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

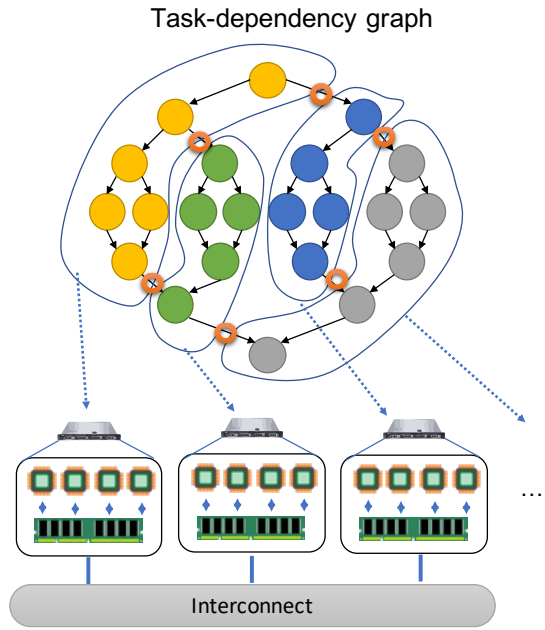
遠藤 亘, 田浦 健次朗

東京大学 大学院 情報理工学系研究科
電子情報学専攻

2018 年 7 月 31 日 @ SWoPP 2018

発表概要

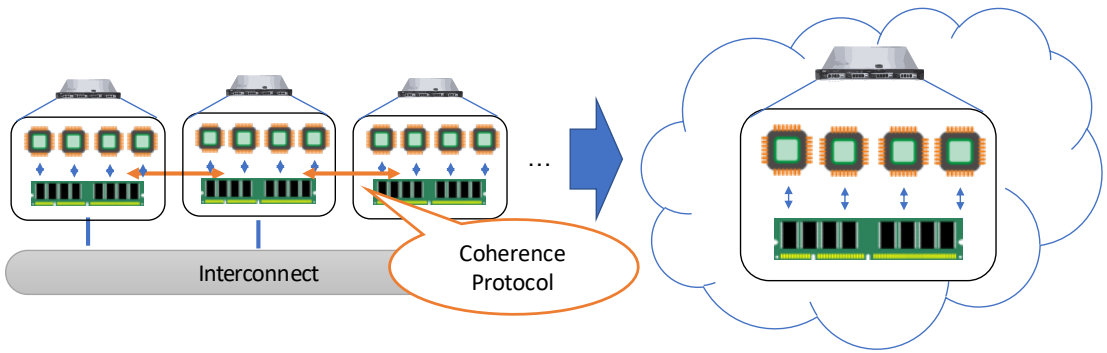
- 研究目的: 分散メモリ型計算機での
  アプリケーション生産性向上
- 中心となるアイディア:
  共有メモリ + タスク並列
  - 並列プログラミングで最も汎用的かつ
    生産的なモデル
- そのための実装方式:
  分散共有メモリ + 分散タスクスケジューラ



- 今回の発表: 分散共有メモリの実装について
  - SWoPP 2017 での発表を踏まえつつ再実装した
  - 新規手法を導入しつつ，アプリケーションの評価も可能になった

序論: 分散共有メモリ

- **分散共有メモリ (Distributed Shared Memory, DSM)**
  - 分散メモリ型計算機上での共有メモリシステム
  - ハードウェア・ソフトウェア両方で実装可能
  - システムがキャッシュメモリを自動的に同期

