Vol.2018-HPC-165 No.23
2018/7/31

論理タイムスタンプに基づく分散共有メモリライブラリの実装

遠藤 亘[1,a)]　田浦 健次朗[1,b)]

概要：共有メモリ処理系で一般的に用いられるディレクトリベースコヒーレンスは，データ書き込み時のキャッシュ無効化メッセージの送信数増大が問題とされてきた．近年では，ディレクトリを用いないキャッシュ無効化手法として，論理タイムスタンプに基づく手法が提案されているが，不要なキャッシュミス回数が増大する欠点がある．そこで本稿では，論理タイムスタンプに基づく手法と，分散共有メモリ (DSM) で一般的な Write notice に基づく手法を併用した新たなキャッシュ無効化手法を提案し，DSM ライブラリとして分散メモリ型計算機用に実装して，その有用性を検証する．提案するライブラリは，データ書き込み時のノード間通信削減を目的として，Probable owner と呼ばれる手法でホームノードの自動的な再配置を行う特徴も有する．API としては OpenMP を用いており，通常の OpenMP プログラムを原則的に無変更で動作させるため，コールスタックを DSM で管理する手法も導入している．評価では共有メモリ用ベンチマークプログラムを用いて性能を比較し，現状の課題を整理する．

Wataru Endo[1,a)]　Kenjiro Taura[1,b)]

1. 序論

分散共有メモリ (Distributed Shared Memory, DSM) [1] [2] とは，分散メモリ型計算機上で動作する共有メモリ処理系のことである．MPI や PGAS のような分散メモリ用の一般的なメモリモデルでは，プログラマが計算ノード間の通信を明示的に記述し，逐次プログラムからの大幅な改変が必要である一方で，DSM は透過的な共有メモリ処理系であるため既存の共有メモリプログラムを動作させることができる．

DSM の研究は 1990 年代を中心に活発に行われたが，コヒーレントキャッシュのスケーラビリティ向上は困難な課題であり，当時の処理系は数十コア程度までしかスケールさせることができなかったため，現在だと同様の研究は PGAS が主流になった．一方で，分散メモリ型計算機のハードウェア事情は当時から変化しており [3] [4] [5]，ノード間インターコネクトの相対的な性能向上，RDMA と共有メモリの概念的な類似性，マルチコアやメニーコアの一般化といった複数の観点から，DSM について再考する余地は十分残っている．

DSM の実装にはディレクトリベースコヒーレンスと呼ばれるキャッシュ無効化手法が一般に使用されるが，書き込み発生毎に無効化メッセージを全てのデータ共有中のノードに送信する必要があるため，スケーラビリティ向上に限界がある．そこで，本研究では，近年提案された論理タイムスタンプを用いてキャッシュ無効化を行う手法 [6] [7] [8] に着目し，これをソフトウェア DSM で一般的な Write notice [9] [10] という手法と組み合わせることで，ディレクトリが不要な新しいコヒーレンスプロトコルを提案する．

また，false sharing に対処するための手法としてソフトウェア DSM ではホームベース DSM [11] が知られているが，静的に指定されるホームノードと書き込むノードが一致しないと書き込みレイテンシが増大する問題が知られている．そこで，本研究ではオーナー移動のための Probable owner [12] という手法を前提として，書き込み時にホームを常に移動する手法を採用することで，RDMA 化が容易な手法を提案する．

提案手法を実際に DSM ライブラリとして実装し，評価を行った．ノード間通信は MPI を用いており，RDMA 化を前提とした通信処理も現在は MPI-3 RMA を利用して

[1] 東京大学 大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
[a)] wendo@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

いる．DSM の API としては OpenMP を採用し，現在は static scheduling での parallel for 文のような基本的な機能のみをサポートしている．ライブラリのみの手法で共有メモリプログラムを透過的に動作させるため，以前提案したコールスタックを DSM 上で管理する手法 [13] も導入しており，将来的なノード間ワークスティーリングを含めた各種拡張を前提として設計を行っている．今回の評価には NAS Parallel Benchmark [14] の非公式版 [15] を使用し，ハードウェア共有メモリでの実行結果と比較した．

2. 研究背景

2.1 メモリコンシステンシモデル

メモリコンシステンシモデルは，共有メモリシステムにおける各スレッドのメモリの読み書き結果を定義する規則である．最も厳しいコンシステンシモデルは Sequential Consistency (SC) [16] であり，メモリアクセスの実行結果が (1) 全てのメモリアクセスの（実行時に決まる）全順序と (2) プログラムで予め指定されたアクセス順序の 2 つに従うことを要求する．

DSM ではノード間レイテンシを隠蔽するためにメモリコンシステンシの緩和が一般的であり，代表的なものに Release Consistency (RC) [17] [9]，Entry Consistency [18]，Scope Consistency [19]，DAG Consistency [20] [21] などが知られている．

緩和型コンシステンシでは，各メモリアクセスを同期または非同期アクセスに分類し，スレッド間では同期アクセスのみで順序付けを行う．同期アクセスはミューテックスのロック・アンロックやスレッド生成・破棄などのスレッド間の同期を意味し，非同期アクセスはそれら以外の通常の読み書きを意味する．同期アクセスはさらに，アンロックなら release，ロックなら acquire といった形で分類され，例えば RC においては対応する release と acquire の間でのみ順序付けが行われる．この順序付けは happens-before 半順序と呼ばれ，SC における (1) の全順序の条件の代わりにメモリアクセスを順序付ける．

緩和型コンシステンシでは，次の 3 条件を満たす状況をデータレースと呼ぶ．

(1) 複数のメモリアクセスが同じアドレスを指す．
(2) それらのうちの一つ以上が書き込みである．
(3) 互いに happens-before によって順序付けられない．

プログラムにデータレースが存在しないという性質を data-race-free (DRF) [22] と呼び，DRF プログラムが SC と同様に決定的に動作することを保証するコンシステンシモデルは SC-for-DRF と呼ばれる．SC に基づいてプログラムの挙動を把握できる利点があることから，Java や C++11 以降 [23] のような近年のプログラミング言語のスレッディングモデルはいずれも SC-for-DRF の考え方に収束しており，本稿でも SC-for-DRF を前提として議論を進める．

