Vol.2017-HPC-160 No.37
2017/7/28

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

遠藤 亘[1,a)]　田浦 健次朗[1,b)]

概要：本研究では，分散共有メモリ (Distributed Shared Memory, DSM) システムと分散スレッドスケジューラを統合したライブラリを開発し，並列分散環境において透過的でスケーラブルな共有メモリプログラミングを実現することを目指している．従来の DSM 処理系で問題となってきたコヒーレントキャッシュの低スケーラビリティを改善するため，スレッド依存関係に基づいた緩和型コンシステンシモデルを基本として，計算ノードをまたいだ動的負荷分散とコヒーレンスプロトコルによる通信を協調させる手法を導入する．DSM の実装において今回はページベース DSM とし，ディレクトリベースのコヒーレンスプロトコルを実装する．分散スレッドスケジューラにはユーザレベルスレッドを用い，DSM 上にコールスタックを配置することで透過的スレッド移動を実現する．このような実装手法により，利用者にはユーザレベルスレッドやヒープ領域の生成・破棄等の API が提供され，マルチコアプロセッサ上のタスク並列処理系相当の生産性を分散環境において実現できる．また，並行開発した HPC インターコネクト用の低水準通信ライブラリを基礎として，RDMA の利用を踏まえた DSM とスケジューラの実装を行う．本稿では，開発した処理系において共有メモリのベンチマークプログラムを動作させて初期評価を行い，その結果を元に性能上の今後の課題について論ずる．

Wataru Endo[1,a)]　Kenjiro Taura[1,b)]

1. 序論

分散メモリシステムにおけるアプリケーション生産性の問題は数十年に渡って研究されているが，高生産性と高性能を両立できるプログラミングモデルとその実装手法の確立には未だ至っていない．本稿で着目する分散共有メモリ (DSM) [1] [2] は，分散メモリシステム上で共有メモリモデルを提供する手法であり，高いプログラミング生産性を実現できるが，現在に至るまで数百ノード以上でスケールする実装例は登場していない．そのため，現在では DSM に代わり，共有メモリの特徴であるグローバルアドレス空間を提供する一方でコヒーレントキャッシュを持たない PGAS [3] [4] [5] [6] が主に研究されている．PGAS はスケーラブルな実装が比較的容易であるが，キャッシュを持たないために DSM に比べてユーザへの負担が大きい．

筆者らは以前, MassiveThreads/GAS 2 [7] という, PGAS に手動のキャッシュ機構を加えた API とシステムを開発していた．このシステムはコヒーレンスを自動管理しないため，キャッシュを破棄したり書き出したりするタイミングをユーザがどう判断すべきかが問題であった．この問題を克服し，より一般的なプログラミングモデルに近づけるため，本研究では PGAS ではなくあえてソフトウェア DSM に着目することとした．2010 年代に再び DSM に取り組むにあたっては，従来のソフトウェア DSM 研究とは異なってマルチコアプロセッサ研究の知見やスレッドスケジューリングの観点を積極的に取り入れるという戦略を取っている．

本研究では，透過的な分散スレッドスケジューリングを実現するために共有メモリ機構が本質的に必要である，という観点から，ソフトウェア DSM を前提に分散スレッドスケジューラを組み合わせたシステムを設計し実装した．現時点の処理系は最低限のバグを取り除いたという状況であり，実用的な性能が出る段階ではないが，今後の実装方針を明確にするために現状のプロトタイプの性能評価を行った．

[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

2. 関連研究

2.1 分散共有メモリ (DSM)

DSM は「全てのメモリ階層が分散している共有メモリ処理系」と定義できる．DSM において，各計算ノード間のキャッシュはコヒーレンスプロトコルによる自動的な通信によって管理される．1990 年代頃は DSM が盛んに研究されており，当時のハードウェア DSM の代表例として DASH [8]，ソフトウェア DSM の代表例として Ivy [9]，Munin [10]，TreadMarks [11] [12] などが知られている．

DSM のような共有メモリシステムにおいてプログラミングを行うには，メモリコンシステンシモデルを定義することが不可欠である．コンシステンシモデルは，共有メモリシステムにおいて各コアがメモリから読み出せる値を決定する規則であり，実用上最も厳しい制約は Sequential Consistency (SC) [13] である．SC はリオーダリング等の各種最適化技法を妨げるため，DSM が想定するような大規模環境では様々な緩和型コンシステンシモデルが提案されており，代表的なものに Release Consistency (RC) [14] [11]，Entry Consistency [15]，Scope Consistency [16] などが挙げられる．また，スレッド間の依存性をコンシステンシモデルとして取り入れた DAG Consistency [17] [18] も知られている．

ソフトウェア DSM においてキャッシュの存在判定を行うための手法として，MMU のメモリ保護機構を用いる手法（ページベース DSM） [9] と，メモリアクセス毎に特殊な命令を挿入する手法（コンパイラベース DSM） [19] が存在する．ページベース DSM はキャッシュヒット時のコストを無くせる利点があるが，キャッシュミス時には OS が介在するためオーバーヘッドが大きい欠点がある．逆に，コンパイラベース DSM はキャッシュヒット時にも場合によってはコストが発生するが，キャッシュミス時のコストは小さくできる．

キャッシュコヒーレンスを保つためには，あるキャッシュブロックの所在や，それを共有しているノード（シェアラー）の有無を調べる必要がある．そのための手法は，“ディレクトリ” という専用のデータ構造にそれらを記録するディレクトリベースプロトコルと，ブロードキャストを用いるスヌーピープロトコルの 2 つに大別される．DSM は大規模なネットワークトポロジを前提とするため，通常はディレクトリベースが採用される．

