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

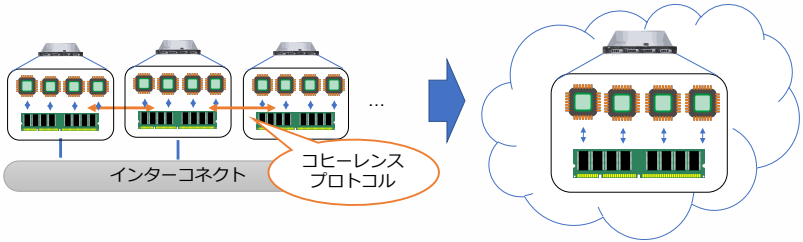
遠藤 亘, 田浦 健次朗

東京大学 大学院 情報理工学系研究科
電子情報学専攻

2017 年 7 月 28 日 @ SWoPP 2017

研究概要 (1)

- 分散共有メモリ **(Distributed Shared Memory, DSM)** について再考
  - 1990 年代に盛んに研究されたが，PGAS に取って代わられてしまったモデル
- 分散ワークスティーリングと組み合わせる
- 別で開発した低水準通信システムと組み合わせる
  - 昨年の SWoPP で発表



研究概要 (2)

- なぜ DSM について研究すべきか
  1. DSM がスケーラブルでないとされる本当の理由は？
  2. メニーコア化の限界とも関係

研究概要 (2)

- なぜ DSM について研究すべきか
  1. DSM がスケーラブルでないとされる本当の理由は？
  2. メニーコア化の限界とも関係
- DSM とスケジューラを約半年間かけて実装
  - 並行性バグとの戦い（非決定的バグ，デッドロック）
  - 最近になって正常動作するようになった
- 現状の評価結果は良好でない
  - チューニングはこれから
  - 本発表では，必要だった構成要素について主に解説

提案システムの基本方針

- 分散メモリ型計算機上で「共有メモリ」+「スレッド」
  - 並列プログラムにとって必要最小限
  - その他の大半のモデルは，全てこの 2 つで表現できる
- 「分散メモリ用の新たな **API**」を提案しない
  - 特定の実装方式の都合でプログラミング制約を導入しない
- 分散メモリ型計算機上で実装するための 2 つの仕組み:
  1. ソフトウェア分散共有メモリ
  2. 分散ユーザレベルスレッド
     - 分散ワークスティーリングによる動的負荷分散

DSM のスケーラビリティ問題

- PGAS と共有メモリの違い
  - **PGAS** = グローバルアドレス空間 + ローカルの非コヒーレント領域
  - 共有メモリ = グローバルアドレス空間 + コヒーレントキャッシュ
- コヒーレントキャッシュが問題？
  - PGAS のプログラミング生産性には課題が多い (e.g. [Zhang et al. '11])

DSM のスケーラビリティ問題

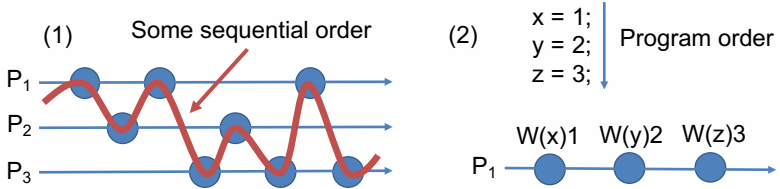
- PGAS と共有メモリの違い
  - **PGAS** = グローバルアドレス空間 + ローカルの非コヒーレント領域
  - 共有メモリ = グローバルアドレス空間 + コヒーレントキャッシュ
- コヒーレントキャッシュが問題？
  - PGAS のプログラミング生産性には課題が多い (e.g. [Zhang et al. '11])
- 「コヒーレンスがスケーラブルでない」と言われる理由:
  - strict すぎるコンシステンシモデル
  - 利用可能なアドレス空間の大きさ
  - メモリ保護機構によるオーバーヘッド
  - キャッシュ複製数の制限
  - 無効化メッセージ送信先の増大
  - データやディレクトリへのアクセスが一箇所に集中

先行研究: ArgoDSM [Kaxiras et al. '15]