2.2 透過的なキャッシュシステムの方式

ソフトウェアだけで透過的な DSM 処理系を実現する場合，ページベース [12] とコンパイラベース [24] の 2 つの手法が知られている．ページベース DSM では，プロセッサと OS が持つメモリ保護機構を用いて，キャッシュされていないデータ領域を `mprotect()` 関数によって意図的にアクセス不能にし，セグメンテーションフォールト (SIGSEGV) をシグナルハンドラで捉えてキャッシュ読み込みを実行する．これに対し，コンパイラベース DSM は，コンパイラによって全てのメモリアクセス命令を書き換えることでキャッシュミス判定を実現する．今回は実装の簡便さからページベースの手法を用いる．ページベース DSM の利点はキャッシュヒット時にオーバーヘッドが発生しない点が挙げられ，欠点としてシステムコールのオーバーヘッドが避けられない点が挙げられる．

一般にキャッシュシステムはメモリを一定サイズのブロックに分割して取り扱い，ハードウェアにおいては特にこの単位をキャッシュラインと呼ぶ．ページベース DSM において，ブロックサイズはページサイズ（例えば 4 KiB）の倍数である必要があるが，コンパイラベース DSM にはこうした制約がない．

2.3 DSM における書き込みの管理手法

ここでは，共有メモリシステムにおけるキャッシュブロックへの書き込みの管理手法について述べる．まず，共有メモリでは，複数のプロセッサが同一キャッシュブロック上の異なるワードに書き込みを行った場合，false sharing と呼ばれる状況が発生する可能性がある．一般には false sharing は性能低下の原因になるため，プログラマが意識して避けるべきとされるが，共有メモリシステムにとっては false sharing が発生したとしてもプログラムを正常動作させる必要がある．

共有メモリシステムにとって最も単純な false sharing の対処方法は Single-Writer 型プロトコルで，各時点において書き込み可能なプロセスを 1 つに限定する．通常のマルチコアプロセッサにおけるコヒーレントキャッシュは Single-Writer 型である．Single-Writer 型はシンプルであるという利点があるが．false sharing 時には性能が低下しやすい欠点がある．特にページベース DSM におけるブロックサイズはキャッシュラインよりも大きく，false sharing の頻度がマルチコアのコヒーレンスよりも高いことを考慮する必要がある．また，ページベース DSM を RDMA で実装することを想定した場合，Single-Writer を実現するにはリモートプロセスでの書き込みを `mprotect()` システムコールで禁止する必要があり，RDMA 化することは困難である．

以上のような問題の解決のため，ソフトウェア DSM においては Multiple-Writer 型と呼ばれる手法が知られてい

る．Multiple-Writer 型では複数プロセッサからの同一ブロックへの書き込みを許可し，後で同一ブロックへの書き込みを併合することでコヒーレンスを保つ．書き込みの併合に使われるのが “twinning” [25] と呼ばれる手法で，各プロセッサは書き込み開始前のブロックのコピー (twin) を保持しておき，併合時には差分 (“diff”) を適用することで並行な書き込みを管理できる．Multiple-Writer 型の利点は，通信なしで書き込みを開始できるため，書き込み開始時のレイテンシが小さい点である．一方で，欠点としては twin と diff を管理するための時間・空間コストが挙げられる．近年ではハードウェアにおける Multiple-Writer 型プロトコル [26] [27] も研究されているが，基本的な発想は diff と同様である．

Multiple-Writer 型はさらにホームレス型 [9] [10] とホームベース型 [11] の 2 つに分けられる．同一ブロックへの書き込みについて，ホームレス型ではデータを読むプロセスが diff の併合を行うのに対し，ホームベース型ではブロック毎にホームプロセスを割り当て，複数プロセスの diff をホームに集約して併合する．ホームレス型については以下の問題が知られている [11] [28] ため，ホームベース型が登場してからはホームベース型の採用が一般的である．

(1) ホームレス型において，書き込んだプロセスは読み出されうる全ての diff を保持しなければならず，そのままではメモリ容量を消費し続けてしまう．TreadMarks [9] [10] は diff を破棄するためにガーベッジコレクション機能を備えていたが，定期的なバリア同期が必要となってスケールしにくかった．ホームベースでの diff は適用後に破棄されるので，この問題は起きない．
(2) データを読み出す側が diff を適用するので，全ノードで考えると同じ diff が複数回適用されることになる．これに対して，ホームベースでは diff の適用は必ず一度しか発生しない．

ホームベース型の問題点は，基本的にホームプロセスが静的に割り振られるため，書き込むプロセスと一致しない場合は書き込み毎に通信が発生することである．そこで，書き込みレイテンシを低減するためにホームを動的に “移動” する手法 [29] [30] も研究されている．その場合，ホーム移動後も全てのプロセスから正しくホームを特定できる必要があり，「現在のホームプロセス ID」についてコヒーレンスの問題が生じる．この解決には 3 つの手法が存在し，後に述べるデータのコヒーレンス同様にブロードキャストとディレクトリに加えて，Probable owner [31] がある．Probable owner とは各ノードでホーム（あるいはオーナー）へのリンクを持ち合う手法で，仮に最新のホームを指していなくても連結リストを辿ることで最新のホームを取得できる．Probable owner は特定プロセスに固定されたデータ構造を要せず，動的にホームを決定できるため柔軟性が高い．一方で，最悪の場合はリンク長がプロセス数まで増大し，あるプロセスがホームにアクセスしようとしても他のプロセス間でホーム移動を続けることでライブロックが発生しうる問題 [32] もある．

diff 適用のタイミングについても，Eager 型と Lazy 型の 2 つに分けられる [9]．RC の場合において，Eager 型（あるいは Release 型）は release 時にすぐに diff を適用するが，Lazy 型（あるいは Acquire 型）はその後の acquire または load 時に diff の適用を指示する．Lazy 型は diff の適用を遅らせることが可能なため，書き込みレイテンシを短縮できる利点があるが，読み込み側が diff の適用を待つ必要があるため読み込みレイテンシは増大する．Single-Writer 型の議論とも関係して，Lazy 型の場合は読み込み側が diff の適用を書き込み側に依頼する必要があり，一般にこうした処理はメッセージングで実装されるため RDMA で実装することは比較的困難である．

2.4 ディレクトリを用いたキャッシュ無効化手法

共有メモリにおけるもう一つの課題として，コンシステンシモデルに従ったデータの読み込み結果を保証する必要がある．まず，データ書き込みが読み込み側にいつ実際に転送されるかによって，Invalidate 型と Update 型の 2 つの実装方式に分けられる．Update 型では書き込み発生時に読み込み側がデータを即時更新するが，Invalidate 型では書き込み発生時にデータを無効化するのみで次回の読み込み時までデータ本体の更新を遅延させる．Update 型は読み込みレイテンシが短縮できる可能性があるが，実際には使われないデータを転送する可能性も高いため，本稿では以降 Invalidate 型を前提に議論する．