2.2 Self-invalidation/downgrading と DRF 仮定

共有メモリシステムのスケーラビリティを向上させる手法の一つが，Self-invalidation [20] である．Self-invalidation とはシェアラーが自発的に自身のキャッシュを捨てることで，キャッシュ置換とは無関係に実行される．ディレクトリベースプロトコルを考えると，一般的に書き込みの開始前にまず全ノード上の同じキャッシュブロックを invalidate する必要がある．この際，シェアラーが自発的に Self-invalidation を行っておけば，書き込み時の invalidation メッセージ送信が不要となり，書き込みレイテンシが低減できる．

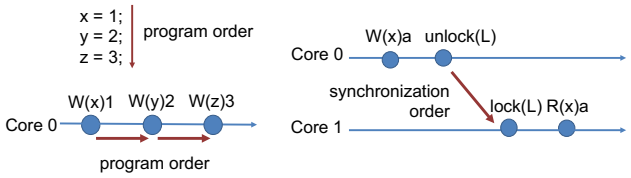

図 1 happens-before 半順序の定義

近年のマルチコアプロセッサ研究では，ユーザプログラムの Data-Race-Free (DRF) [21] を前提としたコヒーレンスの簡略化が注目されており，例として DeNovo [22] や VIPS-M [23] が挙げられる．DRF の保証は C++11 のメモリモデル [24] にも導入され，最近の言語では一般的になっている．

まず，DRF を議論するのは緩和型コンシステンシの場合であり，メモリアクセスは同期/非同期に分類される．次に，図 1 のようにプログラム記述順 (program order) と同期順 (synchronization order) の和として happens-before 半順序を定義する．synchronization order とは，例えばミューテックスの unlock/lock であったり，スレッドの待ち合わせであったりというような，ノード間の同期が発生した時の順序付けである．happens-before 半順序を利用すると，非決定的な非同期メモリアクセスをデータレースとして定義できる．DRF とは，データレースが発生しないとユーザが保証していることを意味する．

ユーザプログラムの DRF を仮定して，かつ同期アクセス時にシェアラーが “全てのキャッシュブロックに対して必ず Self-invalidation を実行する” と取り決めれば，（非同期アクセスによる）書き込み時に他のシェアラーに対して invalidation メッセージを送信する必要性がなくなる．この事実が，Denovo や VIPS-M がディレクトリ操作を簡略化できることの根拠である．また，読み込み側についても同様の考え方が成り立ち，自発的にキャッシュの書き戻しを行う手法 (Self-downgrading) として対になっている．具体的には，図 1 では unlock 時に Self-downgrading を，lock 時に Self-invalidation を，それぞれのプロセスで全キャッシュブロックに対して行えばコヒーレンスが保たれる．

Kaxiras ら [25] は，DRF を前提とした DSM である Argo DSM を提案している．Argo DSM は DRF と Self-invalidation/downgrading を組み合わせてソフトウェア DSM を実装したという点で特徴的で，提案する処理系もこの点では同様である．Argo DSM では共有されていないブロックに対して Self-invalidation を避けるため，各シェ

アラーがディレクトリの複製を管理していて，他のシェアラーが共有しているかどうか記録している．Kaxiras らはディレクトリ操作の RDMA 化も重視しているが，実装の単純化のため本研究では現在この点については注力しないこととした．

2.3 DSM のスケーラビリティ

筆者らが認識している限りにおいて，DSM あるいはコヒーレントキャッシュがスケーラブルでない原因として挙げられるものは次の 6 つである．

(1) **データやディレクトリへのアクセスが特定のプロセスに集中する．**MPI のブロードキャストのような通信パターンを共有メモリで直接表現することは困難であり，素直に記述すると通信が一箇所に集中する問題がある．この問題の解決策としては，原ら [26] [27] が実装したような Probable Owner [28] のような手法に基づく階層的キャッシュ構造が挙げられるが，レイテンシ増大と複雑性が新たな課題となる．本研究の現行実装はこうした手法には対応していないが，将来的な導入を検討中である．

(2) **アドレス空間が不足している．**純粋に実装上の問題であり，現在の 64-bit CPU においてはあまり問題にはならない．ページベース DSM においては，マルチスレッド対応させながらアドレス空間を増やす手法として fork() を用いた手法 [29] が知られている．

(3) **メモリ保護機構を用いることによってオーバーヘッドが増大する．**これはページベース DSM 特有の問題であり，コヒーレントキャッシュ全般に言えるものではない．

(4) **厳しいメモリコンシステンシモデルを採用している．**DSM においては緩和型コンシステンシモデルの研究が進んでいるが，SC を前提とした処理系も多かった．

(5) **キャッシュの複製数に制限がある．**False sharing における性能低下は，Multiple-Reader Multiple-Writer (MRMW) 型プロトコルによって解決される．

(6) **ノードが増えるほど，invalidation メッセージを送る先のシェアラー数が増大する．** Self-invalidation の積極的な実行により解決される．

このように，様々な理由で DSM にはスケーラビリティ低下の可能性があるものの，それらいずれについても解決不可能とまではいえない．コヒーレントキャッシュがスケーラブルであると主張する研究例には，Martin ら [30] によるものがある．

