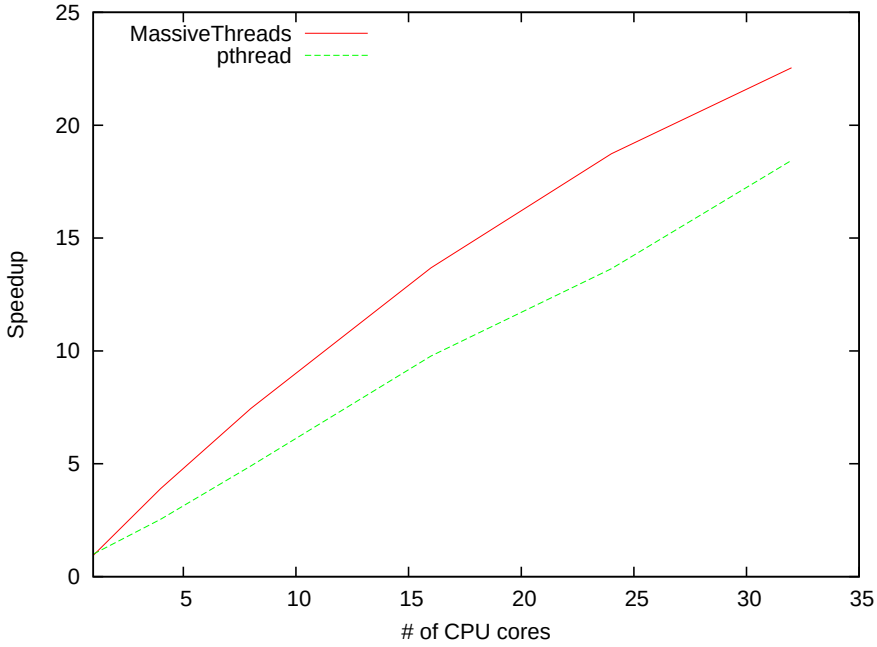

図 6.5: ノード A における性能向上率

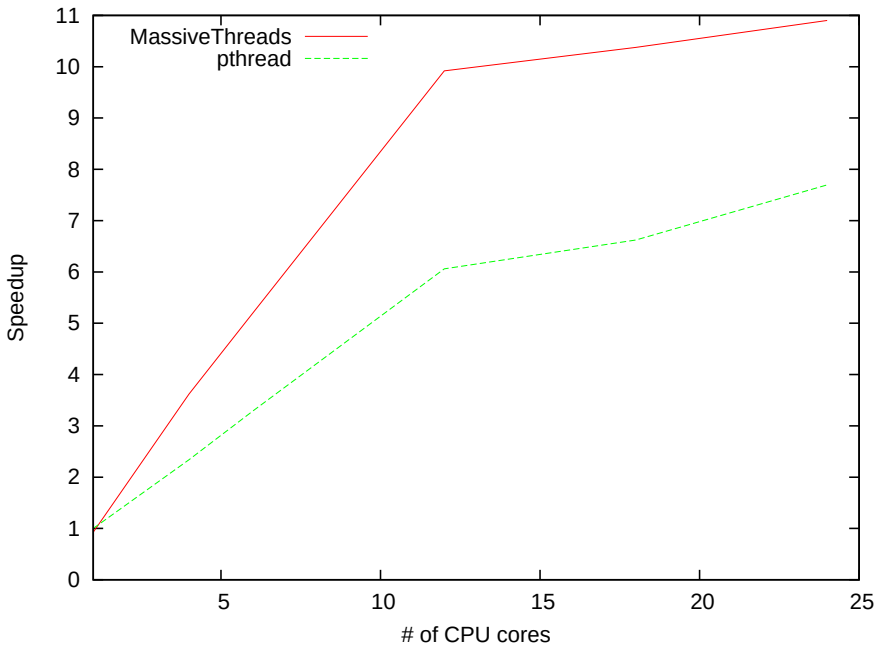

図 6.6: ノード B における性能向上率

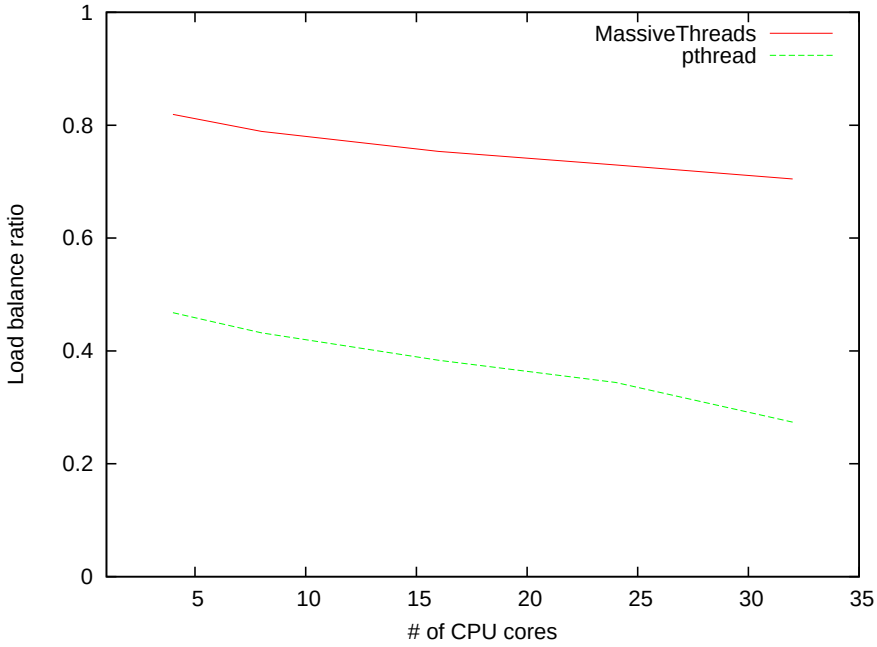

図 6.7: ノード A における負荷分散の効率