Invalidate 型では，コンシステンシモデルに従って適切なブロックを選択し無効化する必要がある．最も単純なキャッシュ無効化手法はブロードキャストベース（あるいはスヌーピング）で，書き込み毎に全プロセスにキャッシュ無効化メッセージを送信するものであるが，小規模なシステムでのみ用いられる．これに対し，より大規模なシステムでの標準的な手法はディレクトリベースコヒーレンスであり，ディレクトリと呼ばれる集中型のデータ構造を用いてキャッシュ共有中のプロセスの ID を記録する．ディレクトリベースは無効化メッセージ数を削減できるため，ブロードキャストベースに比べてスケーラブルであるが，(1) ディレクトリに必要な容量が最大でプロセス数に比例する，(2) 無効化メッセージを送らないと書き込みが進行できない，(3) 状態遷移が複雑といった問題が知られている．

ディレクトリベースの改良に関しては，近年のハードウェア共有メモリにおいても多くの研究例 [33] [34] [35] があり，ディレクトリのデータ構造を工夫することでプロセス ID の集合の容量を削減する（そして回路面積を減らす）ことがこれらの手法の目的である．一方で，ディレクトリを用いるという前提に変わりはなく，無効化メッセージを

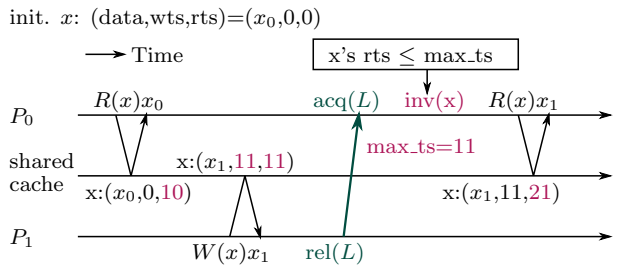


図 1: Tardis の動作原理 (lease=10)

送信しないと書き込みが進行できないという問題は解決できていない．

2.5 ディレクトリを用いないキャッシュ無効化手法

前述のディレクトリの問題点を解消するため，ディレクトリが不要なキャッシュ無効化手法も存在する．キャッシュ無効化メッセージを減らすため，読み込み側が自発的にキャッシュを無効化する考え方は “Self-invalidation” [36] と呼ばれる．Self-invalidation を利用したコヒーレンスの例として，ブロックサイズよりも粗い粒度（例えばページサイズ）で共有の有無を管理する P/S 分類 [27] [37] [38]（あるいは P/S3 分類 [5]）と呼ばれる手法や，コンパイル時解析を用いる手法 [39] などがある．

タイムスタンプベースコヒーレンス [40] [41] [42] [6] [7] [8] は近年登場したディレクトリ不要のキャッシュ無効化手法の一種である．タイムスタンプベースの基本的な発想として，読み込み側が「いつキャッシュが無効になるか」を示すタイムスタンプをブロックごとに記録しておき，書き込み側はそのタイムスタンプが過ぎた際に全ての複製が無効になったと判断する．ディレクトリベースとの最大の違いは，書き込み側から読み込み側に対して無効化メッセージを送信しないため，どのプロセスがキャッシュを保持しているか追跡する必要がない点である．

タイムスタンプベースの手法の多くは，一定時間毎に全プロセスで同期的にインクリメントされる “物理タイムスタンプ” に基づいている．物理タイムスタンプの実装には大域的なバリア同期が必要なため，スケーラビリティ向上に限界がある．より深刻な問題として，読み込み側がキャッシュ無効化を行うタイムスタンプに達するまで書き込み側は待つ必要があり，書き込みレイテンシが増大して性能が低下する [42] .

本研究では，タイムスタンプベースの手法の中でも，論理タイムスタンプを用いるという特色を持つ Tardis [6] [7] [8]（図 1）という手法に注目した．Tardis における論理タイムスタンプは Lamport clock [43] とほぼ同義で，単一の整数値で並行システムのイベント間を順序付ける．Tardis において，各キャッシュブロックは write timestamp と read timestamp を持ち，読み込み側がまず（ハードウェア共有メモリでは）共有キャッシュに write timestamp よりも大きい read timestamp を記録しておく．書き込み側はその値を読み取り，read timestamp よりも大きい write timestamp を保持する．そして，書き込み側と読み込み側で同期が発生した時，読み込み側は書き込み側の write timestamp を受け取り，以降はその値未満の read timestamp を持つ全てのブロックが無効になる．

物理タイムスタンプを用いる手法と比べて，Tardis が用いる論理タイムスタンプの生成には大域的なバリアが不要である．また，書き込むプロセスが read timestamp まで待たずに即座に write timestamp を生成できるため，書き込みレイテンシが増大する問題も解消される．

Tardis のように論理タイムスタンプを用いることの問題点は，論理タイムスタンプの性質として並行発生したイベントを正確に表現できないため，実際には書き込まれていないキャッシュブロックを誤って無効化してしまい，キャッシュミス率が増大して性能低下する点である．正確な半順序を求めるにはベクトルタイムスタンプを用いる必要があり [44] , その場合は容量の観点からスケールしない．Tardis はタイムスタンプによる無効化に続いてミスが起きた時に read timestamp の再延長（“renewal”）を行うが，この renewal による性能低下を回避するため，renewal が完了する前に命令を投機的実行する機能や，キャッシュブロックごとに動的に適切な read timestamp の加算幅 (lease) を設定する機能などを備えている．

TreadMarks [9] [10] のようなソフトウェア DSM においては “Write notice” と呼ばれる手法が知られており，これもディレクトリを用いないキャッシュ無効化の一種である．緩和型コンシステンシにおいては同期アクセス時にだけ書き込み結果を伝搬すればよいという性質から，無効化メッセージに相当する書き込み通知（write notice）を release 側から acquire 側に集約して転送できる．acquire 側は release 側から write notice の配列を受け取り，その配列に acquire 側がキャッシュしているブロックがあれば無効化される．

Write notice はディレクトリを用いずにキャッシュ無効化を実現でき，論理タイムスタンプによるキャッシュ無効化とは異なって不要なキャッシュミスを発生させることはない．データを共有していないプロセスにも write notice が転送される問題点はあるが，ローカルプロセス内でキャッシュの存在判定を行うオーバーヘッドは小さいので問題になりにくい．Write notice を用いることの最大の問題点は，プログラムが進行すると Write notice の配列のエントリ数が増大し続けるため，容量面でスケールしにくい点である．TreadMarks においては，diff だけでなく write notice も大域的なガーベッジコレクションで破棄しており，ホームレスプロトコルと同様の問題を抱えていた．