2.4 共有メモリ型計算機上のユーザレベルスレッド処理系

ユーザレベルスレッド (user-level threads) 処理系[*1] とは，OS の “スレッド” に相当する機能をユーザレベルで実装した処理系であり，各（ユーザレベルの）スレッド毎にスレッドデスクリプタとコールスタックを用意して管理する．スレッドのスケジューリングにはワークスティーリング [31] を用いることが一般的である．代表的なユーザレベルスレッドライブラリには，MassiveThreads [32] [33] [34]，QThreads [35]，Argobots [36], TBB [37]，Nanos++ [38]，Boost.Fiber [39] などがある．

ユーザレベルスレッドのように，コールスタックごと継続を保存する手法を Stackful Coroutines [40] とも呼ぶ．これに対して，コールスタックを保存せずに継続を保存する手法を Stackless Coroutines [41] [42] と呼ぶ．C 言語においてコールスタックの扱いは変更不能であるという事情から，Stackful Coroutines はライブラリのみで実装可能である一方，Stackless Coroutines にはコンパイラの支援が必須である．

2.5 分散メモリ型計算機上のユーザレベルスレッド処理系

分散メモリ型計算機上でワークスティーリングスケジューラを実現する場合，最初に問題となるのはコールスタックの扱いである．コンパイラが生成するコードには，コールスタック内に同じコールスタック内のデータを指すポインタが存在しており，スレッドがプロセス間で移動する際に問題となる．コンパイラ支援なしでユーザレベルスレッドをプロセス間で移動する手法としては，iso-address [43] が知られている．iso-address とは「全プロセス上の各ユーザレベルスレッドのコールスタックについて，全プロセスで共通に一意のアドレスを割り当て，プロセス間での移動時にはコールスタックを元のプロセス上と同じアドレスにコピーする」という手法である．iso-address はグローバルアドレス空間を必要とする点で共有メモリ的な発想であり，iso-address を共有メモリの文脈で捉え直すと「コールスタック専用の緩和型コンシステンシ」であるとみなせる．iso-address を用いた分散メモリ向けライブラリとしては，Charm++ [44] が例として挙げられる．

秋山らの開発した MassiveThreads/DM [45] [46] は，ユーザレベルスレッドを PGAS と組み合わせた分散スレッド処理系である．ユーザレベルスレッドをプロセス間で移動するために，秋山らは全てのコールスタックを同じアドレスに配置する uni-address という手法を提案している．uni-address は仮想アドレスを節約できる利点があるが，スレッドを停止/再開する度にコールスタック全体をコピーする必要があるという欠点がある．より重大な問題として，uni-address はコールスタックへのポインタが使用できないというプログラミング上の制約がある．そのため，共有メモリシステムではポインタを介した間接参照だった処理

[*1] ユーザレベルスレッドと同様の概念を示す用語は乱立しており，軽量スレッド (lightweight threads) や ファイバー (fibers) などがある．より曖昧な用語として タスク並列処理系 (task-parallel systems) とも呼ばれることがある．

を，uni-address 上ではヒープに相当するグローバル領域アクセスに書き換える必要がある．同じようにエイリアシングを利用した仮想アドレス空間の節約手法には，原ら [27] が提案した random-address という，コールスタックの仮想アドレスをランダム化する手法がある．random-address も uni-address 同様のプログラミング上の制約を抱えている上，アドレス衝突時時に新規プロセスを生成する必要があるなどシステム実装の上でも難点が多い．

ユーザレベルスレッドによる仮想アドレスの大量消費は，そもそも分散メモリに限った問題ではなく，ユーザレベルスレッドが Stackful であることが根本的な原因である．より一般的な仮想アドレスの節約手法としては Split Stacks [47] や Stackless Coroutines のようなコンパイラベースの手法があり，これらにはエイリアシングと比較してプログラミング上の制約がない．

DSM は共有メモリシステムの一種なのでコールスタックを管理可能であり，本研究では DSM にコールスタックを管理させることとした．同様の手法に基づく既存研究には Distributed Cilk [17] や SilkRoad [48] [49] が挙げられる．これらは DSM にコールスタックを配置する点は本研究と同様であるが，Self-invalidation のような高速化技法は使用されておらず，現在のハードウェア性能は当時よりも向上していてソフトウェアへの要求も厳しいことが違いとして挙げられる．

DSM にコールスタックを管理させる目的は，実装コストの削減と，システム全体としての簡潔性の確保である．グローバルなヒープについてキャッシュ機構が必要であるとすれば，MassiveThreads/DM のようにスタック領域専用のメモリ管理機構を導入することはシステム全体として機能が重複していて，ユーザには複雑なコンシステンシモデルとして見えることとなる．スタックとヒープ両方とも，必要に応じてキャッシュすべきメモリ領域であることには変わりなく，単一のコンシステンシモデルに基づいたコヒーレントキャッシュとして管理する方がシステムにもユーザにも扱いやすい．

3. 分散共有メモリの実装

本章では，提案する分散共有メモリの実装について述べる．前提として，1 ノードあたり 1 プロセスのハイブリッド並列実行を想定しており，DSM やスケジューラも含めて全てマルチスレッド化されている．基本的な設計としては Argo DSM [25] を踏襲しているが，スケジューラとの連携を踏まえた拡張などが異なっている点である．

筆者らは，以前からマルチスレッド環境で高速動作する低水準通信ライブラリ [50] の開発について取り組んでおり，本稿で提案する処理系はこれを利用して実装されている．但し，通信ライブラリに関しては主に RDMA の高速化に注力したため，Active Messages (AM) の実装は MPI を利用した簡易的な実装に留まっており，現時点では十分にチューニングされていない．

```
void SD_fence(); // Self-downgrading fence
void SI_fence(); // Self-invalidation fence
```

