

プリフェッチ機構を持つマルチコア向けソフトウェア分散共有メモリシステム

李 珍 泌[†] 佐藤 三 久^{†,††} 朴 泰 祐^{†,††}

プロセッサのマルチコア化に従い、マルチコアクラスタは大規模並列計算機のスタンダードな形になりつつある。本研究は複数の計算スレッドによるノード内並列化やページプリフェッチによる通信遅延の削減を行い、マルチコアクラスタが持つ計算資源を有効に使うソフトウェア分散共有メモリの実現を目指す。プリフェッチ機構の試作と性能評価の結果、計算とページプリフェッチのオーバーラップによりページ取得待ち時間の削減を削減することが確認できた。計算スレッドによるノード内並列化に対してはオペレーティングシステムのスレッドスケジューリングの機能に問題があり、満足いくパフォーマンスが得られず、今後の課題となるいくつかの問題点が明らかになった。

Software Distributed Shared Memory System with Page Prefetch Thread for Multi-core Processors

JINPIL LEE,[†] MITSUHISA SATO^{†,††} and TAISUKE BOKU^{†,††}

The multi-core cluster is becoming a standard platform of a large-scale, parallel computer system. In this research, we aimed to develop an effective software distributed shared memory system for multi-core clusters. For the effective use of multi-core processors in the cluster, we examined multi-threaded calculation and page prefetching thread on the software distributed shared memory. By using the page prefetching thread, we can reduce the waiting time for the remote page access. For multi-thread calculation on each node, we found that lack of efficient thread scheduling of the operating system restricts the performance of the software distributed shared memory system.

1. はじめに

PC クラスタはプロセッサとローカルメモリを持つノードをネットワークで接続することで構成される。そのため、共有メモリシステムと比べ拡張性が高く、高性能な並列計算機を容易に構築することができる。しかし、PC クラスタを用いた並列化には通信ライブラリである MPI(Message Passing Interface) を用いてデータの送受信をユーザが明示的に指示しなければならないため、プログラミングが複雑であるという問題がある。更に今日では、プロセッサのマルチコア化が進みノード内のプロセッサが複数のコアを持つようになったため、スレッドライブラリや OpenMP などを用いたノード内の並列化も考慮しなければならない。

分散メモリに対するより簡単なプログラミングモデルを提供するため、ソフトウェア分散共有メモリ (Software Distributed Shared Memory, 以後ソフトウェア DSM) という手法が存在する。ソフトウェア DSM とは、物理的に独立したメモリを、ソフトウェ

ア技術とネットワークによって、ユーザーレベルから見ると共有された一つのメモリのように見せかけるものである。ノード間の通信を記述する必要がなく、全ての共有メモリ領域をローカルメモリのように参照することができる。そのため、分散メモリ上で OpenMP のような簡単なプログラミングモデルを使うことができ、プログラミングが従来より簡単になる。

ソフトウェア DSM は分散メモリの拡張性と共有メモリの簡単さを両立できるメリットがある。しかし、MPI によって作成されたアプリケーションと比べ性能が低いため、普及には至っていない。また、ノード内並列化を考慮した実装は殆どされておらず、マルチコアのプロセッサをノード内に持つ SMP(Symmetric Multiple Processor) クラスタの普及に従い、ノード内の計算資源を有効に活用できる実装が必要とされている。

本研究ではこのような問題点を解決するために、ソフトウェア DSM のマルチスレッド化やプリフェッチ機構の設計を行い、SMP クラスタに対する効率的なシステムの実現を目指す。第 2 章ではマルチコアを有効利用するためのコンセプトを述べる。第 3 章ではページプリフェッチ機構を設計し、その効果について評価を行う。第 4 章では複数の計算スレッドによるノード内並列化やプリフェッチスレッドによつ通信遅延の削