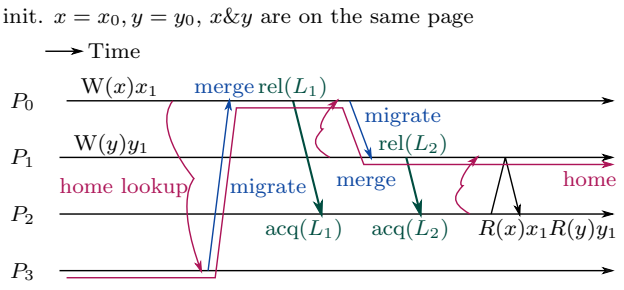


図 2: ホームマイグレーションの手法

3. 分散共有メモリの実装

3.1 ホームマイグレーション

本稿で開発した DSM 処理系は，前章で述べた書き込みレイテンシ低減や RDMA 化などの目的から，Multiple-Writer 型，ホームベース，Eager 型という設計上の選択肢を採用した．これらは Kaxiras ら [5] の処理系と同様である．また，一般的なホームベースプロトコルとは異なり，図 2 に示すように「書き込み時に常にホームマイグレーションする」という特徴的な仕組みを導入し，ホームベースプロトコルのホーム割り当ての問題を解決する．具体的には，通常のホームベースプロトコルがホームに diff を送信して書き込みを依頼するのに対し，提案手法ではデータ書き込み時に即座にホームが移動して移動先で diff の併合が行われる．また，従来のホームマイグレーションの手法 [29] [30] と同様に，Probable owner を用いてホームを検索する．

ホームマイグレーションを常に行う手法には次のような利点がある．

(1) 最新の書き込みを行ったノードが必ずホームになるので，プログラムの書き込みについて時間的局所性を仮定できるなら，再書き込み時のレイテンシを低減できると期待される．
(2) 書き込みを行ったノードがホームでない場合，通常のホームベースプロトコルでは diff の併合時にリモートメモリに書き込みを行う必要があり，単純に RDMA で実装すると細粒度の RDMA WRITE が多発して高速化が難しい．提案方式では，書き込み時にホームが必ずマイグレーションされるため，diff 併合は完全にローカルなメモリ操作のみで実現可能である．

従来のホームマイグレーションとの差異として，当時 RDMA 化は求められておらず，マイグレーションは必要に応じて行われていたという事情がある．

上述の手法の欠点は，マイグレーションの並列実行が困難なため，クリティカルセクションを用いて diff 併合を排他制御する必要がある点である．その場合 false sharing 時の性能が一般的なホームベースプロトコルと比べて低下する可能性があり，さらに書き込みレイテンシにクリティカルセクションのオーバーヘッドが加算されることになる．しかし，これらの欠点を加味しても，上述の利点の影響が大きいと判断したため，本研究ではホームマイグレーションを常に行う方式を採用した．

3.2 キャッシュブロック毎のクリティカルセクション処理

提案する処理系では，キャッシュブロック毎のクリティカルセクションを導入することで，以下の機能の実装を容易にする．

(1) ホームマイグレーション．ホームは同一ブロックに 1 プロセスだけしか存在できないので，その時点でのホームの情報を排他制御する必要がある．
(2) diff 併合処理．あるブロックへの diff の併合が並行に発生しないと保証できれば，SIMD 命令などで併合の並列化を行うことも容易になる．
(3) 論理タイムスタンプによるキャッシュ無効化．ホーム上に保持するタイムスタンプの状態を排他制御する必要がある．
(4) DSM 上でのアトミック命令の実装．キャッシュブロックへの書き込みを排他制御できると，任意の read-modify-write 命令を実装することが容易である．

ブロックに関するクリティカルセクションは，ロック，diff 併合処理，アンロックという順で実行される．まず，Probable owner を辿って前回のホームを特定し，そのホームからロックを取得する．ロックが取得でき次第，元のホームから RDMA READ によってタイムスタンプとデータの 2 つを読み取る．これらの情報に加えて，自ノード上のタイムスタンプとデータを用いて，diff 併合を実行する．さらに，DSM 上でのアトミック命令を実行する場合はここで実行される．最後にアンロックを行って，他プロセスが同じブロックのロックを取得できるようにする．

RDMA CAS によってロックを行う場合，単純なスピンロックによって実装すると，CAS が同一ノードに集中して性能が低下する．そのため，MCS ロック [45] に代表されるキューロックと呼ばれる種類のロックを参考に，リスト 1 のように分散キューロックを実装した．

まず，Probable owner の状態として「アンロックされたホーム (UNLOCKED)」「ロックされたホーム (LOCKED)」「他プロセスへのリンク 」の 3 種類を定義する．初期状態では 1 プロセスのみが UNLOCKED となっていて，他の全プロセスはそのノードへのリンクを持つ．ロック確保時に自プロセスがリンクを持っている場合は，自プロセスを LOCKED に書き換えた後，前述の通りに RDMA CAS を繰り返してリモートに対して「自プロセスへのリンク」への書き換えを試行し，CAS が成功すればロックを確保したことになる．アンロック時には自プロセスの Probable owner 値を LOCKED から UNLOCKED に書き換えるが，他のプロセスが「そのプロセスへのリンク」に書き換えて

```
const int UNLOCKED = 1, LOCKED = 2, LINK0 = 3;
int links[N_PROCS]; // 各プロセスが1要素ずつ持つ
int lock(int cur /* 自プロセスID */) {
  int l = links[cur];
  if (l == UNLOCKED) {
    l = CAS(&links[cur], UNLOCKED, LOCKED);
    if (l == UNLOCKED) return cur;
  }
  links[cur] = LOCKED;
  int p = l-LINK0;
  while (true) {
    l = CAS(&links[p], UNLOCKED, LINK0+cur);
    if (l == UNLOCKED) return p;
    else if (l == LOCKED) {
      l = CAS(&links[p], LOCKED, LINK0+cur);
      if (l == LOCKED) { Recv(p); return p; }
    }
    if (l >= LINK0) p = l-LINK0;
  }
}
void unlock(int cur) {
  int l = link[cur];
  if (l == LOCKED) {
    l = CAS(&links[cur], LOCKED, UNLOCKED);
    if (l == LOCKED) return;
  }
  Send(l-LINK0);
}
```