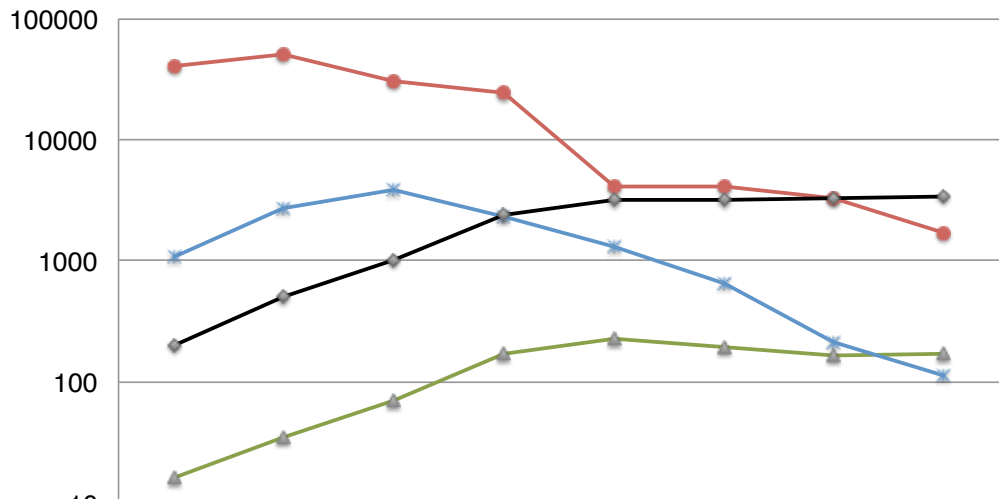


- DSM の研究史
  - 1990 年代: DSM が盛んに研究される
    - e.g. TreadMarks [Keleher et al. ’94], JIAJIA [Hu et al. ’98]
  - 2000 年代以降: **PGAS** の研究に置き換わっていく
    - e.g. UPC [El-Ghazawi et al. ’02], Global Arrays [Nieplocha et al. ’06], OpenSHMEM [Chapman et al. ’10]

序論: DSM 研究の意義 (1/2)

1. インターコネクトの相対的な性能向上 [Ramesh '13]
   - ノード間レイテンシ / DRAM レイテンシ
     ≈ 1000 倍 (1990 年代) → 10 倍 (2010 年代)
   - ノード間バンド幅 / DRAM バンド幅
     ≈ 500 倍 (1990 年代) → 2.5 倍 (2010 年代)
   - シリコンフォトニクスなどの次世代技術でまだまだ縮む可能性

Trends: DRAM, Network Latency & Bandwidth normalized to CPU cycles

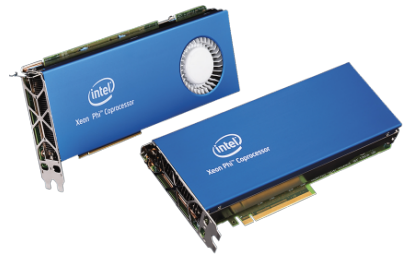


	1992	1994	1997	2000	2005	2007	2009	2011
DRAM min Latency (CPU cycles)	16	35	70	168	224	192	165	170
Net. max BW as CPU cycles/KB	1092	2731	3901	2313	1311	655	211	111
Network min Latency (CPU cycles)	40000	50000	30000	24000	4160	4160	3300	1700
CPU speed (MHz)	200	500	1000	2400	3200	3200	3300	3400

[Kaxiras et al. 15]

序論: DSM 研究の意義 (2/2)

2. マルチコア / メニーコアの一般化
   - コヒーレントキャッシュが性能上のボトルネックの一つ



Intel Xeon Phi [3]

   - ハードウェアのマルチコアコヒーレンスでも，ソフトウェア DSM 研究で培われた成果が取り入れられている
     - e.g. Relaxed consistency [Ros et al. '12] [Sung et al. '15], multiple-writer protocols [Zhao et al. 13]
   - 数十コアまでなら，共有メモリのモデルは成功している
     - 本当にそれ以上にスケールしないのか，しないなら根本的な原因を解明する

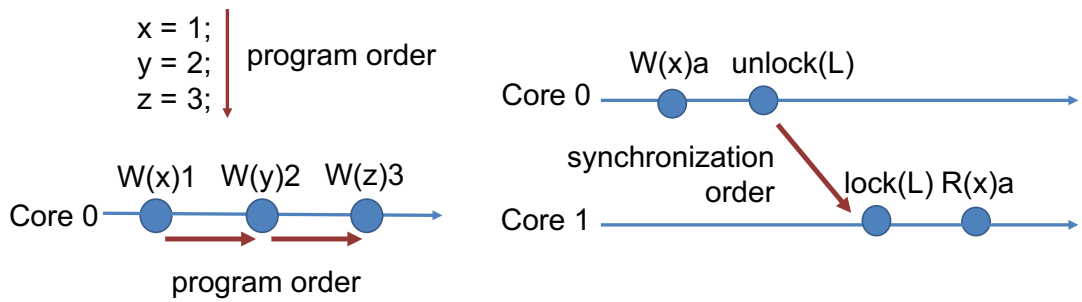
[3] https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

背景

- DSM を実現するための背景知識
  1. メモリコンシステンシモデル
     - Strict / Relaxed
  2. 透過的キャッシュの実現方式
     - ページベース / コンパイラベース
  3. 書き込み管理手法
     - Single-Writer / Multiple-Writer
     - ホームベース / ホームマイグレーション
  4. キャッシュ無効化の判定手法
     - ディレクトリベース / Write notice / タイムスタンプベース

背景: 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
  - 共有メモリで，メモリ読み書きの結果を保証するモデル
- **Sequential Consistency**: 最も厳しいモデル
  - (1) 全アクセスの全順序 / (2) プログラム順序の 2 つに従う
- 緩和型コンシステンシ
  - (1) 同期アクセスの同期順 / (2) プログラム順序の 2 つに従う
  - 全順序の代わりに，happens-before 半順序を用いる
- **SC-for-DRF** [Adve et al. 90]
  - Data-Race-Free プログラムに対して，Sequential Consistency と等価な結果を保証
  - Java や C++11 などで採用