[†] 筑波大学 大学院 システム情報工学研究科
Graduate School of Systems and Information Engineering,
University of Tsukuba
^{††} 筑波大学 計算科学研究センター
Center for Computational Sciences, University of
Tsukuba

減、これらのスレッドのスケジューリングによるマルチコアの有効な利用方法について検討を行う。

2. ソフトウェア DSM におけるマルチコアの有効利用

デスクトップ向けの CPU でもマルチコア化が進んでいる中、PC クラスタにおいても SMP ノードがスタンダードになることは明らかである。そのような環境では分散メモリ上の並列化に必要なノード間通信やデータの分散だけでなく、ノード内並列化によって全てのコアを有効に使う工夫をしなければならない。

2.1 複数の計算スレッドによるノード内並列化

SMP ノードにおいて各コアを有効に使うために複数の計算スレッドを用いてノード内並列化を行うのがもっとも一般的な方法である。しかし、従来の MPI と OpenMP によるハイブリッドなプログラミングモデルはユーザが分散メモリと共有メモリの違いを認識し、両方のプログラミングモデルが効率的かつ正しく動作するように注意しなければならない。その結果、細かい性能チューニングによる効率的な並列化が可能代わりにプログラミングが複雑になる。

それに対し、ソフトウェア DSM はユーザに対し仮想的な共有メモリを提供するため、ノード内並列化に対してもシームレスな形で対応することができる。ノード内では複数の計算スレッドを用いた並列化を行うことでハードウェアによる共有メモリを使い、ノード間ではソフトウェア DSM を用いることで、システムの全てのコアがメモリを共有する環境を実現する。ユーザはクラスタのアーキテクチャを意識せず、全てのコアがメモリを共有すると考えてプログラミングを行うことができるため、プログラミングは従来のハイブリッドなモデルより簡単になる。

2.2 機能特化した専用スレッドによるコアの有効利用

複数の計算スレッドを持つソフトウェア DSM では、ノード内の計算資源を余りなく活用することができる。しかし、現実ではメモリバンド幅とプロセッサ性能の乖離は年々増加していく傾向にあり、プロセッサに対するデータ供給が追いつかない状態になっている。従って、プロセッサ内の全てのコアを計算に割り当てるとしても、メモリバンド幅がボトルネックとなり台数分の性能向上を得られない可能性が高い。そういったことを考えると、計算性能のアップに貢献しない余ったコアをソフトウェア分散共有メモリの性能改善に使った方が効率的である。

例えば、ソフトウェア分散共有メモリではページ単位のリモートメモリアクセスが必要であり、ハードウェアによるサポートがない環境ではソフトウェアによる通信機構を実装している。こういった実装では自分が発行する通信の処理だけでなく、他のノードからの通信要求にも応えないといけないため、ハードウェアサポートがある場合と比べて大きな性能低下の原因となる。他のノードからのリモートメモリアクセス要求は自分のノードで行われる計算とは独立して行うことが

できるため、リモートメモリアクセスのための専用スレッドを生成し、ノード内の一つのコアで実行させることで計算が停止することなくリモートメモリアクセスの要求に対応することができる。

更に、専用スレッドによるページプリフェッチを行うことで見かけ上の通信遅延の削減を図ることも考えられる。ソフトウェア DSM におけるページプリフェッチとはデータが実際に必要とされる前にページの転送を行い、通信による遅延を削減する機能である。専用のスレッドによるページプリフェッチ機構は従来のプリフェッチ機構と比べて計算と通信のオーバーラップを実現できるというメリットがある。プリフェッチが開始してから対象ページの参照が行われる間に、ページの転送と計算を同時に行うことができ、見かけ上の通信の遅延を軽減することができる。

他には書き込みを行ったページの差分を計算し、メモリ同期ポイントの前に転送することで、差分の転送時間を削減するなど考えられる。

第3章では計算スレッドによるノード内並列化と専用スレッドによるプリフェッチに注目し、マルチコアクラスタを有効に利用できるソフトウェア分散共有メモリの実現を目指す。