リスト 1: Probable owner と組み合わせたキューロックの擬似コード

表 1: XOR による diff 併合

		twin (t) / data (d)			
		00	01	11	10
home (h)	0	0	1	0	race
	1	1	race	1	0

出力 $h' = h \oplus t \oplus d$，データレース検出 $r = (h \oplus t) \cdot (d \oplus t)$

くる可能性があるので，MCS ロック同様にアンロックにも CAS を利用する．ロック衝突時には，先にロックしたプロセスが次プロセスにロックを直接転送する．このように分散メモリ上でロックを実装すると，ロック取得の試行中に同じ変数に RDMA CAS を繰り返す必要がなくなり，キューロック一般の性質としてスケーラビリティが向上する．なお，現在の実装ではロック転送を MPI のタグマッチングで実装しているが，RDMA WRITE で実装することも可能である．

表 1 に示すように，diff 併合処理はデータレースを無視することで XOR として実装可能 [4] であり，上述のホームマイグレーションの恩恵としてローカルメモリ操作のみになるので SIMD 化も容易である．さらに，提案するシステムでは，diff 併合の際に write-write 型のデータレースを検知することが可能で，具体的には表 1 において r が 1 となるかを調べている．この機能は，システムとユーザプログラムの双方における不正なメモリ操作を動的に検出するのに有用であり，提案する処理系の実装中にも複数のバグの解決に寄与した．一方で，この手法では read-write 型のデータレースを検出することはできない．

3.3 キャッシュ無効化の管理

本研究では，キャッシュ無効化の手法として，論理タイムスタンプと Write notice を併用する方式を提案する．論理タイムスタンプを用いる方式は通信回数や容量に関してスケーラブルであるが，キャッシュミスを増大させる問題がある．一方，Write notice はタイムスタンプベース同様にディレクトリが不要な上，キャッシュミス増大の問題もないが，容量の点でスケールしなかった．そこでこの 2 つを組み合わせて，容量とキャッシュミス数の問題を互いに解決し合うことが狙いである．

具体的な実装について以下に述べる．まず，プロセス間で happens-before 半順序に基づいて同期が必要になった際，“signature” [42] と呼ぶメタデータを転送して acquire 側のキャッシュ無効化に用いる．signature は要素数が一定以下の write notice の配列と，タイムスタンプベースの無効化に用いる minimum write timestamp (`min_wr_ts`) の組で構成される．さらに，write notice を (書き込んだプロセスの ID, ブロック ID, write timestamp, read timestamp) の 4 つの整数からなるタプルとして定義する．

各プロセスは自身の signature を管理しており，初期値として write notice の配列は空で，`min_wr_ts` も最小値となっている．そして，キャッシュブロックを release する度に自プロセスの signature に新しい write notice が追加される．ここで，signature には有限個の write notice しか記録しないため，溢れた場合に write notice を破棄する処理が必要となる．具体的には，signature 中で write timestamp が最小の write notice について，write timestamp と `min_wr_ts` の最大値を取って新たに `min_wr_ts` とし，そしてその write notice を破棄する．

acquire を行う際，release を行ったプロセスから signature が送られてくるので，その signature に基づいてキャッシュ無効化を行う．まず，write notice の配列に含まれているブロックについては，自プロセスのブロックが持つ read timestamp よりも write notice 中の write timestamp が大きい場合に無効化される．この際に，write notice の情報が自プロセスのブロック毎のメタデータとして保存され，後に同ブロックを再度読み出す時に用いられる．さらに論理タイムスタンプによるキャッシュ無効化を実行し，`min_wr_ts` に基づいてこれより小さい read timestamp を持つ全てのブロックを全て破棄する．この機能を実装するため，自プロセスにおいて読み出し可能な全てのブロックの ID を記録する priority queue を用意し，read timestamp

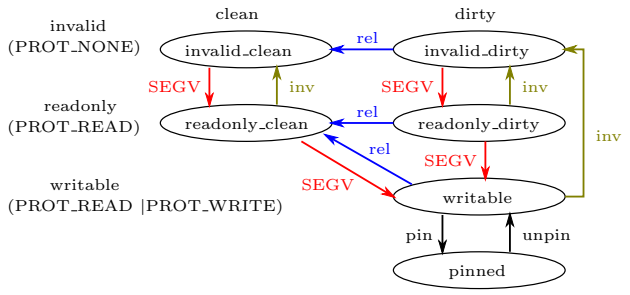


図 3: キャッシュブロックの状態遷移図

の小さい順に取り出せるようにすることで効率的にタイムスタンプによる無効化を行う．

signature の write notice を破棄しても依然としてコンシステンシが保たれる理由は，破棄した write notice の書き込みに関して `min_wr_ts` を更新しているため，タイムスタンプによるキャッシュ無効化が発生するからである．このため，古い write notice に関してはタイムスタンプによって無効化され，新しい write notice に関しては write notice が無効化することになる．

キャッシュ無効化後にキャッシュミスを起こしてデータを再度読む場合，論理タイムスタンプに基づいて無効化されたブロックは，上述のクリティカルセクションの仕組みを用いてホームから最新のデータを読み出す．タイムスタンプによって無効化されたデータは，どのプロセスによって書き込まれたかの情報が失われているため，最新のホームを検索する必要がある．一方で，write notice によって無効化された場合，happens-before 半順序に基づいてデータを読むと考えると，ホームからその時点で最新のデータを読み出す必要がなく，write notice を生成したプロセスから直接データを転送することが可能である．具体的には，write notice を生成したプロセスからデータを RDMA READ で読み出すだけでよく，ブロック毎のクリティカルセクションを利用する必要がない．そして，signature は新しい write notice を優先的に保持しており，新しい書き込みは write notice で無効化されて acquire 先で高速に読めるため，時間的局所性のあるプログラムで高速化が見込める．

タイムスタンプによって無効化されたブロックであっても，ホームが持つ read timestamp が有効であると確認できるなら，クリティカルセクションを実際に実行して read timestamp を更新する必要はない．しかし，現在は実装の単純化のため，キャッシュミス時にホームにアクセスする場合は必ずブロックのクリティカルセクションに入るよう実装している．