リスト 1 フェンス命令のインターフェイス

3.1 DSM の API

まず，提案する DSM 処理系においては，デフォルトの言語やシステムの機能を DSM が乗っ取ることはない．例えば，malloc を差し替えて DSM 用のメモリ領域を返却する，といったことは技術的には可能であるが，現在の実装では行っていない．従って，DSM 用にユーザプログラムを書き換える必要はあるが，その書き換えは（既に DRF を保証していれば）アロケーション関数の置換程度で済む．

初めに DSM からメモリ領域を確保する際は，セグメントという単位で連続したアドレス領域をユーザから確保する．それらの各セグメントに対し，一般的な共有メモリで用いられるアロケータを使えば，malloc/free 相当のヒープ領域の機構を提供できる．後述するスケジューラにおけるコールスタック領域も，基本的にはヒープと同様に DSM のセグメントの一つとしてまとめて管理される．また，DSM 上にグローバル変数を配置することも可能となっており，全ノードで共有したいグローバル変数について GCC の section 属性を使って特殊なセクションに置き，リンカスクリプトを用いて特殊な扱いをすることでこれを実現している．

コンシステンシモデルとしては DRF を前提とした緩和型であり，リスト 1 に示すように Argo DSM [25] と同様に 2 つのフェンス命令（フルフェンス）を導入する．SD_fence は，自プロセス上でキャッシュされている全てのキャッシュブロックに行った全ての変更について，全てのプロセスから可視にする．SI_fence は，DSM によって管理される全てのキャッシュを無効化し，それ以前に他プロセスが SD_fence で適用した全ての変更が，以降の読み出しでは可視になることを保証する．

今回の評価では用いなかったが，Argo DSM になかった API として，コールバック関数を渡す「非同期フェンス」も実装し，動作を確認している．具体的な用途としては，スレッド移動のレイテンシ短縮が挙げられる．（分散スケジューラ上の）あるユーザレベルスレッドが一旦中断された時，その時点で非同期の SD_fence を実行しておくことで，その後スレッドが他プロセスに移動する際に SD_fence を実行しなくて済む．実際にこの手法を実装したものの，現状では逆に全体性能が低下したため，評価では用いていない．DSM の実装がノード内で十分に並列化されていないことが原因と予想しており，今後再検証する予定である．

3.2 DSM の設計

DSM のキャッシュの存在判定にはページベース DSM を用いている．設計上の選択肢としてはコンパイラベース DSM も考慮したが，早期のプロトタイピングを目指した場合にページベースが現実的であると判断したためである．また，Multiple-Reader Multiple-Writer (MRMW) 型のコヒーレンスプロトコルとし，複数の書き込みプロセスからの変更を diff によって併合する．キャッシュディレクトリは各キャッシュブロックに存在し，reader/writer を別個に分けて記録している．これらの基本的方針も Argo DSM と同様である．

あるブロックに対する各プロセスの役割を以下の 3 つに分けて解説する．

- **マネージャ (manager)**：キャッシュディレクトリを管理するプロセス．現在の実装では，キャッシュディレクトリはフラットな構成になっており，一つのブロックに対する全てのシェアラーを単一のマネージャノードで追跡している．キャッシュブロックに対するマネージャは静的に割り振られる．
- **オーナー (owner)**：ホームベース DSM [51] でいうホームに相当するプロセス．オーナーは diff を書き出す先のブロックを持っている．オーナーのブロックを指すアドレスはマネージャに記録されていて，ブロックへの初回アクセス時にオーナーが動的に割り当てられる．
- **シェアラー (sharer)**：キャッシュにアクセスしているプロセス．

これらの役割は 1 つのプロセスに複数当てはまる場合もあり，その場合はリモートプロセスに通信を発行する必要がなくなるので，性能が向上することがある．

3.3 キャッシュブロックの状態遷移

キャッシュブロックの共有状態は，基本的には MSI プロトコルと同様に Modified（読み書き可能），Shared（読み込み可能），Invalid（無効）の 3 つがある．read miss が起きた時は，マネージャが持っているディレクトリに自身を reader として記録し，（ホームベース DSM なので）オーナーからページ全体を取得する．write miss が起きた時は，ディレクトリに自身を writer として記録し，ここで元の reader や writer に対して新しい writer が増えたと通知する．この通知は invalidation メッセージに似ているが，通知を受け取った reader や writer は自身が保持しているディレクトリの複製にシェアラーの増加を記録するだけで，メモリブロック自体については「次の」SI_fence や SD_fence まで実際には何もしなくてよい．これも DRF の仮定により，シェアラー自身が発行したフェンスでのみデータ転送を行えばコヒーレンスを保てるからである．次のフェンスにたどり着いた際は，ディレクトリの複製を調べると共有状態が変化しているので，その際は適切に Self-invalidation や Self-downgrading を行う．

```
struct ult_id { void* p; };
struct allocated_ult {
    ult_id  id;
    void*   ptr;
};
allocated_ult allocate(size_t size);
void fork(allocated_ult th, void (*func)(void*));
void join(ult_id id);
void detach(ult_id id);
void yield();
void exit();
```

リスト 2 スケジューラの API

後述するスケジューラでは，実行中のコールスタックを含めて全てソフトウェア DSM によって管理しており，実行中のコールスタックに対して DSM がページ保護状態を変更するとデッドロックが生じる問題があることが実装過程で判明した．この問題を解決するため，DSM に `pin` と `unpin` という関数を用意し，さらに DSM のブロックに対しプロセス内のみで有効な “Pinned” という特殊な状態を導入した．この状態になっているブロックは，必ずアプリケーションから読み書き可能になっており，SI_fence や SD_fence が発行されても実際には Self-invalidation や Self-downgrading が起きない．