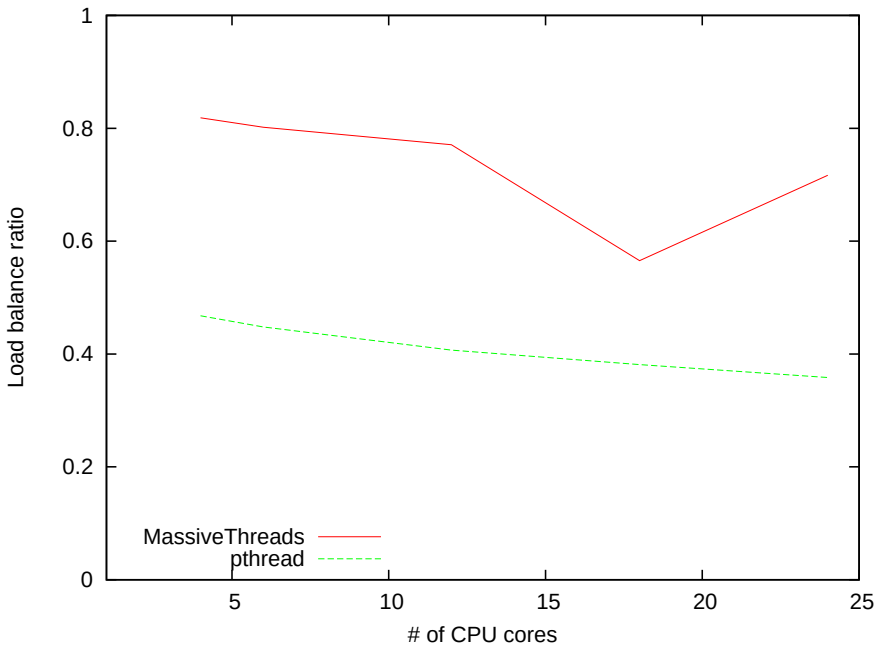

図 6.8: ノード B における負荷分散の効率

この値は直感的には「負荷分散の乱れに起因する性能低下の度合い」を表しているといえる．ただし，本実験で用いた実装においてはデータ構造に起因するオーバーヘッドが無視できず，計算が必ずしも支配的ではないため，この値のみが直接性能向上率を決定づけているわけではない．