3.4 キャッシュブロックの状態遷移

図 3 に，キャッシュブロックの状態遷移図を示す．ブロックの状態は各プロセスごとに独立に保持されており，MSI のような一般的なプロトコルと同様に invalid, readonly, writable の 3 つの基本的な状態に分けられる．これらの状態に追加して，提案する処理系では clean/dirty への細分化，pinned 状態の追加の 2 つの改良を加えている．

clean と dirty の違いは，ホーム上への diff 適用が完了しているかを意味し，clean の場合は data と twin が一致しているが，dirty の場合は一致していない．通常の twinning を用いる DSM では dirty と writable を同一視し，writable の状態でデータを書き戻す場合はホームに diff 適用を行って readonly に遷移するものが多い．そのような手法では，acquire 時に writable ブロックが無効化されて invalid に遷移する際，data と twin を一致させなければ自プロセスの書き込みが失われるため，ホームに diff 適用をする必要がある．一方で，提案手法のように clean/dirty を区別すると，writable の時は単に invalid_dirty に遷移しておき，diff 適用は後で行うということが可能となる．そのため，acquire 時に diff 適用をせず，後に release が要求されたときだけ diff を適用することが可能となるので，acquire 時のレイテンシを低減することができる．

pinned 状態は以前提案した手法 [13] で，コールスタックを DSM で管理するために設けている．DSM が acquire や release などを実行する際，DSM 管理下のメモリ領域は `mprotect()` によってメモリ保護状態を変更される可能性がある．そのため，プロセス内のあるスレッドが DSM 上にコールスタックを配置して実行している最中に，DSM 処理系がそのコールスタックをアクセス禁止にすると，その時点で即座にそのスレッドはシグナルハンドラに突入することになる．一般にほとんどの標準ライブラリ関数はリエントラントでなく，シグナルハンドラ内で再度同じ関数を呼び出すことができない (async-signal-unsafe) が，SIGSEGV は同期シグナルの一種とされシグナルの発生時点は予測可能である．そのため，DSM 管理下のメモリ領域にアプリケーションプログラムが読み書きするだけなら，リエントラント性の問題は基本的に回避できる．しかし，仮に DSM に置かれたコールスタックへのアクセスがシグナルを発生させる場合，アプリケーションが呼び出す任意の標準ライブラリ関数の途中でシグナルハンドラに突入する可能性があるためにリエントラント性の要求が厳しくなる．実装上このような制約は不都合であるため，上述の pinned 状態を導入することで実行中のスレッドのコールスタックについて必ず読み書き可能な状態を保っている．

3.5 その他の実装上の問題

開発した DSM 処理系は 1 ノード 1 プロセスでプロセス内マルチスレッドを活用することを前提とし，プロセス内のスレッドスケジューリングにはユーザレベルスレッド処理系である MassiveThreads [46] を使用した．ユーザレベ

表 2: 評価環境 (ReedBush-U [48])

CPU	Intel® Xeon® E5-2695 v4
	2.1 GHz (max. 3.3 GHz with Turbo boost)
	18 cores × 2 sockets / node
メモリ	256GB / node
インターコネクト	Mellanox® Connect-IB® dual port
	InfiniBand EDR 4x
OS	Red Hat® Enterprise Linux® 7.2
コンパイラ	GCC 4.8.5 ("-O3" オプション)
MPI	MVAPICH 2.2

表 3: DSM ライブラリのパラメータ

グローバル変数のブロックサイズ	32 KiB
コールスタック領域のブロックサイズ	16 KiB
コールスタックのサイズ	16 KiB
read timestamp 生成時の lease 値	10
write notice 配列の長さ	最大 1024 要素

ルスレッドを用いることで，DSM 内部で複数のキャッシュブロックを並列処理したり，アプリケーションの計算中にバックグラウンドで DSM の処理を実行したり，キャッシュミス時の通信を隠蔽したりといった各種高速化が可能となる．

今回は API として OpenMP を用いており，コンパイラに付属している OpenMP ランタイムを用いずに，barrier などの基本的機能のみを実装した．具体的な実装方法として，アプリケーションのコンパイル時に OpenMP を有効にして，リンク時に提案する処理系を代わりにリンクする．元々のランタイムと ABI を揃えることによって，ライブラリのみの手法で OpenMP プログラムをそのまま DSM 上で実行可能となる．

DSM 処理系のデバッグの過程で，シグナルハンドラ内で MassiveThreads の関数を呼び出してコンテキストスイッチを行うと結果が不正になるバグが発生したため，現在はシグナルハンドラ内でスピンロックのみを利用し，コンテキストスイッチを無効化してこのバグを回避している．しかし，この回避策は以前開発したソフトウェアオフローディング機構 [47] と組み合わせた際に著しく性能が低下するため，この機構も今回は無効化して MPI を単に呼び出すだけとした．また，MPI の thread level を `MPI_THREAD_MULTIPLE` にして実行すると RMA の実行結果が異常な値を返す場合があったため，全ての MPI 呼び出しもスピンロックで排他制御している．

4. ベンチマークによる評価

評価には，NAS Parallel Benchmark 3.0 の C 言語への移植版 [15] を使用した．公式版は Fortran で記述されているため今回は非公式版を使用したが，公式版にも対応を検討中である．評価環境を表 2 に，評価に用いた DSM ライブラリのパラメータを表 3 に示す．

ベンチマークは OpenMP で記述されており，通常の OpenMP ランタイムで動作させた結果と，OpenMP ランタイムをすり替えて DSM で動作するようにした場合の結果を比較する．OpenMP を無効化した逐次実行の結果も併せて測定した．

今回の評価には NAS EP，CG，BT の 3 つを用いたが，EP と CG には OpenMP の reduction 演算が含まれており，GCC がこれをプロセッサのアトミック命令として出力するという事情から，現時点で DSM 上でそのまま実行することはできていない．そのため，reduction に関しては DSM 上のグローバル変数にスレッドごとの結果を一旦保存し，マスタースレッドがそれらを逐次に reduction するよう変更を加えている．また，DSM 上のグローバル変数を threadprivate とすることは現時点でできていないため，EP において threadprivate を削除して shared に変え，copyin 句の代わりに手動で memcpy を行いローカルのバッファにコピーするよう変更した．

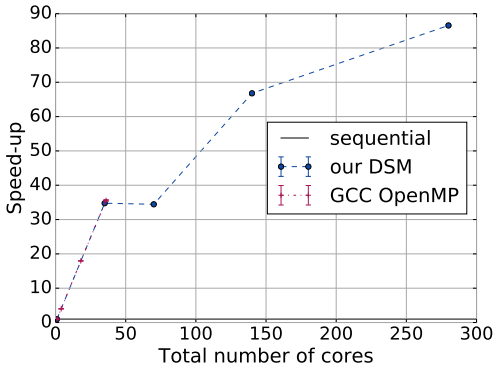
実験結果のスピードアップは図 4 に示す通りで，逐次実行 (sequential) の結果が 1 となっている．NAS EP の結果については GCC OpenMP の結果よりもマルチノード時の性能を向上させることができたが，EP は Embarrassingly Parallel の略であり並列化が容易なアプリケーションである．CG については 1 ノード内での実行結果が GCC OpenMP よりも低速で，マルチノードでの性能もそれ以上向上していない．BT に至っては逐次実行時よりも逆に性能が低下するという状況になっており，深刻なオーバーヘッドが発生していることが分かった．