- DSM の基本的方式を踏襲：
  - 緩和型コンシステンシモデル (SC-for-DRF)
  - ページベース (vs. コンパイラベース）
  - ディレクトリベース (vs. スヌーピー）
  - Multiple-Reader Multiple-Writer 型（vs. Single-Writer 型）
  - Eager 型（vs. Lazy 型）
  - ホームベース （vs. ホームレス）
- 従来の DSM に見られなかった手法：
  - **Self-invalidation/downgrading** に基づくフェンス命令
  - **P/S3** 分類と分散ディレクトリ
- 提案手法もこれらの方式を採用
  - さらに追加機能をいくつか実装

Sequential Consistency

- コンシステンシモデル：
  - ある時点で各コアがメモリから読む値を規定
- **Sequential Consistency (略称 SC)**
  - 実用上最も厳しいコンシステンシモデル
  - メモリアクセスの順序が以下と一致:
    1. 全アクセスに渡ったある全順序 (global order)：リオーダリングを許さない
    2. プログラム順 (program order)

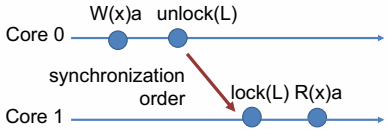


緩和型コンシステンシモデル

- SC よりもコンシステンシを緩和する目的：
  - リオーダリング許可による性能向上
- DSM における研究例は数多い：
  - Release Consistency [Keleher et al. ’92]
  - Entry Consistency [Bershad et al. ’93]
  - Scope Consistency [Iftode et al. ’98]
- メモリアクセスを 2 つに分類：
  - 非同期アクセス: load/store 命令
  - 同期アクセス: mutex の lock/unlock，スレッド生成/破棄，アトミック命令

happens-before とデータレース

- **happens-before** 半順序 = 同期順 (synchronization order) ∪ プログラム順
  - 同期順の例. mutex の unlock (release 側) → lock (acquire 側)
  - 同期命令がなければ，一貫性は全く保証されない



- データレース **(data races)**
  1. 同一アドレスへの複数のメモリアクセスが,
  2. かつ一つ以上が書き込みで,
  3. happens-before で順序付けられない
- データレースを含むプログラム＝実行結果が不定 (nondeterministic)

SC-for-DRF

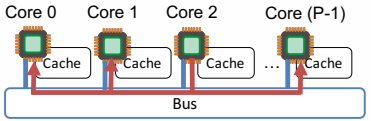
- **Data-Race-Free (DRF)** [Adve et al. '90]
  - 「データレースが存在しない」とプログラマが保証
  - アトミック命令の扱いによっていくつか種類がある (DRF0, DRF1)
- **SC-for-DRF** : 緩和型コンシステンシの一種
  - DRF プログラム → システムが Sequential Consistency を保証
  - 非 DRF プログラム → *"undefined behavior"*
- 本研究でも SC-for-DRF を基本とする
  - C++11 など，最近の言語のメモリモデルで採用
  - 現状の言語のセマンティクスを変える必要がない
  - Release Consistency など重要なコンシステンシモデルを包含

DSM の基本的手法 (1)

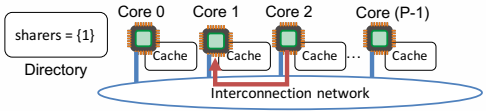
- ページベース
  - キャッシュ存在判定をメモリ保護機構（mprotect + SEGV ハンドラ）で行う
  - 問題点: システムコールのオーバーヘッド，ブロックサイズを 4KiB 未満にできない
  - c.f. コンパイラベース:
    - コンパイラで load/store 命令を特殊な処理に置換
    - 問題点: 開発コストが大きい，キャッシュヒット時の性能低下

DSM の基本的手法 (1)

- ページベース
  - キャッシュ存在判定をメモリ保護機構（mprotect + SEGV ハンドラ）で行う
  - 問題点: システムコールのオーバーヘッド，ブロックサイズを 4KiB 未満にできない
  - c.f. コンパイラベース:
    - コンパイラで load/store 命令を特殊な処理に置換
    - 問題点: 開発コストが大きい，キャッシュヒット時の性能低下
- ディレクトリベース
  - “ディレクトリ” にシェアラーを記録しておき，それを元にコヒーレンスを保つ
  - c.f. スヌーピー: 全ノードに毎回問い合わせる