4. 分散スレッドスケジューラの実装

4.1 スケジューラの API

提案するスケジューラのインターフェイスは，一般的なマルチスレッドのインターフェイス（例えば Pthreads）や，共有メモリ上で動作するユーザレベルスレッド処理系のインターフェイスと同様である．提案手法ではヒープやコールスタックが DSM によって管理されているため，インターフェイスとしては共有メモリと同一であり，分散メモリ専用のプログラミングモデルを導入する必要が無い．但し，DSM が DRF を仮定しているため，データレースが存在するプログラムに対して正常動作を保証しないことはスケジューラと組み合わせた場合も同様である．

リスト 2 に，提案する処理系におけるスケジューラのインターフェイスを示す．基本的には一般的な共有メモリ用のユーザレベルスレッドと同様のインターフェイスであるが，性能向上のためにスレッドを `fork` する際の手順が異なる．通常のスレッド処理系では `fork` 時にスレッドに必要な資源（スレッドデスクリプタとコールスタック）をまとめて確保するが，提案するインターフェイスでは `allocate` 関数によって一旦スレッドの資源を確保してから `fork` 関数に渡すという形態になっている．このようなインターフェイスにすることで，C++11 の `std::thread`

のような「任意サイズの関数オブジェクト」を受け取れるインターフェイスに接続する際に，allocate 関数によって確保したコールスタックに対して *placement new* を行うことで，関数オブジェクトをヒープから確保しなくて済む．

4.2 分散スレッドスケジューラの実装

実装の流れとして，マルチコアプロセッサ上（ハードウェア共有メモリ）で動作するようユーザレベルスレッドライブラリを実装し，それを DSM 上で動かせるように修正するという手順を取った．具体的には，ユーザレベルスレッドの各モジュールを C++のテンプレートによって抽象化し，ハードウェア共有メモリ上の実装と DSM 上の実装の両方を実体化できるようにした．このような手法により，DSM から独立してスケジューラをデバッグ可能となり，統合テストの負担を最小化できる．

分散メモリ実装において問題となるのは，「スレッドデスクリプタをどう定義するか」である．DSM を最大限活かす発想として，「スレッドデスクリプタも DSM 上に配置する」という手法があるが，現在の DSM の実装においてはアトミック命令などが導入されていないためにこの手法は実装できない．そのため，デスクリプタ領域に関しては DSM の管理外の領域とし，リモートプロセス上のデスクリプタは RDMA か AM によって操作するという方式を採用した．

4.3 ページベース DSM 上のコールスタックによるデッドロック問題と回避策

スケジューラの実装過程で，コールスタックをページベース DSM 上に置くことによってデッドロックの問題が生じることが分かった．具体的には，次のような状況で再帰的ロックが発生した．

(1) 分散スレッドスケジューラ上のあるユーザレベルスレッドが，通信システムのロックを取得する．
(2) 通信システム内でスタックを伸長させる．
(3) 伸長したスタック領域が DSM 上でキャッシュされておらず，Segmentation fault ハンドラに突入する．
(4) リモートノードからのデータ読み出しを開始しようとする．
(5) 通信システムの関数が呼び出され，再びロックを取得する．

これと同様の問題は，実際にマルチスレッド化された DSM とスケジューラを組み合わせると頻発したため，前述の DSM に pin/unpin という API を導入して対処することとした．分散スレッドスケジューラが別のスレッドにスイッチする際，まずスイッチ先のコールスタックを Pinned にしてからコンテキストスイッチを実行する．そして，スイッチ先に移った後でスイッチ元のコールスタックの Pinned を解除する．このような手順により，現在実行されているスレッドのコールスタックが必ず Pinned になっていることを保証でき，デッドロックを回避できる．

Pinned になっているメモリブロックへの変更は他ノードと一切共有されないが，これは DRF を仮定しているので問題が発生しない．あるスレッドが行った変更を読めるのは，そのスレッドが同期命令を発行したときだけであり，ロックやアトミック命令などを無視すればそれはスレッドが中断/終了した時しかないからである．そして，スレッドが中断/終了した場合はスイッチが起きて Pinned は解除されるので，その場合でも問題は発生しない．

4.4 分散メモリ型計算機上のワークスティーリングスケジューラとメモリフェンス

提案するシステムにおいて，スレッド間の依存関係はコンシステンシが規定する synchronization order の一部に含まれる．つまり，あるスレッドが行った変更は，そのスレッドから fork されたスレッドや，そのスレッドを join したスレッドといったような後続のスレッドに対して可視になることが保証される．言い換えれば，分散スレッドスケジューラのスレッド API を呼び出すと，スレッド間の依存関係に基づくコンシステンシ (DAG Consistency [17] [18]) が保たれるよう DSM に働きかける．以下では，DAG Consistency を保つため，ワークスティーリングスケジューラの各機能についてどのように SI_fence/SD_fence 命令を挿入するかを述べる．