2.3 マルチコアプロセッサにおけるスレッドスケジューリングの問題点

前節に述べたように、計算スレッド以外の専用スレッドによる機構を計算スレッドとともに動かそうとすると、スレッドスケジューリングが問題になる。専用スレッド機構が計算スレッドの実行の邪魔にならない範囲で実行されるためにはスレッドスケジューリングの優先度をスレッド毎に指定する必要がある。現在のデスクトップ向けオペレーティングシステムではリアルタイムスケジューリングを提供するものが殆どなく、スケジューリングの優先度をユーザが指定することができない。第4章では現在の OS による問題を明らかにすると共にその解決策を考える。

3. 専用スレッドによるページプリフェッチ機構

3.1 ページプリフェッチ機構の概要

ここではソフトウェア DSM に専用スレッドによるプリフェッチ機構を導入し、ページ転送に伴う遅延をどれだけ削減できるかを検討する。ソフトウェア DSM 実装の一つである SCASH-MPI¹⁾ (以後、単に SCASH と呼ぶ) をベースに専用スレッドによるプリフェッチ機構を実装し、その効果について評価を行う。同じパターンのメモリアクセスが反復して行われるアプリケーションを対象とし、アクセス履歴を用いたプリフェッチを行う。

本プリフェッチ機構は計算とのオーバーラップを図るため、計算スレッドとは独立した専用のスレッドによって実装される。計算スレッドとメッセージキューを共有し、キューにプリフェッチの要求があった場合にリモートメモリアクセスを行う。プリフェッチに必要なメッセージの送受信やデータの転送は全てプリフェッチスレッドが行うため、SCASH の通信レイヤ

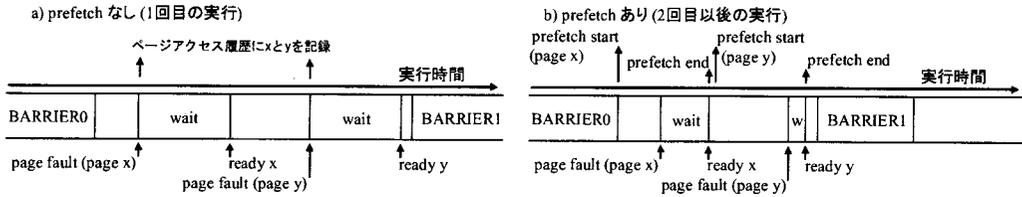


図1 SCASH のページプリフェッチ

とは独立して動作する。

SCASH ではあるメモリ同期ポイントとその次の同期ポイントまでを計算フェーズと呼び、プリフェッチを行う単位とする。プリフェッチを行う区間はユーザが指定する。反復を行うループを囲むようにして、プリフェッチ開始関数と終了関数を挿入する。プリフェッチが始まるまでの手順を以下に示す。

- (1) 1 回目の反復が実効される。プリフェッチはまだ開始されない。
- (2) 他のノードに対するアクセスが発生すると、そのページ番号を現在の計算フェーズにおけるアクセス履歴として記録しておく。
- (3) 1 回目の反復が終了。2 回目の実行が開始され、プリフェッチが開始される。
- (4) 計算フェーズが始まる直前のメモリ同期バリアでその計算フェーズでアクセスされたページの転送を開始する（専用スレッドによるプリフェッチ）。

図1でSCASHによるページプリフェッチの動作を示す。1回目の実行で、ページxとyに対するリモートメモリアccessが行われたため、この計算フェーズの参照履歴としてxとyが記録される。2回目実行では計算フェーズの直前のメモリ同期ポイントでページxとyに対するプリフェッチが行われる。計算はプリフェッチと並列に実行されるが、ある時点でページxとyに対するメモリアccessが発生する。ページプリフェッチによりメモリアccessより前にページの転送が行われていたため、ページ転送による計算の遅延が削減される。