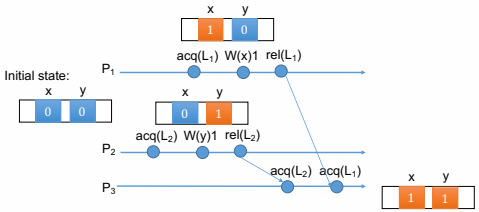
スヌーピー



ディレクトリベース

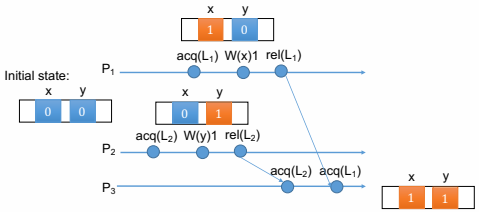
DSM の基本的手法 (2)

- **Multiple-Reader Multiple-Writer 型**
  - ページの差分（“diff”）を用いて複数のノードからの書き込みを併合
  - c.f. Single-Writer 型 : 書き込めるノードを 1 つに制限



DSM の基本的手法 (2)

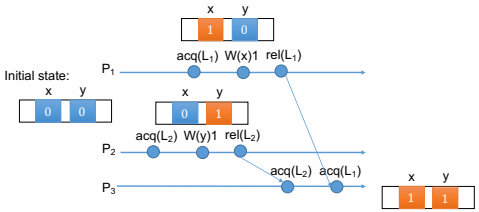
- **Multiple-Reader Multiple-Writer** 型
  - ページの差分（“diff”）を用いて複数のノードからの書き込みを併合
  - c.f. Single-Writer 型 : 書き込めるノードを 1 つに制限



- **Eager** 型
  - diff の併合を release 時で行う
  - バンド幅は増えるが，レイテンシが減らせる
  - c.f. Lazy 型 : diff の併合を acquire 時に行う

DSM の基本的手法 (2)

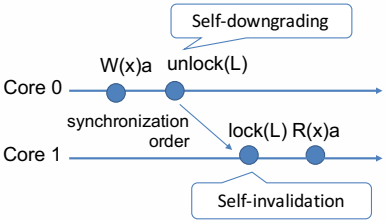
- **Multiple-Reader Multiple-Writer** 型
  - ページの差分（“diff”）を用いて複数のノードからの書き込みを併合
  - c.f. Single-Writer 型 : 書き込めるノードを 1 つに制限



- **Eager** 型
  - diff の併合を release 時で行う
  - バンド幅は増えるが，レイテンシが減らせる
  - c.f. Lazy 型 : diff の併合を acquire 時に行う
- ホームベース
  - diff は「ホーム」上にまとめる
  - c.f. ホームレス : diff を各 writer から集める

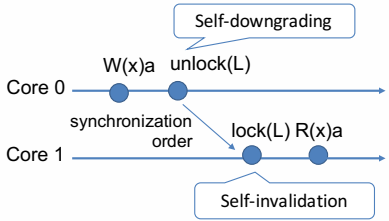
同期アクセスの実装方式

- 同期アクセスの naive な実装手法
  - **release 側** 自ノードが変更した全キャッシュブロックをホームに書き戻す
    （自らキャッシュを書き戻すので **Self-downgrading** と呼ぶ）
  - **acquire 側** 自ノードの全キャッシュブロックを無効化する
    （自らキャッシュを無効化するので **Self-invalidation** と呼ぶ）



同期アクセスの実装方式

- 同期アクセスの naive な実装手法
  - **release 側** 自ノードが変更した全キャッシュブロックをホームに書き戻す
    （自らキャッシュを書き戻すので **Self-downgrading** と呼ぶ）
  - **acquire 側** 自ノードの全キャッシュブロックを無効化する
    （自らキャッシュを無効化するので **Self-invalidation** と呼ぶ）



- naive な手法の問題点：**acquire** 後のキャッシュミスの多発
  - 無効化しなくてよいはずのキャッシュブロックまで無効化してしまう
- 不要な Self-invalidation をどう回避するか？