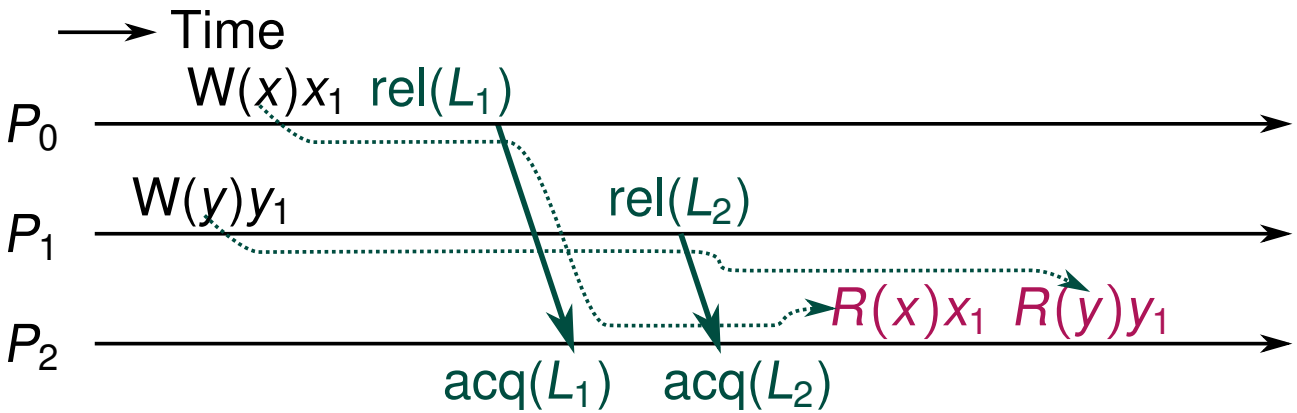


背景: 透過的キャッシュシステムの実現方式

- ソフトウェアで透過的キャッシュを実現する 2 つの手法
  - **ページベース**: メモリ保護機構 + シグナルハンドラ
  - **コンパイラベース**: コンパイラで全アクセスを置換
- ページベースの利点
  - キャッシュヒット時のオーバーヘッドなし
  - コンパイラ拡張を必要としない
- ページベースの欠点
  - システムコールのオーバーヘッドが発生
  - キャッシュブロックサイズをページサイズ未満にできない
- 今回はページベースを前提
  - ライブラリのみで実現でき，開発コストが低いため
  - 現在なら LLVM が利用可能で，コンパイラベースも開発しやすくはなった

背景: 書き込み管理 (1/3)

- 問題: false sharing
  - 複数ノードが同一キャッシュブロック上の別々のワードに書き込む状況
  - 後続する読み込みでは全ての書き込みを読める必要