いずれの環境においても，MassiveThreads 版の実装は pthread 版よりも良好な負荷分散を示している．しかしその傾向は計算ノードによって異なり，ノード A においては負荷分散の効率が徐々に低下していくのに対し，ノード B では 18 スレッドにおいて負荷分散の効率が一旦落ち込んでいる．

この現象はハイパースレッディングを考慮することによって説明できる．12 コア 24 スレッドの環境上で 18 個のワーカースレッドを起動すると，6 つのコアがハイパースレッディングによって 2 スレッドを同時に実行し，残りの 6 コアは 1 つのスレッドのみを実行する．しかし，ハイパースレッディングにより 2 スレッドを実行している物理コアのスレッドは 1 スレッドのみを実行している物理コアのスレッドと比較して，1 スレッドあたりの実行性能が低い．すなわち，実行性能の異なる 2 種類のワーカースレッドが混在することになる．そして，これらのワーカースレッド間でアイドル状態のスレッドを最小限にするよう動的な負荷分散がとられた結果，1 つのコアを占有しているスレッドがより多くの計算を行ったことに起因すると考えられる．

第7章　おわりに

7.1　結論

次世代のコンピュータの性能を十分に引き出すアプリケーションを高生産に記述するための基盤ソフトウェアとして，(1) 軽量なスレッド管理 (2) マルチコア上でのスケーラブルな負荷分散 (3) 大量のスレッドによる I/O の効率的な実行 (4) 既存の処理系に適用できるインターフェースの 4 つの特徴をもつマルチスレッド処理系，MassiveThreads を提案する．

上に述べた 4 つの特徴を達成するため，MassiveThreads は，多くのスレッドを軽量に管理し，動的な負荷分散をとれる既存の処理系の手法と，I/O を多重化して効率的に処理できる既存の処理系の手法を，既存のコードとも高い親和性をもち，なおかつ多くの CPU コアをもつ計算機上でスケーラブルに動作するよう拡張して組み合わせて実装されている．

マイクロベンチマークによって，処理系のオーバーヘッド，負荷が不均衡な計算におけるスケーラビリティ，およびネットワーク 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ない小さいオーバーヘッドや良好な動的負荷分散の能力，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することができた．

さらに，実アプリケーションにおける MassiveThreads の有用性を評価するために，適合格子細分化法による移流計算を MassiveThreads を用いて実装し，多くの CPU コアをもつ 2 種類の計算機を用いて実行性能を評価した結果，いずれの環境においても，MassiveThreads の軽量スレッドによる動的な負荷分散によって静的な領域分割を行う場合よりも良好な負荷分散を行うことができ，実行性能の向上がみられることを確認できた．

7.2　今後の課題

7.2.1　局所性を意識した実装の改善

現在の MassiveThreads の実装は局所性を意識したスケジューリングを行っていない．そのため局所性を意識したスケジューリングや，ヒントを与えるユーザーインターフェースの実装を行うことにより，よりキャッシュや NUMA アーキテクチャなどのメモリ階層を活用できるようになると考えられる．

7.2.2 計算と I/O 処理が混合する場合のスケジューリング

軽量スレッドを実現している処理系はほとんどのスレッドが計算を行う場合に，ノード内の CPU コアをすべて活用することを目的として設計されており，高効率な I/O を実現している処理系はほとんどのスレッドが I/O を行う場合に，I/O の多重化によってネットワークのスループットを最大限に活用することを目的として設計されている．いずれの要件も単独では実行可能なスレッドを間断なく実行していけば達成できるため，これら既存の処理系はランキューから取り出した順にスレッドを実行するような比較的単純なスケジューリングポリシーを用いている[1]．