Self-invalidation が不要なメモリブロックを調べる手法 (1)

- Vector timestamp + Write notices:
  - 古典的な DSM で使われていた手法 (e.g. TreadMarks [Keleher et al. '92])
  - 各ページごとに全プロセッサの変更時点を記録
  - 利点
    - 不要な Self-invalidation を正確に回避できる
  - 欠点:
    - Write notice は最大でメモリブロック数に比例したサイズになる
    - プロトコルとして複雑（RDMA 化もしづらい）

Self-invalidation が不要なメモリブロックを調べる手法 (2)

- **P/S3** 分類：
  - ArgoDSM で用いられている
  - その時点で「他ノードが書き込んでいる」場合だけ Self-invalidation する
    - どのノードも書き込んでいないキャッシュブロック (read-only) は Self-invalidation 不要
    - 自ノードだけ書き込んでいる場合も Self-invalidation が不要
  - なぜこのような手法が有効か？
    - ある種のアプリケーションでは **“Private”** ブロックが **8** 割程度を占めるとも [Esteve et al. ’16]

分散ディレクトリ

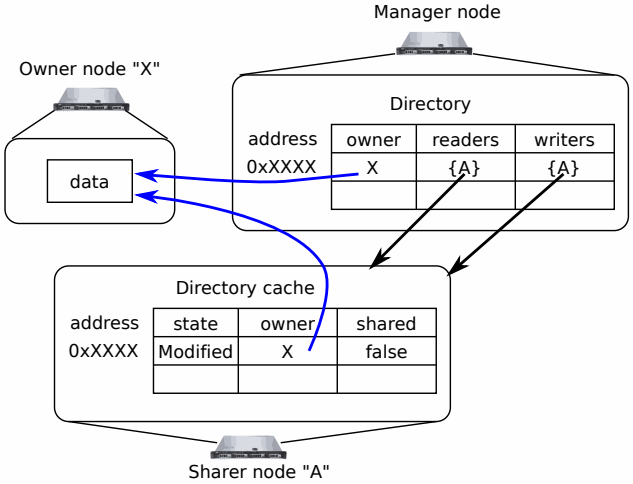
- どうやって P/S3 分類を実装するか？
  - ディレクトリに毎回問い合わせれば，writer の有無は確認できる
  - Self-invalidation（による通信）削減のために通信をしては本末転倒

分散ディレクトリ

- どうやって P/S3 分類を実装するか？
  - ディレクトリに毎回問い合わせれば，writer の有無は確認できる
  - Self-invalidation（による通信）削減のために通信をしては本末転倒
- 分散ディレクトリ（ディレクトリキャッシュ）
  - ディレクトリ情報の一部をシェアラーにも記録
  - P/S3 分類の場合は，「他ノードが書き込んでいるかどうか」の 1 bit (Private or Shared)
- 分散ディレクトリのコヒーレンス
  - 新しく Writer が出現した時が問題
  - そのメモリブロックが “Private” だと記録している
    シェアラーのディレクトリキャッシュを Shared に書き換える
    (**Deferred Self-invalidation**)

提案する DSM システムのディレクトリ構造

- 基本的には ArgoDSM と同じディレクトリ構造を使用
  - ディレクトリ操作は Active Message を用いて実装
- ホームを動的に変更できるように設計
  - 初回アクセスしたノードに配置
  - 動的再配置（一昨年の SWoPP で発表）は未実装



提案する DSM システムの API

- メモリアクセスの API : ArgoDSM と同様
  - load/store，SI/SD_fence
- メモリ領域確保（mmap のようなもの）
  - セグメントという単位で確保
  - セグメント内はアドレス連続が保証
  - 各セグメントに異なるブロックサイズを使用可能
  - 共有メモリなので，通常のメモリアロケータが使える (e.g. dlmalloc)
- 後述する分散スケジューラで使うため，以下の関数を追加
  - pin/unpin，非同期フェンス