現在はこの評価結果の改善に取り組んでいる最中であるが，既に明らかになっている問題としては，MPI のマルチスレッド性能の問題や，OpenMP バリア同期時に DSM でデータを同期するための `mprotect()` システムコールによるオーバーヘッドの問題などがある．いずれの問題も実装の工夫で改善の余地があるため，今後取り組む予定である．

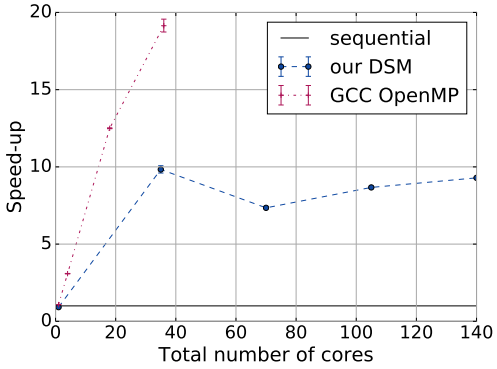
5. 結論

本稿では，論理タイムスタンプに基づく分散共有メモリライブラリの実装手法を提案し，実際に分散メモリ型計算機用に実装してベンチマークアプリケーションによる評価を行った．現在の評価結果では実用的な性能に達しているとは言い難く，性能上の課題は残されているが，今後マイクロベンチマークや他処理系との比較を重ねながらボトルネックの解消に取り組んでいく．今回の実装では以前開発したソフトウェアオフローディング機構 [47] を用いていないが，通信のマルチスレッド性能向上に有用であり，今後評価結果に組み込む予定である．

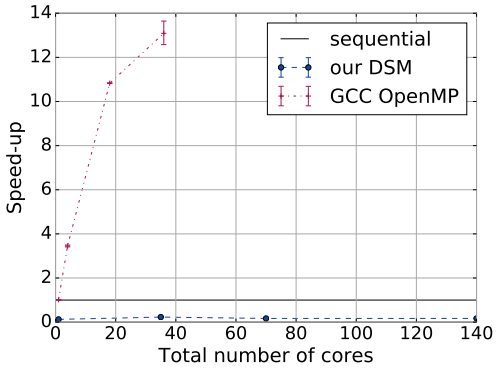
謝辞 本研究の一部は科研費基盤研究 (A) 16H01715 の助成を受けて行われている．



(a) NAS EP (CLASS=C)



(b) NAS CG (CLASS=C)



(c) NAS BT (CLASS=A)

図 4: 開発した DSM ライブラリと GCC OpenMP との性能比較

参考文献

[1] Culler, D. E., Gupta, A. and Singh, J. P.: *Parallel Computer Architecture: A Hardware/Software Approach* (1999).

[2] Protic, J., Tomasevic, M. and Milutinovic, V.: Distributed Shared memory: Concepts and Systems, *IEEE Parallel and Distributed Technology*, Vol. 4, No. 2, pp. 63–77 (online), DOI: 10.1109/88.494605 (1996).

[3] Ramesh, B., Ribbens, C. J. and Varadarajan, S.: Is it time to rethink distributed shared memory systems?, *ICPADS 2011: Proceedings of the International Conference on Parallel and Distributed Systems*, pp. 212–219 (online), DOI: 10.1109/ICPADS.2011.75 (2011).

[4] Ramesh, B.: Samhita: Virtual Shared Memory for Non-Cache-Coherent Systems (2013).

[5] Kaxiras, S., Klaftenegger, D., Norgren, M., Ros, A. and Sagonas, K.: Turning Centralized Coherence and Distributed Critical-Section Execution on their Head, *HPDC '15: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ACM Press, pp. 3–14 (online), DOI: 10.1145/2749246.2749250 (2015).

[6] Yu, X. and Devadas, S.: Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory, *PACT '15: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, IEEE, pp. 227–240 (online), DOI: 10.1109/PACT.2015.12 (2015).

[7] Yu, X., Liu, H., Zou, E. and Devadas, S.: Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models, *PACT '16: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ACM Press, pp. 261–274 (online), DOI: 10.1145/2967938.2967942 (2016).

[8] Yu, X.: Logical Leases : Scalable Hardware and Software Systems through Time Traveling, PhD Thesis, Massachusetts Institute of Technology (2017).

[9] Keleher, P., Cox, A. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *ISCA '92: Proceedings the 19th Annual International Symposium on Computer Architecture*, IEEE, pp. 13–21 (online), DOI: 10.1109/ISCA.1992.753300 (1992).

[10] Keleher, P., Cox, A. L., Dwarkadas, S. and Zwaenepoel, W.: TreadMarks : Distributed Shared Memory on Standard Workstations and Operating Systems, *WTEC '94: Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 115–132 (online), DOI: 10.1.1.85.8057 (1994).

[11] Zhou, Y., Iftode, L. and Li, K.: Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, *OSDI '96: Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, New York, New York, USA, ACM Press, pp. 75–88 (online), DOI: 10.1145/238721.238763 (1996).

[12] Li, K.: IVY: a shared virtual memory system for parallel computing, *ICPP '88: Proceedings of the 1988 International Conference on Parallel Processing*, pp. 94–101 (1988).

[13] 遠藤　亘，田浦健次朗：分散スレッドスケジューラと協調する分散共有メモリ処理系の初期評価，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2017-HPC-1, No. 37, pp. 1–10 (2017).

[14] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V. and Weeratunga, S.: The NAS Parallel Benchmarks, *The International Journal of Supercomputing Applications*, Vol. 5, No. 3, pp. 63–73 (online), DOI: 10.1177/109434209100500306 (1991).

[15] University of Versailles Saint Quentin en Yvlines: NAS-C-OpenMP3.0, `http://benchmark-subsetting.github.io/cNPB`.

[16] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers*, Vol. 28, No. 9, pp. 690–691 (online), DOI: 10.1109/TC.1979.1675439 (1979).

[17] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiproces-

sors, *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26 (online), DOI: 10.1109/ISCA.1990.134503 (1990).

[18] Bershad, B. N., Zekauskas, M. J. and Sawdon, W. A.: The Midway distributed shared memory system, Technical report (1993).