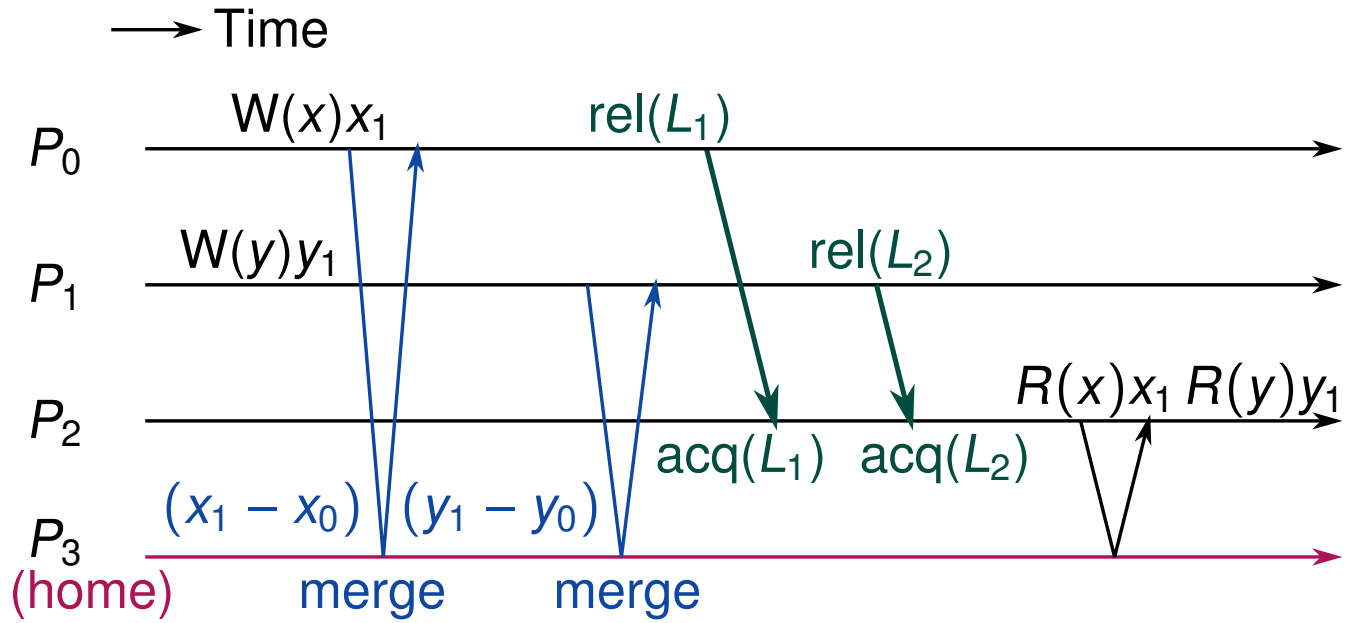


- 解法 1: Single-Writer protocol
  - シンプルに実装できるので，マルチコアプロセッサでは一般的
  - 前回の writer 上で `mprotect()` を用いて書き込み不能にする必要があるので，ページベース DSM で RDMA 化は困難
- 解法 2: **Multiple-Writer** protocol
  - ノード内に twin（複製）を作っておき，後で diff（差分）を適用
  - ソフトウェア DSM で一般的

背景: 書き込み管理 (2/3)

- **ホームベースプロトコル** [Zhou et al. '96]
  - Multiple-Writer 型の実装方式の一つ
  - diff をホームに集約する
  - "ホームレス型" と比べて，計算量・容量両面で利点が多い

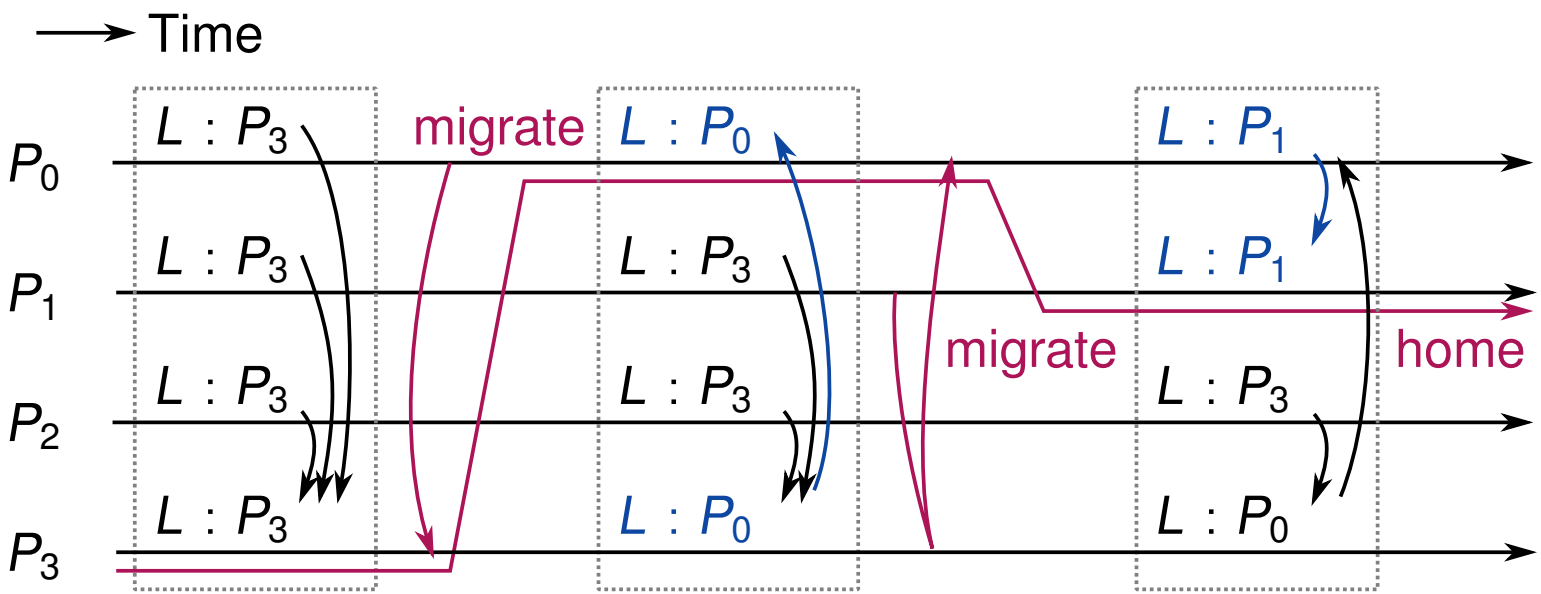
init. $x = x_0, y = y_0$, $x\&y$ are on the same page