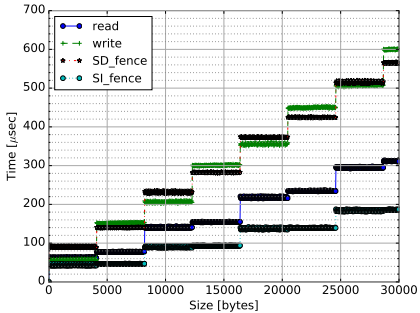
```
// Self-invalidationフェンス
void SI_fence();
// Self-downgradingフェンス
void SD_fence();

// セグメントの確保
void* make_segment(
  size_t seg_size,
  size_t block_size);
```

初期評価 (1) : DSM のみのマイクロベンチマーク

- 東大情報基盤センターのスパコン ReedBush-U 上で実験
- 各操作を繰り返すマイクロベンチマーク
  - 2 ノードで実験，片方のノードがメモリアクセスを繰り返す
  - 縦軸: 実行時間，横軸: アクセスサイズ
- 単一のメモリブロックに対して各操作が数十 μsec かかる
  - c.f. RDMA レイテンシは 2〜3μsec 程度
  - 一番遅いのは SD_fence
    - diff を生成 → ホーム上で展開

```
for (int k = 0; k < size; ++k)
    s += p[k]; // read
for (int k = 0; k < size; ++k)
    p[k] = x; // write
SD_fence();
SI_fence();
```

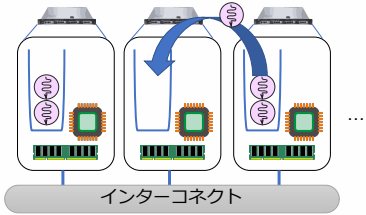


現状の DSM 実装の課題

- 複数メモリブロックにまたがる場合の並行処理が現状できていない
  - 本来，別々のメモリブロックは並行に処理できる
- Active Message の性能が遅い
  - MPI で試験的に作ったものでチューニングできていない
    - 測定したところ，往復のレイテンシが約 50 μsec（97000 サイクル）だった
  - RDMA 化は選択肢の一つだが，一操作あたりの往復回数は増える
- メモリアクセスの予測的発行が必要
  - load/store : プリフェッチ
  - SI_fence : *Dynamic* self-invalidation [Lebeck et al. '95]
  - SD_fence : ライトバックスレッド

分散スレッドスケジューラ

- **DSM** を前提としたユーザレベルスレッド処理系
  - 分散ワークスティーリングによるスケジューリング
  - コールスタックを DSM に配置
  - スケジューリングに応じて，自動的に DSM 空間全体のコヒーレンスを保つ



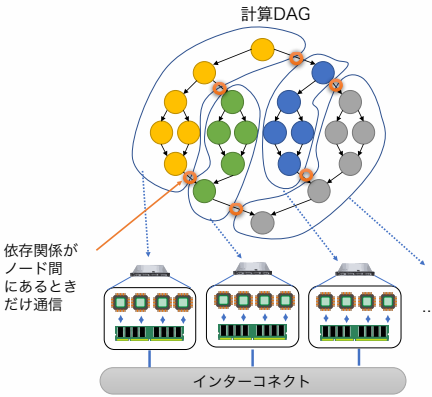
- まずはランダムワークスティーリングを実装
  - steal 時に全ノードからランダムに victim のワーカーを選ぶ
  - 将来的には DSM の情報を活用したスケジューリングも検討（いわゆる locality-aware と呼ばれる分野）

ユーザレベルスレッドの実装

- ユーザレベルスレッドとは？
  - コンテキストスイッチをユーザレベルで実装したスレッド処理系
  - 共有メモリでの実装は多数 (e.g. MassiveThreads [中島ら '14])
- 各（ユーザレベル）スレッドが所有するデータ：
  1. コールスタック: DSM で管理
     - システム全体の簡潔性のため
     - コールスタックを DSM が高速に管理できれば,
       それ以外の領域も同様に高速にできて全体の実装量が減らせる？
  2. スレッドデスクリプタ: DSM 管理外
     - 各スレッドの状態を管理
     - 現状は Active Message で操作

DSM とスケジューラを組み合わせる先行研究

- **DAG Consistency** [Blumofe et al. ’96]
  - スレッド依存関係（計算 DAG）に基づいたコンシステンシモデル
  - SC-for-DRF に包含されている
    - 計算 DAG で happens-before 半順序が定まる
- 先行研究: Distributed Cilk [Blumofe et al. ’96], SilkRoad [Peng et al. ’03]
  - コールスタックを DSM に管理させる手法
  - 当時のハードウェア事情と比べて，実装方針は違う
    - 当時は Lazy 型重視，メモリ保護機構が未整備，etc.