[19] Iftode, L.: Scope Consistency: A Bridge between Release Consistency and Entry Consistency, *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, Vol. 31, No. 4, pp. 451–473 (online), DOI: 10.1007/s002240000097 (1998).

[20] Blumofe, R., Frigo, M., Joerg, C., Leiserson, C. and Randall, K.: Dag-Consistent Distributed Shared Memory, *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pp. 132–141 (online), DOI: 10.1109/IPPS.1996.508049 (1996).

[21] Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E. and Randall, K. H.: An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms, *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pp. 297–308 (online), DOI: 10.1145/237502.237574 (1996).

[22] Adve, S. V. and Hill, M. D.: Weak ordering - a new definition, *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, Vol. 18, No. 3, pp. 2–14 (online), DOI: 10.1145/325096.325100 (1990).

[23] Batty, M., Owens, S., Sarkar, S., Sewell, P. and Weber, T.: Mathematizing C++ Concurrency, *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 55 (online), DOI: 10.1145/1926385.1926394 (2011).

[24] Cheong, H. and Veidenbaum, A. V.: Compiler-Directed Cache Management in Multiprocessors, *Computer*, Vol. 23, No. 6, pp. 39–47 (online), DOI: 10.1109/2.55499 (1990).

[25] Carter, J. B., Bennett, J. K. and Zwaenepoel, W.: Implementation and Performance of Munin, *SOSP '91: Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Vol. 25, pp. 152–164 (online), DOI: 10.1145/121133.121159 (1991).

[26] Zhao, H., Shriraman, A., Kumar, S. and Dwarkadas, S.: Protozoa: Adaptive Granularity Cache Coherence, *ISCA '13: Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 547–558 (online), DOI: 10.1145/2485922.2485969 (2013).

[27] Ros, A. and Kaxiras, S.: Complexity-Effective Multicore Coherence, *PACT '12: Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM Press, p. 241 (online), DOI: 10.1145/2370816.2370853 (2012).

[28] Yu, B., Huang, Z. and Cranefield, S.: Homeless and home-based lazy release consistency protocols on distributed shared memory, *ACSC '04: Proceedings of the 27th Australasian Conference on Computer Science*, Vol. 26, pp. 117–123 (online), DOI: 10.1.1.112.2373 (2004).

[29] Chung, J. W., Seong, B. H., Park, K. H. and Park, D.: Moving Home-Based Lazy Release Consistency for Shared Virtual Memory Systems, *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, p. 282 (1999).

[30] 岡本秀輔，阿部拓弥：ソフトウェア分散共有メモリにおけるアクセス履歴に基づくホーム移動，情報処理学会論文誌：コンピューティングシステム，Vol. 45, No. May, pp. 66–74 (2004).

[31] Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–359 (online), DOI: 10.1145/75104.75105 (1989).

[32] 原健太朗：再構成可能な高性能並列計算のための PGAS プログラミング処理系 (2011).

[33] Zebchuk, J., Srinivasan, V., Qureshi, M. K. and Moshovos, A.: A Tagless Coherence Directory, *MICRO-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, p. 423 (online), DOI: 10.1145/1669112.1669166 (2009).

[34] Ferdman, M., Lotfi-Kamran, P., Balet, K. and Falsafi, B.: Cuckoo directory: A scalable directory for many-core systems, *HPCA '11: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 169–180 (online), DOI: 10.1109/HPCA.2011.5749726 (2011).

[35] Sanchez, D. and Kozyrakis, C.: SCD: A scalable coherence directory with flexible sharer set encoding, *HPCA '12: Proceedings of the 18th International Symposium on High Performance Computer Architecture*, IEEE, pp. 1–12 (online), DOI: 10.1109/HPCA.2012.6168950 (2012).

[36] R. Lebeck, A. and A. Wood, D.: Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors, *ISCA '95: Proceedings of 22nd Annual International Symposium on Computer Architecture*, pp. 48–59 (online), DOI: 10.1109/ISCA.1995.524548 (1995).

[37] Cuesta, B., Ros, A., Gomez, M. E., Robles, A. and Duato, J.: Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Noncoherent Memory Blocks, *IEEE Transactions on Computers*, Vol. 62, No. 3, pp. 482–495 (online), DOI: 10.1109/TC.2011.241 (2013).

[38] Esteve, A., Ros, A., Gomez, M. E., Robles, A. and Duato, J.: Efficient TLB-Based Detection of Private Pages in Chip Multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, No. 3, pp. 748–761 (online), DOI: 10.1109/TPDS.2015.2412139 (2016).

[39] Choi, B., Komuravelli, R., Sung, H., Smolinski, R., Honarmand, N., Adve, S. V., Adve, V. S., Carter, N. P. and Chou, C.-t.: DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism, *PACT '11: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 155–166 (online), DOI: 10.1109/PACT.2011.21 (2011).

[40] Khan, O., Devadas, S., Shim, K. S., Cho, M. H. and Lis, M.: Library Cache Coherence (2011).

[41] Singh, I., Shriraman, A., Fung, W. W. L., O'Connor, M. and Aamodt, T. M.: Cache coherence for GPU architectures, *IEEE Micro*, Vol. 34, No. 3, pp. 69–79 (online), DOI: 10.1109/MM.2014.4 (2014).

[42] Yao, Y., Chen, W., Mitra, T. and Xiang, Y.: TC-Release++: An Efficient Timestamp-Based Coherence Protocol for Many-Core Architectures, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 28, No. 11, pp. 3313–3327 (online), DOI: 10.1109/TPDS.2017.2719679 (2017).

[43] Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565 (online), DOI: 10.1145/359545.359563 (1978).

[44] Charron-Bost, B.: Concerning the size of logical clocks

in distributed systems, *Information Processing Letters*, Vol. 39, No. 1, pp. 11–16 (online), DOI: 10.1016/0020-0190(91)90055-M (1991).

[45] Mellor-Crummey, J. M. and Scott, M. L.: Synchronization without contention, *ACM SIGARCH Computer Architecture News*, Vol. 19, No. 2, pp. 269–278 (online), DOI: 10.1145/106972.106999 (1991).

[46] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond*, Vol. 8665, pp. 222–238 (online), DOI: 10.1007/978-3-662-44471-9 (2014).

[47] Endo, W. and Taura, K.: Parallelized Software Offloading of Low-Level Communication with User-Level Threads, *HPC Asia 2018: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, Vol. Part F1346, New York, New York, USA, ACM Press, pp. 289–298 (online), DOI: 10.1145/3149457.3149475 (2018).

[48] 東京大学 情報基盤センター：Reedbush スーパーコンピュータシステム 利用手引書 1.7.