シンプルな実装としてプリフェッチ開始時に以後参照される全てのページのプリフェッチを行ってしまう方法が考えられる。しかし、ページが必要とされる計算フェーズより前の計算フェーズでページの内容が変更される場合、変更が行われた直後のメモリ同期ポイントでページの無効化が行われてしまうため、プリフェッチが無駄になってしまう（同期にinvalidateプロトコルを採用している場合）。そのため、プリフェッチを

行うのは履歴に存在する全てのページをプリフェッチするのではなく、次のメモリ同期ポイントまでに参照されたページのみをプリフェッチの対象とする。

通信と計算が上手くオーバーラップし、ページフォルトが発生する前に転送が終了すると、メモリアccessによるページフォルトが起きても、ページ保護属性を変えるだけで計算を再開することができる（プリフェッチがページ参照の前に終わっていない場合でも通信の遅延を軽減させる効果がある）。

3.2 プリフェッチ機構の性能評価

ここではプリフェッチ機構に対する性能評価を行う。表1に本研究で用いた評価環境を示す。プリフェッチ機構の性能評価では計算スレッドとプリフェッチスレッドが1個ずつ生成される。

3.2.1 プリフェッチと計算のオーバーラップによるページ取得待ち時間の削減

まず、専用スレッドによるプリフェッチと計算のオーバーラップによりページ取得に伴う通信遅延をどのくらい削減できるか調べる。評価のため、以下のようなコードを用いる。

```
begin_prefetch_page(n); // step 1
for(i = 0; i < COUNT; i++)
    temp[i] = i; // step 2
x = array[N]; // step 3
```

step 1でページnのプリフェッチを開始する。ページプリフェッチはstep 2で行われる計算とオーバーラップして実行される。その計算量はCOUNTを変えて調整することができる。step 3で参照されるarray[N]はstep1でプリフェッチ対象としたpage(n)にマッピングされている変数である。よって、step 3が実行される前にプリフェッチが完了していれば通信時間が完全に隠蔽されたことになる。そうでなくても、ある程度の通信時間が隠蔽されれば、全体の実行時間は短縮されるはずである。

表1 評価環境

項目	名称
CPU	Dual-Core Opteron 2218 (2.6GHz) x2
Memory	4GB
Network	1000BASE-T ethernet
OS	Linux kernel 2.6.18.53
MPI	OpenMPI 1.2.6 (disable progress thread)

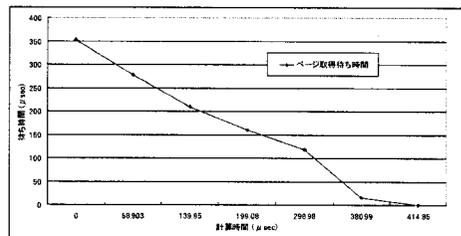


図2 オーバーラップする計算時間によるページ取得待ち時間の変化

□ 計算式

```
for(i = i_start; i < i_end; i++)
  for(j = 0; j < JMAX; j++)
    uu[i][j] = (u[i][j+1] + u[i][j-1] + u[i+1][j] + u[i-1][j]) / 4;
```

□ 反復毎に参照されるリモートページ

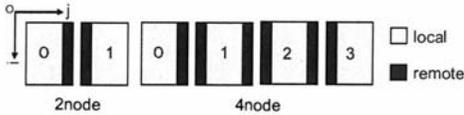


図3 laplace のメモリアクセスパターン

図2に評価結果を示す。プリフェッチと参照の間に計算が存在しない場合、プリフェッチを行わない場合のページ転送と同じ程度の待ち時間が発生した。間の計算量を増やしていくと待ち時間が減少し、ページ転送に掛かる時間と同程度以上の計算が存在する場合は待ち時間が殆ど発生しない。この結果はスレッドによるプリフェッチ機構が計算と通信をオーバーラップすることで見かけ上の通信遅延を隠蔽していることを示す。通信時間と同程度の計算時間がプリフェッチと参照の間に存在する場合、通信は殆ど隠蔽される。