ページベース **DSM** とコールスタック

- 共有メモリとスケジューラを両方ソフトウェアで構築すると，一体何が起きるのか？
  - 実装を始めた当初はよく分かっていなかった
- 実行中のコールスタックに対して SEGV ハンドラで存在確認を行うと，
  以下のようなデッドロックになる例が見つかった：
  1. あるミューテックスをロックする（例えば通信システム内のロック）
  2. スタックを伸長する
  3. SEGV ハンドラに突入
  4. 同じミューテックスを再び取る ← 二重ロック

SEGV ハンドラによるデッドロックの解決策

1. 同じミューテックスをロックしない
   or 全てのミューテックスを再帰的ミューテックスにする
   - 標準ライブラリや通信システムなど，ミューテックスは各所にある
   - 再帰的ミューテックスは，可能であれば使いたくない
2. コールスタックでそもそも **SEGV** が起きないようにする
   - 「実行中のコールスタック」が常に DSM 上でキャッシュヒットするようにする
   - こちらの方が現実的だったので採用

コールスタックと pin/unpin

- DSM に pin/unpin という関数を導入
  - コールスタックのメモリブロックを特殊な状態に遷移（名前が適切だったかは不明）
  - コンテキストスイッチ時にスケジューラが自動的に呼ぶ
- コンテキストスイッチ時の動作
  1. 次に実行するスレッドのコールスタックを pin
  2. スタックポインタを次に実行するスレッド上に変更
  3. 元々実行していたスレッドのコールスタックを unpin

```
void pin(void*, size_t);
void unpin(void*, size_t);
```

スレッド移動とメモリフェンス

- 自ノードで実行されていたスレッドを再開する場合，フェンスは要らない
  - スレッド移動が起きない限り，通信は要らない

スレッド移動とメモリフェンス

- 自ノードで実行されていたスレッドを再開する場合，フェンスは要らない
  - スレッド移動が起きない限り，通信は要らない
- 「他ノード上で実行されていたスレッドを再開」する状況でフェンスが必要
  - 具体的には，steal / join / exit の 3 つ
  - コールスタックを含めた全ての変更を取り込む必要がある

スレッド移動とメモリフェンス

- 自ノードで実行されていたスレッドを再開する場合，フェンスは要らない
  - スレッド移動が起きない限り，通信は要らない
- 「他ノード上で実行されていたスレッドを再開」する状況でフェンスが必要
  - 具体的には，steal / join / exit の 3 つ
  - コールスタックを含めた全ての変更を取り込む必要がある
- 最初に実装したスレッド移動の実装方式
  1. 元々実行していたノードに Active Message を送信
  2. SD_fence を実行させて変更が適用されるまで待つ
  3. 再開側のノードで SI_fence を実行して変更を取り込む

非同期フェンスの活用

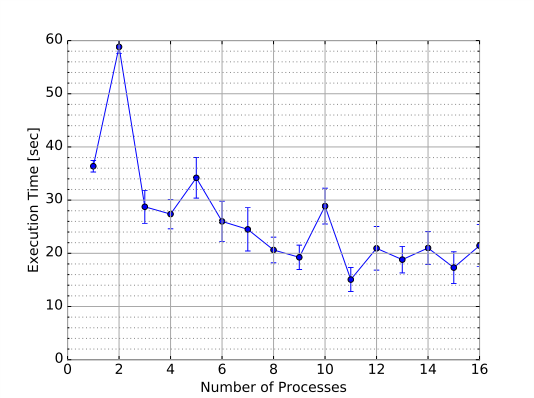
- Active Message による SD_fence 要求の問題点
  1. スレッド再開のレイテンシ増大
     - “後でから”downgrading を行うので，Lazy プロトコルの挙動に近い
  2. RDMA 化できない
- 改善案: 非同期フェンスの導入
  - スレッド中断/終了時に SD_fence を予め発行
  - 他ノード上で再開時に SD_fence を要求せずに済む
- アプリケーションスレッド上で実行すると，アプリケーションの実行を妨げる
  - 非同期に別スレッド（＝ライトバックスレッド）でやる