まず，スレッドを他ノードから盗む (steal) 状況を考える．ここでは，AM によってワークスティーリングを実装することを想定する．この場合，そのスレッドが元々実行されていたノード (victim) で SD_fence を実行し，それが終わり次第これから実行するノード (thief) が SI_fence を実行する．このような手順により，thief 側でも最新のスレッドのコールスタックが読めるようになっているので，盗んだスレッドを再開できる．

join の呼び出し時には，join されるスレッドがその時点で終了しているかどうかで扱いが異なる．まず終了していない場合，現在のスレッドはブロックされるため，現在のコンテキストを保存してスレッドデスクリプタに “joiner” として記録する．次に，join されるスレッドが既に終了していて，かつ終了時に自プロセスで実行されていた場合は，DSM のフェンスに関しては何も挿入する必要がなく，スレッドの破棄などを行うのみでよい．問題となるのは，他プロセスで実行されてそこで終了したスレッドを join する場合である．この場合，そこで行われた変更を含めた全ての（happens-before で順序付けられた）変更を，join するプロセス側が見えるようにする必要がある．現在の実装では，join される側のスレッドが実行されていたノードに対し，AM を用いて SD_fence を発行させ，その後 join する側のプロセスで SI_fence を発行するという方式を取った．こ

表 1　評価環境 (ReedBush-U [52])

CPU	Intel® Xeon® E5-2695 v4
	2.1 GHz (Turbo boost 時 最大 3.3 GHz)
	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
ドライバ	Mellanox® OFED 3.4-2.1.4
OS	Red Hat® Enterprise Linux® 7.2
コンパイラ	GCC 4.8.5 (with the option "-O3")
MPI	Intel® MPI Library 2017 Update 2

の際に発行する SI_fence においては，現在のスレッドを実行しているコールスタックが Pinned になっていて，しかもそのコールスタックへの変更を取り込む必要があるため，一旦 DSM 管理外のコールスタックにスイッチして Pinned を解除し，そこで SI_fence を実行した後再び Pinned に戻すという実装方式を取った．

exit 時には "joiner" が記録されていればそのスレッドにスイッチするので，joiner がリモートプロセスとして実行されていた場合にフェンスを挿入する必要がある．これも前述の join と同様に解決しており，元々実行されていたノードに対して SD_fence を実行するよう AM を送信する．exit の場合は自スレッドが破棄されるため，自スレッドのコールスタックへの変更を取り込む必要はなく，Pinned を解除せずにそのまま SI_fence を実行してよい．

以上のような実装方式により，SI_fence と SD_fence のみを用いて，スレッド間の依存関係を適切に DSM に対して通知し，コヒーレンスとして反映することができる．

現在の実装方式では，複数の状況で AM によって SD_fence を要求する必要がある．この方式は RDMA 化が困難であるだけでなく，スイッチ時のレイテンシ増大につながっている．3.1 節で述べた非同期フェンス命令を用いれば，スレッドが中断/終了した際に毎回非同期 SD_fence を裏で実行しておくことができる．そして，そのフェンスが終了次第，コールバック関数を用いてそのスレッドを他ノードで実行可能であると記録しておけば，後になって AM によって SD_fence を要求する必要がなくなる．この方式は，自らのキャッシュは自ら downgrade するという，Self-downgrading の考え方にも合致している．

5. 評価手法

評価環境を表 1 に示す．今回のベンチマークは主にノード間通信を前提として DSM やスケジューラの性能を測ることが目的のため，1 ノード＝1 プロセス内に 1 ワーカースレッドのみを配置しており，従ってノード内のマルチコアはほとんど活用されていない．但し，通信オフローディング用のスレッドはワーカーと別に各プロセスに存在し，AM がデッドロックすることを避ける目的 [53] で異なる

```
char* p = /* allocate from DSM */;

double t0 = cur_time();
for (int k = 0; k < size; ++k)
    s += p[k]; // read
double t1 = cur_time();
for (int k = 0; k < size; ++k)
    p[k] = x; // write
double t2 = cur_time();
SD_fence();
double t3 = cur_time();
SI_fence();
double t4 = cur_time();
```

リスト 3　DSM のマイクロベンチマーク

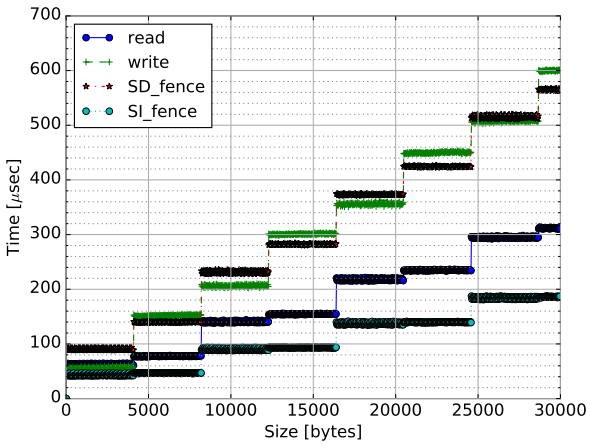

図 2　DSM のマイクロベンチマーク結果

AM ごとに別々のスレッドを用意している．しかし，いずれもユーザプログラムの計算の並列性を向上させるものではない．

DSM のマイクロベンチマークとしては，リスト 3 のようにメモリの読み書きやフェンス命令を実行した際の時間を計測した．2 ノードで計測し，各ノードに 1 プロセス用意した上で，片側を計測用のプロセスとして使用する．計測していない側のプロセスは，メモリ領域を書き込み可能な状態で共有しているため，計測側のプロセスはフェンス発行時に Self-downgrading と Self-invalidation を省略することはできない．

スケジューラの性能計測には，タスクを大量に生成するフィボナッチ数計算のベンチマークを用いた．このベンチマークの計測中，バグで異常終了する場合があることが確認されているが，計測結果から異常終了した結果は除いている．バグの原因は現在調査中である．

6. 評価結果

図 2 に，DSM のマイクロベンチマーク結果を示す．最も時間がかかっているのが SD_fence であり，1 ページサイズに収まる小さい書き込みであっても 100μsec 程度 (約 20 万