3.2.2 実アプリケーションにおける性能評価

もう一つの評価として、2次元ラプラス方程式をSCASH上で計算するアプリケーション(以後laplaceと呼ぶ)を作成し、ページ取得の待ち時間の変化を見る。laplaceは2次元ラプラス方程式を差分法によって計算するもので、反復により同じ計算式が繰り返し実行されるため、ページプリフェッチによる効果が期待される。図3にlaplaceのメモリアクセスパターンを示す。各ノードでは計算の序盤や終盤、または両方で境界領域に対するリモートメモリアクセスが発生する。データ配列はdouble型変数の1024x1024個の要素を持ち、境界領域の大きさは片方でページ2個分になる。図4にプリフェッチ機構の導入によるページ取得の待ち時間の変化を示す。プリフェッチ区間で行われたページ転送による遅延時間の累積をノード毎に表したものである。

2ノードの場合、両ノードの遅延の変化が著しく異なる。ノード0の場合、プリフェッチによりページの

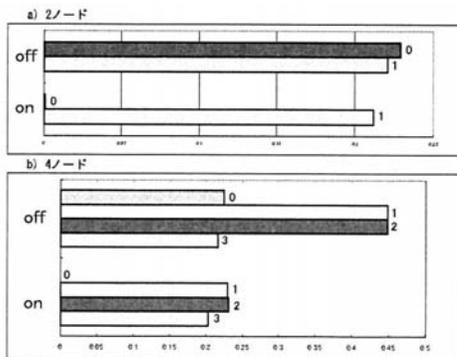


図4 ページプリフェッチによるページ取得待ち時間の変化

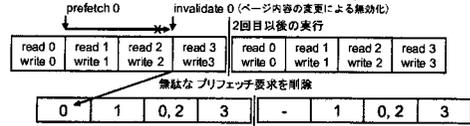


図5 ページプリフェッチ機構の改良案

待ち時間が殆ど削減できたのに対し、ノード1ではプリフェッチなしの場合と比べ、僅かな変化しか得られなかった。これはリモートメモリアクセスのタイミングに原因がある。ノード0の場合、計算フェーズの終盤で通信が発生するため、プリフェッチされるページが実際に必要になるまで十分な余裕がある。転送するページの数も2個で少ないという要因もあり、計算と通信が上手くオーバーラップしてページ取得に必要な見かけ上の遅延がほぼ削減できた。ノード1の場合は計算フェーズの最初に通信が発生するため、直前のメモリ同期ポイントで発行されたプリフェッチが間に合わない。ページプリフェッチはページ取得の見かけ上の時間を削減するもので、通信の遅延を削減するものではない。ノード1ではプリフェッチとオーバーラップできる計算が殆ど存在しないため、通信の遅延による影響を隠蔽できず、プリフェッチを導入しなかった場合と比べてほぼ同じ結果となった。4ノードの場合も同様の理由で計算フェーズの先頭で発生する通信を上手く隠蔽できず、ページ取得遅延はノード1とノード2で半分程度になる。

3.2.3 性能評価の考察

性能評価の結果、ページプリフェッチによりページ取得に伴う遅延を削減することができた。しかし、その効果は通信の発生するタイミングにより大きな差があり、更なる改良が必要となる。今後は様々なアプリケーションに対して調査を行い、最適なプリフェッチの開始タイミングを、なるべく小さいプログラミングコスト(ユーザからの情報)で見えるように検討する。その案の一つを図5に示す。前節までの実装では履歴の記録が終わってから各計算フェーズに対するプリフェッチを行ったが、図5の案では計算フェーズで参照されたページを直後のメモリ同期バリアでプリフェッチするようにする。そうすることでなるべく多くの計算とプリフェッチをオーバーラップさせることができる。途中で無効化されるページの場合、ページ履歴を変更し、前の計算フェーズの履歴からプリフェッチ要求を削除する。そうすることでページを最終に更新したメモリ同期ポイントを探すことができ、無駄なプリフェッチを防ぐことができる。