非同期フェンスの実装

- 非同期フェンスを用いたスレッド移動
  1. ライトバックスレッドに SD_fence フェンスの実行を指示
  2. SD_fence フェンスが終わり次第，
     「スレッドを他ノードで再開してよい」とスレッドデスクリプタに書き込む
- 実装を行ったものの，性能が逆に低下
  - ライトバックスレッドがアプリケーション側のスレッドと衝突している？
  - 非同期フェンスがなければ書き込み可能だったはずのページが，
    全部書き込み不能になる

初期評価 (2) : スケジューラと組み合わせた性能

- スケジューラの性能だけを測定するフィボナッチ数計算のベンチマーク
  - fib(30) の実行結果
  - この実験では 1 ノード 1 ワーカーのみ
    - マルチスレッド化は既に行っているが評価はこれから
  - 予稿時点であったバグは解決済み
- 現状の性能結果は良好でない
  - スケールしていない
  - 逐次性能も MassiveThreads の 264 倍遅い



縦軸: 実行時間，横軸: ノード数

スケジューラを組み合わせた際の現状の問題点

- DSM が遅い
  - スケジューラもそれに依存しているので遅い
  - フェンスを呼び出さない状態でも，pin/unpin は呼び出される
- Active Message の性能はスケジューラ単体にも影響
  - ワークスティーリングが Active Message に依存しているため

結論

- 「共有メモリ」+「スレッド」を分散メモリ型計算機上で実現
- ソフトウェア分散共有メモリ
  - Self-invalidation/downgrading と P/S3 分類を用いた分散ディレクトリ
  - オーナーの再配置，ブロックサイズ変更が可能なように実装
  - スケジューラ開発に必要な機能を導入（pin/unpin 関数・非同期フェンス）
- 分散スレッドスケジューラ
  - DSM でのコールスタック管理
  - Self-invalidation/downgrading フェンス命令をスケジューラに挿入
- 性能チューニングはこれから

MassiveThreads/GAS [秋山ら '12]

- 2 年前に取り組んだプロジェクト
  - マルチスレッド化，
    RDMA 化 [遠藤ら '15]
- 特徴 : PGAS + 「手動のキャッシュ」
  - キャッシュの有効・無効をプログラマが判断
  - 「コヒーレンスプロトコルを使わないからスケーラブル」？
  - この種のプログラミングモデルに向き合うには，コヒーレンスをシステムが保たない利点を明確にすべき（であった）

```
void* localize(madm_ptr_t, size_t,
    madm_flag_t);
void commit(madm_ptr_t, void*, size_t);

madm_ptr_t gp = /*...*/;
size_t sz = /*...*/;
// 必ずキャッシュを使いまわす
void* p = localize(gp, sz, REUSE);
// or 必ず最新データをキャッシュ
p = localize(gp, sz, UPDATE);

// キャッシュを使って作業

// 必ずキャッシュを書き戻す
commit(gp, p, sz);
```

各操作の流れ (1)

- load miss
  1. 自ノードを “reader” として記録
  2. オーナーの情報と，共有の有無（is_shared）を取得
     - 初回アクセスの場合は，自ノードがオーナーとなる (first touch)
  3. オーナーからブロック全体を（RDMA で）読む
  4. ページを読み込み可能にする
- store miss
  1. 自ノードを “writer” として記録
  2. 自分が最初の writer の時だけ，**Self-invalidation** の有効化要求を送信
     - 他のシェアラー上の “shared” を 1 に変える
     - 他のシェアラー上の “state” は変えない
       （次にシェアラーが自ら SI_fence を呼ぶまで実際には invalidation されない）
  3. ページを書き込み可能にする

各操作の流れ (2)