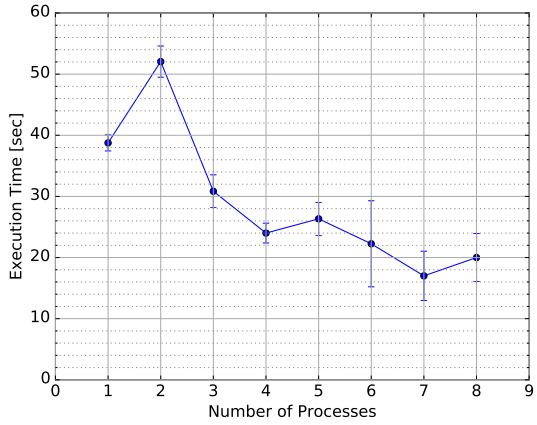

図 3 fib(30) の実行時間

サイクル) の時間がかかっていることが分かる．InfiniBand EDR でのノード間往復レイテンシは実測で約 3μsec 程度である [54] ため，仮にフェンス命令 1 回に対して数回の通信が必要であるとしても，現状の実装には改善の余地が大きいことが分かる．アクセスサイズを増加させた時，ページ境界を超えた時に各操作の実行時間も増大している．また，奇数番目のページ境界をまたいだ場合については，偶数番目の場合よりも実行時間の増大が小さい．その原因は，各ブロックを管理しているディレクトリの割り当てがラウンドロビンで行われているため，同じプロセスにあるディレクトリに対してだけ高速にアクセス可能だからである．

DSM の主な処理はディレクトリ操作とデータ転送であり，このうちディレクトリ操作には特に AM の性能を向上させることが重要である．Argo DSM のようにディレクトリ操作を含めて全て RDMA 化することも不可能ではないが，RDMA 化は利用可能なデータ構造を限定し，メッセージの往復回数を増大させるので，一概に性能向上に寄与するともいえない．データ転送は RDMA が活用できるが，diff を利用する場合は細粒度の通信を頻発させるので，現在は書き込みにも AM を利用している．

図 3 に，開発した処理系での fib(30) の実行時間を示す．1 プロセスでの実行時間は平均 38.8 秒であった．参考として，共有メモリ上で動作するユーザレベルスレッド処理系である MassiveThreads を同環境で実行すると，fib(30) を 1 ワーカーで 0.147 秒で実行するため，開発した処理系はその 264 倍遅いということになる．分散メモリ向けの処理系としては MassiveThreads/DM も MassiveThreads に対して数割のオーバーヘッドしか持たないため，分散メモリを加味しても現状の実装は逐次性能に問題があるといえる．並列性能も現時点ではスケールしておらず，特に 2 プロセス使用時の性能低下が著しい．この原因は，1 プロセス内で実行している場合と比べてリモートへの RDMA や AM が増大し，かつ 2 プロセスしかないのでそれらの処理が集中していることであると推察される．

プロセス数が増えるほど steal 数も増大するので，その分メモリフェンスの発行数も増大することになり，前述の DSM のフェンス性能が重要となる．また，現状のスケジューラはコンテキストスイッチの度に pin や unpin の呼び出しを必要とし，DSM 内部ではシェアラー用のテーブルをロックしている．このため，DSM の実装は通信が実際には発生していなくともオーバーヘッド発生の要因となりうる．スケジューラ自身の実装上の問題としては，スレッドデスクリプタの管理に現状では RDMA アトミックを使用しているため，ローカルであっても RDMA を使用しなければならないという問題 [55] がある．この他，ワークスティーリングに AM を使用しているので，AM の性能は DSM 同様に重要である．

7. 結論

本稿では，DSM とそれに基づいた分散スレッドスケジューラについて，既存研究を踏まえた現在のハードウェアにおける設計と実装について示した．評価結果においては，現時点の実装は既存のスケジューラと比べて極めて低速で，実用的といえる段階には至っていないことを述べたが，今後改善していく方針についても述べた．現時点の実装は試験的なものであり，未だバグを解消しきれていないため，まずは安定性を高めていく予定である．

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．

参考文献

[1] 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).

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

[3] El-Ghazawi, T. and Cantonnet, F.: UPC Performance and Potential: A NPB Experimental Study, *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, p. 26 (online), DOI: 10.1109/SC.2002.10034 (2002).

[4] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H. and Aprà, E.: Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 203–231 (online), DOI: 10.1177/1094342006064503 (2006).

[5] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S. and Oskin, M.: Grappa : A Latency-Tolerant Runtime for Large-Scale Irregular Applications, Technical report (2014).

[6] Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C. and Smith, L.: Introducing OpenSHMEM, *PGAS '10: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, No. c, New York, New York, USA, ACM Press,