- ホームベースの問題点
  - "固定されたホーム" が書き込むノードと一致しないと，書き込みレイテンシが増大

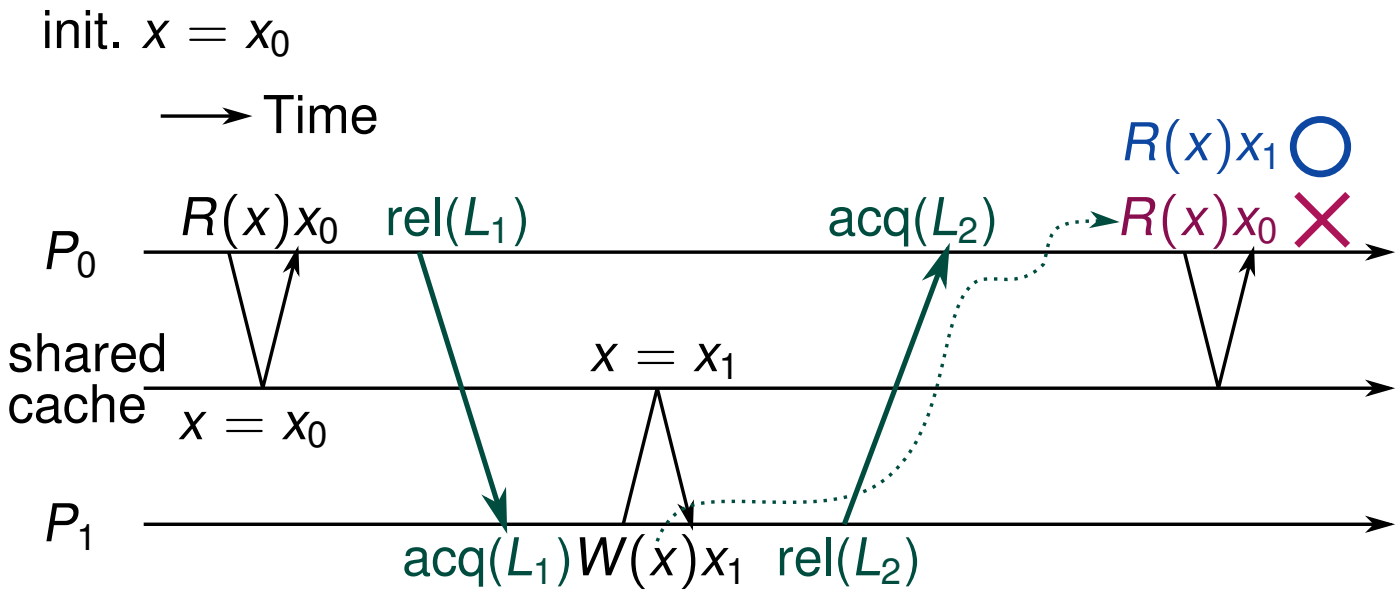
背景: 書き込み管理 (3/3)

- **ホームマイグレーション**: ホームを動的に移動する
  - “ホームを指すメタデータのコヒーレンス” を保つ必要
- ホームマイグレーションの手法 3 つ
  - ブロードキャストベース，ディレクトリベース
  - **Probable owner** [Li et al. ’89]
    - 全プロセスでホームへのリンクを持ち合う
    - ホームの決定を動的かつ分散化