4. ノード内マルチコアの有効利用

従来のソフトウェアDSMは一つのノードに対し一つのコアを想定して実装されているものが多く、マルチコアに対応しているものは数少ない。マルチコアに対応しているものも多く各コアに対してプロセスを生成することで実装を行っている²⁾。分散メモリの各ノードではアプリケーションがプロセスとして実行されており、ノード内での並列化をプロセスによって行

うことは自然な拡張である。旧 Linux カーネル (2.4 以前) がスレッド単位のシグナル発行をサポートしていないなどの問題も背景にあり、従来のソフトウェア DSM はこのような形で SMP クラスタに対応している。しかし、プロセスによる実装はノード内のページに対する保護属性を共有できないため、ノード内でもプロセス間の通信による情報交換が必要になる。スレッドによる実装ではノード内のページデータはもちろん、その保護属性を共有することができるため、性能低下の原因となるノード内通信が不要となる。

4.1 SCASH のマルチスレッド化

ノード内のコアを有効に使うため、実行開始時に複数の計算スレッドを生成する実装を行う。計算スレッドはメモリ領域を共有し、各自並列に計算を行う。ノード毎にプリフェッチを担当する専用スレッドが生成され、計算スレッドからの要求を処理する。

SCASH は通信のための専用スレッドを持ち、プリフェッチスレッドと同様、通信の要求をメッセージキューを用いて処理していた。このような実装はスレッドセーフでない MPI 実装でも安全にリモートメモリアクセスを実装できるというメリットがある。また、マルチコアプロセッサの余ったコアを用いることでハードウェアサポートなしで効率的な通信を行うことができる。しかし、通信スレッドがポーリングしながら CPU 時間を消費するため、パフォーマンス面で問題がある (これはプリフェッチスレッドにも当てはまる)。また、コア数が増加し、計算スレッド数が増加するにつれてメッセージキューの排他処理によるオーバーヘッドも性能低下の原因となる。スレッドセーフな MPI 実装を用いる場合は送信は計算スレッドが行い、受信を処理するスレッドを生成することで外部から通信があった場合のみ動作させる実装ができる。研究の背景もあるように今後クラスタの CPU がマルチコア化を続けていくにつれ、スレッドセーフな MPI 実装は必須要素となっていくと考えられるため、本研究ではポーリングする通信スレッドの実装を変更し、メッセージの送信を各計算スレッドによって行うようにした。外部からメッセージが届いた場合は受信専用のスレッドが対応し、ページデータのやり取りや受信されたメッセージの処理を行う。

ノード内では計算だけでなく、シグナルハンドラの実行を含めたソフトウェア DSM の処理も並列化される。その一例としてメモリ同期が挙げられる。メモリ同期はページ単位で行われるため、各計算スレッドが異なるページの処理を並列に行うことができる。並列化の結果、メモリ同期に掛かるオーバーヘッドを軽減させることができ、ソフトウェア DSM の性能を上げることができる。

スレッド毎のページフォルトの処理やスレッドセーフな MPI 実装など、背景技術の充実により SCASH のマルチスレッド化は容易に実現できるものと考えていたが、幾つかの問題点が明らかになった。次の節でマルチスレッド化に伴い明らかになった大きな問題点