- SD_fence
  1. ページを書き込み不能にする
  2. 自キャッシュの state を Shared に
  3. diff の書き出し
  4. ディレクトリから writer を削除
     - 自分が writer でなくなるのでディレクトリを書き換えるが，その完了は待たなくてよい
     - しかし，現状では実装を簡単にするため返答を待ってしまっている
- SI_fence
  1. ページを読み書き不能にする
  2. 自キャッシュを無効化する（state を Invalid に）
  3. ディレクトリから reader を削除
     - この際の通信も同様に返答を待たなくてよい

アクセス集中の問題

- 複数ノードが同一メモリアドレスにアクセスする状況
  - 資源が競合すればするほど低速に
  - 共有メモリに限った話ではない
    （状況によっては MPI でも PGAS でも起きうる）
- c.f. MPI の集団通信
  - Broadcast を共有メモリで単純に表現 → 全ノードが同じアドレスを読む
  - 共有メモリで陽に通信を分散させることは可能 (e.g. [Ramos et al. ’13])
- 解決策の一つ: Probable Owner [Li et al. ’89]
  - オーナーの情報をシェアラーに分散させて “辿る”
  - ディレクトリへのアクセス集中が改善
  - オーナーへのアクセス集中は解決できていない
  - 最良時のレイテンシ増大とトレードオフ

分散ワークスティーリングとコールスタック

- 分散メモリでのユーザレベルスレッド
  - 分散メモリでアドレス空間が共有されていない
  - 「コールスタック内のポインタ」が問題に
- **iso-address** [Antoniu et al. '99] : 分散ワークスティーリングの実装方式
  - 全ノード上の全ユーザレベルスレッドについて，異なる仮想アドレスを割り振っておく
  - 仮想アドレスの消費が激しい

分散ワークスティーリングとコールスタック

- 分散メモリでのユーザレベルスレッド
  - 分散メモリでアドレス空間が共有されていない
  - 「コールスタック内のポインタ」が問題に
- **iso-address** [Antoniu et al. '99] : 分散ワークスティーリングの実装方式
  - 全ノード上の全ユーザレベルスレッドについて，異なる仮想アドレスを割り振っておく
  - 仮想アドレスの消費が激しい
- コールスタックの仮想アドレス消費を削減する手法
  - コンパイラを修正してスタックの扱いを変える
    - Split Stacks, Stackless Coroutines
  - スレッド間で仮想アドレスを重複させて仮想アドレス空間を節約
    - e.g. uni-address [秋山ら '15]，random-address [原ら '11]
    - 他のコールスタックをポインタで指せない問題点
      （追加のプログラミング制約の導入）

コールスタックとメモリフェンス

- pin されたメモリブロックの挙動
  - SI_fence / SD_fence の対象から除外（コヒーレンス無効化）
- なぜ実行中のスレッドのコールスタックは他ノードと通信しなくてよいのか？
  - DRF 仮定から，同期時点まで他スレッドとデータを同期しなくてよい
  - スレッドが中断/終了するまで，他スレッドは変更を読めるかどうか分からない
  - ミューテックスやアトミック命令を実装するなら別で考慮する必要あり
- コールスタック用の API を DSM に導入したら，ただの iso-address とどう違うのか？
  - いつ downgrading を行うかは DSM が決めてよい
  - 実行中でないコールスタックには自由にアクセス可能

Invalidation と Downgrading

- SC とした時にどういう invalidation/downgrading 要求が追加されるか？
- Writer-initiated invalidations
  - 「S だけ」→「1 コアだけ M」に遷移する
  - 書き込みを開始する際に，他のコアのキャッシュを無効化する
  - 無効化がなければ，他のコアはずっと古いキャッシュを読み続ける
- Reader-initiated downgrading
  - 「1 コアだけ M」→「S だけ」に遷移する
  - 読み込みを開始する際に，Writer のコアのキャッシュを書き出させる

Self-invalidation/downgrading

- Self-invalidation
  - Invalidation 要求が送られてくる前に，自キャッシュを予め破棄する
  - acquire 側で必ず Self-invalidation を行えば，writer-initiated invalidations を無くせる
- Self-downgrading
  - downgrading 要求が送られてくる前に，自キャッシュを書き出す
  - release 側で必ず Self-downgrading を行えば，reader-initiated downgrading を無くせる