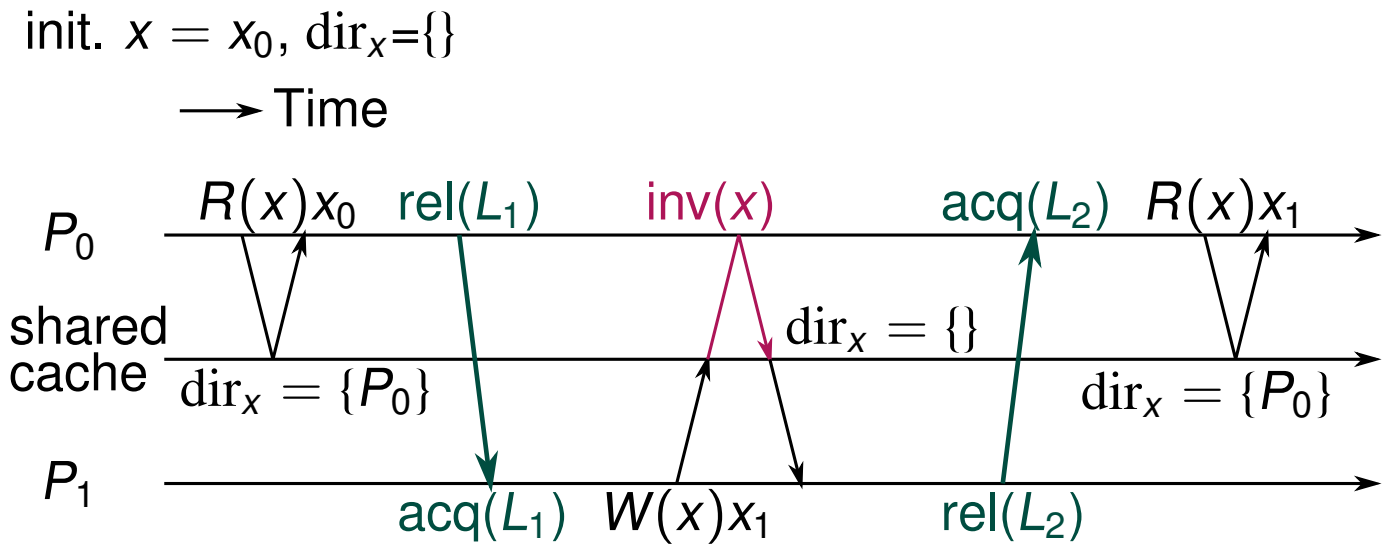
背景: キャッシュ無効化手法 (1/5)

- 書き込みが発生した場合，古いキャッシュを無効化しなければならない
  - 緩和型コンシステンシの場合，happens-before 半順序に基づいて無効化
- 問題: 無効化すべきキャッシュブロックの検出
  - どのキャッシュブロックを無効化すべきかを効率よく計算するには？
  - ブロック ID を元とする集合演算に相当:
    { プロセス P_a が無効化すべきブロック } =
    $(\bigcup_P$ { プロセス P が以前書いたブロック }$) \cap$ {P_a がキャッシュ中のブロック }



背景: キャッシュ無効化手法 (2/5)

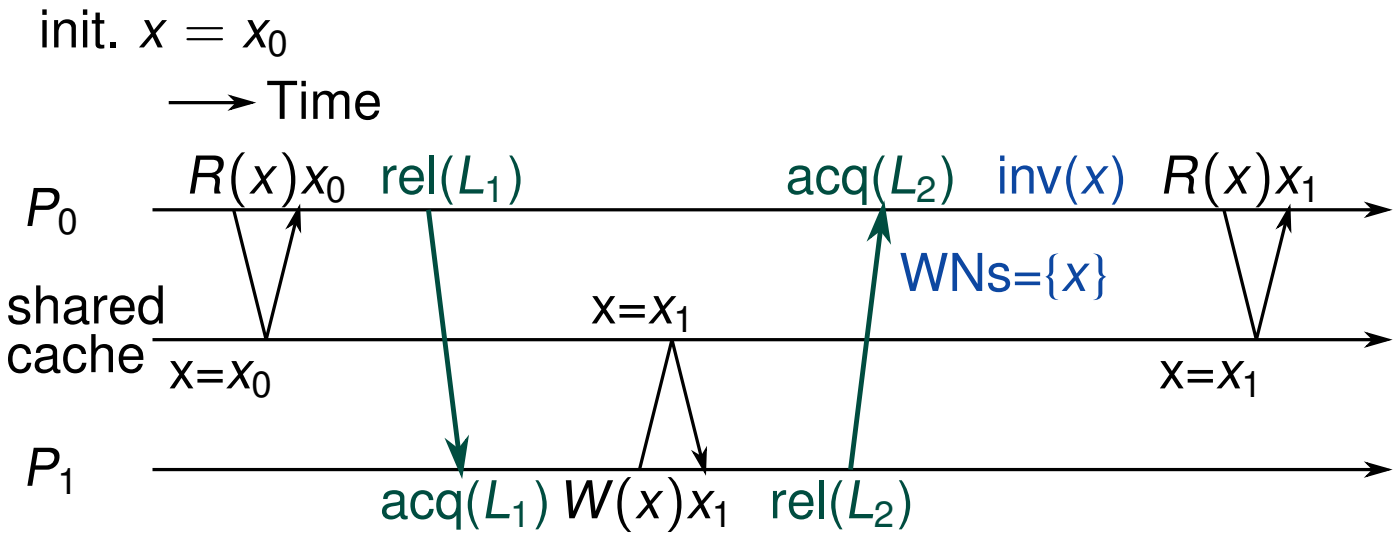
- **ディレクトリベースコヒーレンス** [Lenoski et al. '90]
  - キャッシュ共有中のプロセスの ID を，“ディレクトリ’ に記録
  - 書き込みが起きたら記録されたプロセスに無効化メッセージを送信
  - ハードウェア・ソフトウェア問わず，共有メモリの標準的な手法