pp. 1–3 (online), DOI: 10.1145/2020373.2020375 (2010).
[7] Endo, W. and Taura, K.: 再配置可能な大域アドレス空間システムの設計と RDMA を用いた実装，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2015-HPC-1, No. 5, pp. 1–8 (2015).
[8] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J.: The directory-based cache coherence protocol for the DASH multiprocessor, *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, IEEE Comput. Soc. Press, pp. 148–159 (online), DOI: 10.1109/ISCA.1990.134520 (1990).
[9] 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).
[10] 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).
[11] 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).
[12] 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).
[13] 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).
[14] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors, *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26 (online), DOI: 10.1109/ISCA.1990.134503 (1990).
[15] Bershad, B. N., Zekauskas, M. J. and Sawdon, W. A.: The Midway distributed shared memory system, Technical report (1993).
[16] 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).
[17] 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).
[18] 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).
[19] 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).
[20] 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).
[21] 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).
[22] 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).
[23] Ros, A. and Kaxiras, S.: Complexity-Effective Multicore Coherence, *PACT '12: Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, New York, New York, USA, ACM Press, p. 241 (online), DOI: 10.1145/2370816.2370853 (2012).
[24] 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).
[25] Kaxiras, S., Klaftenegger, D., Norgren, M., Ros, A. and Sagonas, K.: Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory, *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).
[26] Hara, K.: 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).
[27] Hara, K. and Taura, K.: Parallel Computational Reconfiguration Based on a PGAS Model, *Journal of Information Processing*, Vol. 20, No. 1 (2012).
[28] 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).
[29] Kee, Y.-S., Kim, J.-S. and Ha, S.: Memory management for multi-threaded software DSM systems, *Parallel Computing*, Vol. 30, No. 1, pp. 121–138 (online), DOI: 10.1016/j.parco.2003.09.001 (2004).
[30] Martin, M. M. K., Hill, M. D. and Sorin, D. J.: Why on-chip cache coherence is here to stay, *Communications of the ACM*, Vol. 55, No. 7, p. 78 (online), DOI: 10.1145/2209249.2209269 (2012).
[31] Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, *Journal of the ACM*, Vol. 46, No. 5, pp. 720–748 (online), DOI: 10.1145/324133.324234 (1999).
[32] Nakashima, J.: 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 (2011).
[33] Nakashima, J. and Taura, K.: 高効率な I/O と軽量性を両立させるマルチスレッド処理系，情報処理学会論文誌プログラミング（PRO），Vol. 4, No. 1, pp. 13–26 (2011).
[34] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond*, Vol. 8665, pp. 222–238 (on-

line), DOI: 10.1007/978-3-662-44471-9 (2014).
[35] Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *IPDPS '08: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, (online), DOI: 10.1109/IPDPS.2008.4536359 (2008).
[36] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Castello, A., Genet, D., Herault, T., Jindal, P., Kale, L., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y. and Beckman, P. H.: Argobots : A Lightweight , Low-Level Threading and Tasking Framework, Technical report (2016).
[37] Intel: Threading Building Blocks, `https://www.threadingbuildingblocks.org/`.
[38] Barcelona Supercomputing Center: Nanos++, `https://pm.bsc.es/nanox`.
[39] Kowalke, O.: Boost.Fiber, `http://www.boost.org/doc/libs/release/libs/fiber`.
[40] Kowalke, O. and Goodspeed, N.: N3985: A proposal to add coroutines to the C++ standard library, Technical report (2014).
[41] Nishanov, G. and Radigan, J.: N4402: Resumable Functions (revision 4), Technical report (2015).
[42] Nishanov, G., Maurer, J., Smith, R. and Daveed Vandevoorde: P0057R7: Wording for Coroutines, Technical report (2015).
[43] Antoniu, G., Bougé, L. and Namyst, R.: An efficient and transparent thread migration scheme in the PM2 runtime system, *RTSPP '99: Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, Vol. 1586, pp. 497–510 (online), DOI: 10.1007/BFb0097934 (1999).
[44] Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Lukasz Wesolowski and Kale, L.: Parallel Programming with Migratable Objects: Charm++ in Practice, *SC '14: Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 647–658 (online), DOI: 10.1109/SC.2014.58 (2014).
[45] Akiyama, S. and Taura, K.: Scalable Work Stealing of Native Threads on an x86-64 Infiniband Cluster, *Journal of Information Processing*, Vol. 24, No. 3, pp. 583–596 (online), DOI: 10.2197/ipsjjip.24.583 (2016).
[46] Akiyama, S. and Taura, K.: Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing, *HPDC '15: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ACM Press, pp. 15–26 (online), DOI: 10.1145/2749246.2749272 (2015).
[47] Taylor, I. L.: Split Stacks in GCC, `https://gcc.gnu.org/wiki/SplitStacks` (2011).
[48] Peng, L., Wong, W., Feng, M. and Yuen, C.: SilkRoad: a multithreaded runtime system with software distributed shared memory for SMP clusters, *CLUSTER '00: Proceedings of IEEE International Conference on Cluster Computing*, pp. 243–249 (online), DOI: 10.1109/CLUSTR.2000.889067 (2000).
[49] Peng, L., Wong, W. F. and Yuen, C. K.: SilkRoad II: Mixed paradigm cluster computing with RC_dag consistency, *Parallel Computing*, Vol. 29, pp. 1091–1115 (online), DOI: 10.1016/S0167-8191(03)00093-0 (2003).
[50] Endo, W. and Taura, K.: PGAS 向け低水準通信レイヤーのマルチスレッド実装，研究報告ハイパフォーマンスコンピューティング（HPC），Vol. 2016, No. 28, pp. 1–11 (2016).
[51] 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).
[52] 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.
[53] Agarwal, S., Barik, R., Bonachea, D., Sarkar, V., Shyamasundar, R. K. and Yelick, K.: Deadlock-free scheduling of X10 computations with bounded resources, *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, p. 229 (online), DOI: 10.1145/1248377.1248416 (2007).
[54] Papadopoulou, N., Oden, L. and Balaji, P.: A Performance Study of UCX over InfiniBand, *CCGrid '17: Proceedings of 17th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing* (2017).
[55] Akimoto, H., Miura, K., Okamoto, T., Yuichiro, A. and Shinji, S.: InfiniBand Atomic Operation の性能評価，HPC 研究発表会，Vol. 2012, No. 8, pp. 1–6 (2012).