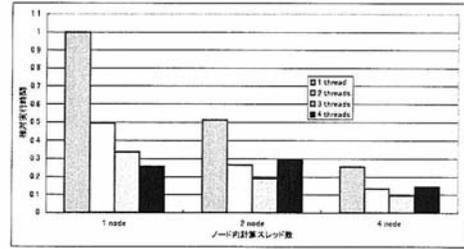


図 6 予備評価の結果

を二つ挙げる。

4.2 マルチコアの効率的な利用のための課題

4.2.1 プリフェッチスレッドのスケジューリング

まず問題となるのはプリフェッチスレッドによるオーバーヘッドである。プリフェッチスレッドはポーリングしながらメッセージキューを処理するため、常時 CPU 時間を消費する。コア数分の計算スレッドが生成された場合、プリフェッチスレッドによる CPU 時間消費はシステム全体の性能を低下させる要因となる。そのため、プリフェッチスレッドのスケジューリングが本研究の重要な要素となる。予備評価として通信・メモリ同期などが一切発生しないアプリケーションを作成し、プリフェッチスレッドの CPU 時間消費量の最小値を調べる。一次元配列を並列にアクセスするアプリケーションを作成し、計算時間を計ることで評価を行った。評価には前章で用いたものと同じ環境を利用している。

図 6 に評価結果を示す。ノード内の計算スレッド数を調整しながら評価を行った結果、コア数と同じ 4 スレッドにした場合、性能の低下が見られた (1 ノードの場合は通信スレッドが生成されないため、計算スレッド数をコア数まで増やしても性能低下が発生しない)。これはプリフェッチスレッドが計算スレッドと CPU コアを巡って競合するため起きる性能低下で、通信やプリフェッチがまったく発生しないアプリケーションの場合でもプリフェッチスレッドの実行が頻繁に行われていることを示す。このような性能低下を防ぐためにはプリフェッチの要求がない場合、スレッドの実行を停止させることが望ましい。しかし、計算とプリフェッチのオーバーラップを行うためにプリフェッチスレッドを計算スレッドと同時に実行する必要がある。従って、プリフェッチスレッドのスケジュール優先度を抑える必要がある。

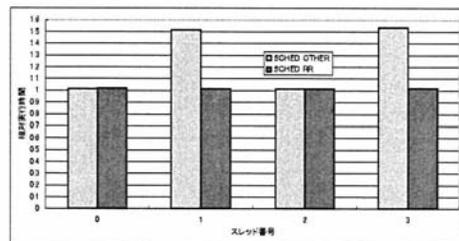


図 7 リアルタイムスケジューリングによる性能変化

スレッド優先度を変えた場合の性能変化を調べるため、別の評価を行った。その結果を図7に示す。これは SCASH を使わず、ノード内のコア数(四つ)分の計算スレッドと余分のポーリングする一つのスレッド(終了フラグを検査するだけで何もしない)を生成して評価した結果である(1ノードのみを利用した実行結果)。Linux のデフォルトのスケジューリングポリシーである SCHED_OTHER(時分割スケジューリング)と Round Robin スケジューリングポリシー(計算スレッドの優先度を最大値に、通信スレッドは最小値に調整)を比較した。評価の結果、デフォルトでは頻りにスケジューリングされていた余分なスレッドの実行が抑えられ、性能の向上が見られた。この結果から SCASH のマルチスレッド実装でもスレッド優先度を変えてプリフェッチスレッドの実行頻度を変えられることが必要であることが分かる。しかし、現在の Linux ではスーパーユーザでしかスレッドの優先度を変えることができない。スーパーユーザとして PC クラスタで並列プログラムを実行するのはセキュリティ上危険であるため、ユーザプロセスによるスレッドスケジューリング機構が必要である。もっとも理想的なモデルは計算スレッドがアイドルな時にプリフェッチスレッドを実行することである。プリフェッチスレッドが計算スレッドより優先して実行されて計算の邪魔になってしまえば本末転倒なので、アイドル時間のような無駄な時間を活用することが有効と考えられる。しかし、CPU のアイドル時間を正確に捉えてスレッドを動作させることは極めて難しい。そのため、プリフェッチスレッドのスケジューリング優先度を動的に変更し、なるべく理想的なモデルに近づけるような機構を目指す。