- ディレクトリの問題点
  1. プロセス ID を記録するために $O(N_P)$ の空間計算量 (N_P: プロセス数)
  2. **細粒度の無効化メッセージ**を大量に送る必要
  3. 状態遷移が複雑になりやすい

背景: キャッシュ無効化手法 (3/5)

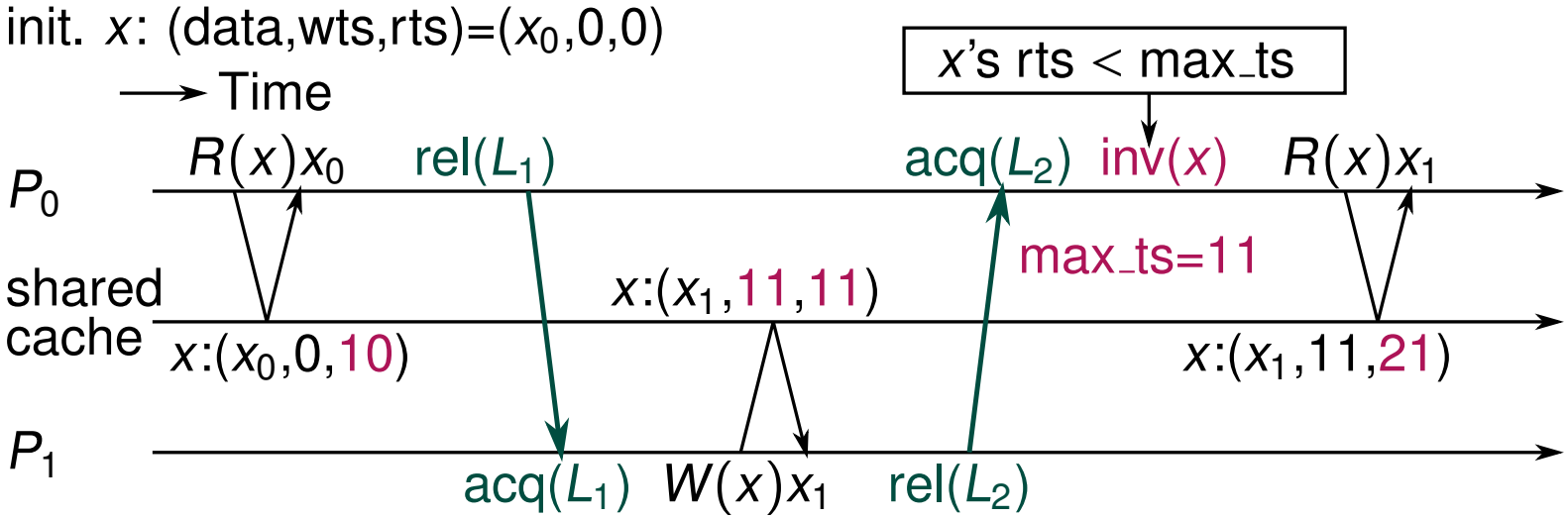
- **Write notice** (WN) [Keleher et al. '94]
  - 書き込まれたブロック ID の配列を **"同期時"** に送る



- Write notice の利点
  - ディレクトリが不要
  - 書き込み時の無効化メッセージが不要
- Write notice の欠点
  - 時間とともに容量を消費していく
    - 最大でキャッシュブロック数に比例した容量が必要
    - TreadMarks は大域的ガーベッジコレクションで対処していた

背景: キャッシュ無効化手法 (4/5)

- **論理タイムスタンプ**によるコヒーレンス [Yu et al. '15]
  - 読み込み時に「いつキャッシュが無効になるか」を記録
  - 書き込み時は読み込み時より大きいタイムスタンプを保持
  - 同期が起きたら，タイムスタンプのみを送ってキャッシュ無効化



- 論理タイムスタンプによる無効化の利点
  - $O(\log N_P)$ の空間計算量で済む
  - 無効化メッセージの送信が不要
- 論理タイムスタンプによる無効化の欠点
  - キャッシュミス数が増大

背景: キャッシュ無効化手法 (5/5)