しかし，MassiveThreads の目標は計算を行うスレッドと I/O を行うスレッドが混在している状況において，アプリケーション全体として最適な性能を達成することである．前述のような単純なスケジューリングでは，これを達成することはできないと考えられるため，アプリケーション全体として最適な性能を達成することが可能なスケジューリング手法を検討する必要がある．

[1] Capriccio は逼迫しているリソースを解放する可能性の高いスレッドを優先して実行する，という高度なスケジューリングを行うが，すべてのスレッドが I/O を行うスレッドであることがそもそもの前提である．

参考文献

[1] Intel Website. Intel AVX, Intel Software Network. http://software.intel.com/en-us/avx/.

[2] Cray. Cray XMT Programming Environment User's Guide. http://docs.cray.com.

[3] Jack Dongarra, Pete Beckman, Franck Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Patrick Aerts, Anne Trefethen, and Mateo Valero. The International Exascale Software Project Roadmap. University of Tennessee EECS Technical Report, May 2010.

[4] Marsha J Berger and Joseph Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, Vol. 53, No. 3, pp. 484 – 512, 1984.

[5] V. Rokhlin. Rapid Solution of Integral Equations of Scattering Theory in Two Dimensions. *J. Comput. Phys.*, Vol. 86, pp. 414–439, February 1990.

[6] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, Vol. 7, pp. 501–538, October 1985.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, Vol. 30, No. 8, pp. 207–216, 1995.

[8] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating Futures into Calling Standards. *SIGPLAN Not.*, Vol. 34, No. 8, pp. 60–71, 1999.

[9] Doug Lea. A Java Fork/Join Framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pp. 36–43, New York, NY, USA, 2000. ACM.

[10] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A Proposal for Task Parallelism in OpenMP. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pp. 1–12, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Chuck Pheatt. Intel®Threading Building Blocks. *J. Comput. Small Coll.*, Vol. 23, No. 4, pp. 298–298, 2008.

[12] Prabhanjan Kambadur, Anshul Gupta, Amol Ghoting, Haim Avron, and Andrew Lumsdaine. PFunc: Modern Task Parallelism for Modern High Performance Computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 43:1–43:11, New York, NY, USA, 2009. ACM.

[13] Rob von Behren, Jeremy Condit, and Eric Brewer. Why Events are a Bad Idea (for high-concurrency servers). In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pp. 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[14] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 268–281, New York, NY, USA, 2003. ACM.

[15] Gene Shekhtman. State Threads for Internet Applications. http://state-threads.sourceforge.net/docs/st.html.

[16] R. S. Engelschall. GNU Pth - the GNU Portable Threads. http://www.gnu.org/software/pth/.

[17] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[18] UPC Consortium. UPC language specifications, v1.2, 2005.

[19] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

[20] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, Vol. 17, pp. 1–31, August 1998.

[21] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 519–538, New York, NY, USA, 2005. ACM.

[22] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pp. 52–60, 2004.

[23] Alejandro Duran, Julita Corbalan, and Eduard Ayguade. Evaluation of OpenMP Task Scheduling Strategies. In *OpenMP in a New Era of Parallelism*, Vol. 5004 of *Lecture Notes in Computer Science*, pp. 100–110. Springer Berlin / Heidelberg, 2008.

[24] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 2, No. 3, pp. 264–280, 1991.

[25] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, Vol. 46, No. 5, pp. 720–748, 1999.

[26] Francois Broquedis, Francois Diakhate, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-Andre Wacrenier. Scheduling Dynamic OpenMP Applications over Multicore Architectures. In *OpenMP in a New Era of Parallelism*, Vol. 5004 of *Lecture Notes in Computer Science*, pp. 170–180. Springer Berlin / Heidelberg, 2008.