アイドル時間を狙った無駄な時間の活用は計算スレッドの数がコア数と同じである場合は合理的な選択である。しかし、実効時間が限定されるため、プリフェッチによる効果は低下する。本稿の最初に述べたように、全てのコアを計算に使うよりはプリフェッチスレッドのような通信機構にコアを使った方が性能向上につながる可能性がある。それを実証するため、ユーザレベルのスケジューリング機構を実現すると共に様々なアプリケーションを用いて実行スレッド数とパフォーマンスの変化についてより深い分析を行う必要がある。

4.2.2 ページ操作の並列実行

もう一つの問題は mprotect のような仮想ページに対する操作が複数の計算スレッドで並列に実行できないことである。Linux では仮想メモリ領域をページより大きい単位で管理しており、ページ毎に独立して属性の変更を行うことができない(仮想メモリ領域全体に対する排他制御が行われる)。従って、計算スレッドを多くして並列に動作させてもコア数分の性能向上を得られないのが現状である。仮想ページに対する保護属性の変更などはソフトウェア DSM システムで頻りに実行されるため、計算スレッドが増加するにつれて性能アップを防ぐ障害となる。実際、laplace による性能評価の結果、前に述べたプリフェッチスレッドの

CPU 消費の問題も重なり、マルチスレッド化による性能向上は殆ど見られない結果となった(もっとも大きな原因は計算中・メモリバリア同期で発生するページ操作の逐次実行であった)。そのため、分散共有メモリの専用メモリ機構を用意するなど、OS の基本的な機能に関わる根本的な対策が必要となる。

5. まとめと今後の課題

本研究ではクラスタ内のマルチコアを効率的に利用するソフトウェア DSM の実現を目指した。スピードアップが見込めるまでは計算スレッドを複数生成してノード内並列化を行う方法を用いるが、計算の性能アップが限界に達した場合は余ったコアを用いてページプリフェッチを行うことで通信の遅延を削減する手法を検討した。

性能評価の結果、プリフェッチと計算のオーバーラップによりページ取得待ち時間を削減することができた。しかし、複数の計算スレッドによるノード内並列化においては Linux のメモリ機構の実装上の問題により、満足のいく性能向上が得られなかった。また、ポーリングするプリフェッチの CPU 消費がパフォーマンス上問題になり、動的に優先度を変更できるスケジューリング機構が必要であることが分かった。これらの問題点は OS の根本的な部分に関わるため、分散共有メモリの管理、動的スレッドスケジューリングを行う機構をカーネルモジュールもしくは OS の機能を変更するなどして解決していくことにする。

謝辞 本研究の一部は科学研究費補助金基盤研究(B) 課題番号 18300006 「次世代 PC クラスタを活用する超大規模仮想メモリ空間支援システムの研究」によるものである。

参考文献

- 1) 小島 好紀ほか, "MPI 上のソフトウェア分散共有メモリシステム", 情報処理学会研究報告(第98回 HPC 研究会), pp. 43-48, 2004
- 2) 菊池 康祐ほか, "SMP 向け分散共有メモリシステム SMS2", 情報処理学会第65回全国大会論文集 vol.1, pp.73-74, 2003
- 3) Sumit Roy, et al., "Strings: A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters", 1998
- 4) Daniel J. Scales, et al. "Fine-Grain Software Distributed Shared Memory on SMP Clusters", Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture, 1997
- 5) 丹羽 純平ほか, "コンパイラが支援するソフトウェア DSM におけるプリフェッチ機構", 情報処理学会研究報告(2004-ARC-156), pp.7-12, 2004
- 6) 松葉 浩也ほか, "動的アクセスパターン解析によるソフトウェア分散共有メモリ", Symposium on Advanced Computing Systems and Infrastructures (SACIS) 2004, pp.355-364, 2004
- 7) E. Bianchi, et al., "Some Experiences in fast hard realtime control in user space with RTAI-LXRT", Realtime Linux Workshop 2000, 2000