[27] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, and Chau wen Tseng. UTS: An Unbalanced Tree Search Benchmark.

[28] Steven T Zalesak. Fully Multidimensional Flux-Corrected Transport Algorithms for Fluids. *Journal of Computational Physics*, Vol. 31, No. 3, pp. 335 – 362, 1979.

[29] R. Löhner. An Adaptive Finite Element Scheme for Transient Problems in CFD. *Comput. Methods Appl. Mech. Eng.*, Vol. 61, pp. 323–338, April 1987.

発表文献

主著論文

1. 中島潤，田浦健次朗．適合格子細分化法による MassiveThreads の評価．先進的計算基盤システムシンポジウム (SACSIS2011)，東京，May 2011(投稿中)．

2. 中島潤，田浦健次朗．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．情報処理学会論文誌 プログラミング (PRO50)(採録予定)，March 2011．

3. 中島潤，田浦健次朗．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．並列/分散/協調処理に関するサマーワークショップ (SWoPP2010)，金沢，August 2010．

共著論文[1]

1. 塩谷 亮太，倉田 成己，中島 潤，五島 正裕，坂井 修一．Switch-On-Future-Event マルチスレッディング．先進的計算基盤システムシンポジウム (SACSIS2010)，東京，May 2010

2. 倉田 成己，塩谷 亮太，中島 潤，五島 正裕，坂井 修一．Switch-on-Future-Event マルチスレッディングの改良．並列/分散/協調処理に関するサマーワークショップ (SWoPP2010)，金沢，August 2010．

3. 原健太朗，中島潤，田浦健次朗．アドレス空間の大きさに制限されないスレッド移動を実現する PGAS 処理系．並列/分散/協調処理に関するサマーワークショップ (SWoPP2010)，金沢，August 2010．

4. 原健太朗，中島潤，田浦健次朗．アドレス空間の大きさに制限されないスレッド移動を実現する PGAS 処理系．情報処理学会論文誌 プログラミング (PRO50)(採録予定)，2011．

[1] これらの論文は本論文の研究とは関連していない．

謝辞

本研究を進めるにあたっては，多くの方々にお世話になりました．

近山隆教授にはご多忙な中何度も発表練習に参加していただき，数えきれないほどの誤りを正していただくのと同時に，特に研究成果を発表するということについて，多くの有益なご意見をいただきました．最初の発表練習でプレゼンテーションの原稿が真っ赤になって帰ってきたとき，自分の至らなさを恥ずかしく思うのと同時に感謝の念がこみ上げてきたのを鮮明に覚えています．田浦健次朗准教授は，右も左もわからなかった私に対して，研究テーマの決定から論文の執筆に至るまで研究のあらゆる段階において，懇切丁寧な指導をしてくださいました．また，私が研究の方向性を見失って行き詰まるたびに親切に相談に乗ってくださり，ともすれば消極的になりがちな私を粘り強く激励してくださいました．

横山大作さん，鴨志田良和さん， 柴田剛志さん，吉本晴洋さん，原健太朗君をはじめとするミーティングに参加してくださった皆さんは，私の研究についての議論に積極的に参加し，ある時は見落としている点を鋭く指摘し，またある時は的確なアドバイスをくださって，私が研究を進めるのを強力に後押ししてくださいました．浦晃君や矢野友貴君には気さくに話しかけられる同室の同期として，いつも私が他愛もない話を持ちかけるのにもかかわらず飽きずに話し相手になっていただき，また時には研究室の先輩として，わからないことだらけな私を導いていただきました．

石井礼子さん，荒井美佐緒さんには研究室の事務において，大いにお世話になりました．また，ここに名前を挙げていない研究室の皆様にも，計算機の管理や，研究室の行事など，様々なところで大いに助けていただきました．

皆様のご助力あってこそ，ここまで研究を進めることができたのだと強く感じております．この場を借りて心よりお礼申し上げます．

平成 23 年 2 月 9 日