- Lamport clock は半順序を正確に表現できない
  - 仮定: a, b: 並行システム上のイベント (e.g. メモリアクセス),
    $C(a)$: a の Lamport clock, $\overset{\text{hb}}{\longrightarrow}$: happens-before 半順序
  - $C(a) < C(b) \Rightarrow a \overset{\text{hb}}{\longrightarrow} b \cup \neg(a \overset{\text{hb}}{\longrightarrow} b \cup b \overset{\text{hb}}{\longrightarrow} a)$
- Lamport clock によるキャッシュ無効化
  - 仮定: $a = W(x),\ b = R(y),\ C(a) < C(b)$
  - 下の 2 つの状況は判別不能なため，保守的に無効化するしかない
    1. $a \overset{\text{hb}}{\longrightarrow} b$: （真の）read-after-write 依存 $\Rightarrow$ 無効化が必要
    2. $\neg(a \overset{\text{hb}}{\longrightarrow} b \cup b \overset{\text{hb}}{\longrightarrow} a)$: **concurrent** $\Rightarrow$ 無効化が不要
- happens-before 半順序を正確に判定するには？
  - Vector clock を使用すれば可能
  - しかし，$O(N_P)$ の空間計算量を必要とする [Charron-Bost '91]
  - キャッシュブロック毎に管理するには大きすぎる

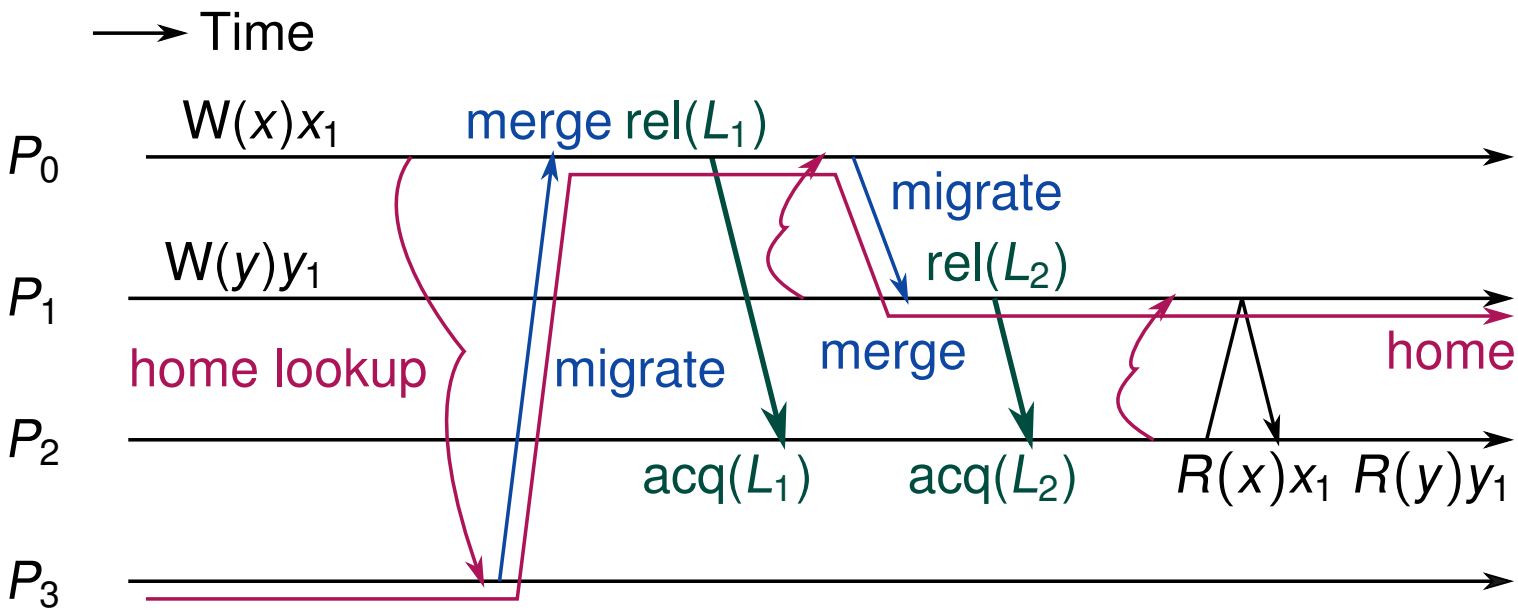
提案手法: DSM の実装方針

- 実装全体の方針（ArgoDSM [Kaxiras et al. '15] を参考）
  1. 全面的な RDMA 化
     - アクティブメッセージの排除
  2. ライブラリのみでの実装
     - コンパイラ改変が不要
  3. ユーザレベルスレッドの活用
- 従来の DSM になかった手法
  1. ホームマイグレーションを**「常に」**行う手法
  2. ディレクトリを用いないキャッシュ無効化
     - **論理タイムスタンプ + Write notice**
  3. 状態遷移の改良（発表では省略，資料のみ）
     - clean/dirty の分割，コールスタックのための状態追加

提案手法: ホームマイグレーション (1/2)

- 提案手法 (1): ホームマイグレーションを**「常に」**行う
  - ホームは Probable owner によって検索する
  - diff の適用は必ず「**移動先**」で行われる

init. $x = x_0, y = y_0$, $x\&y$ are on the same page

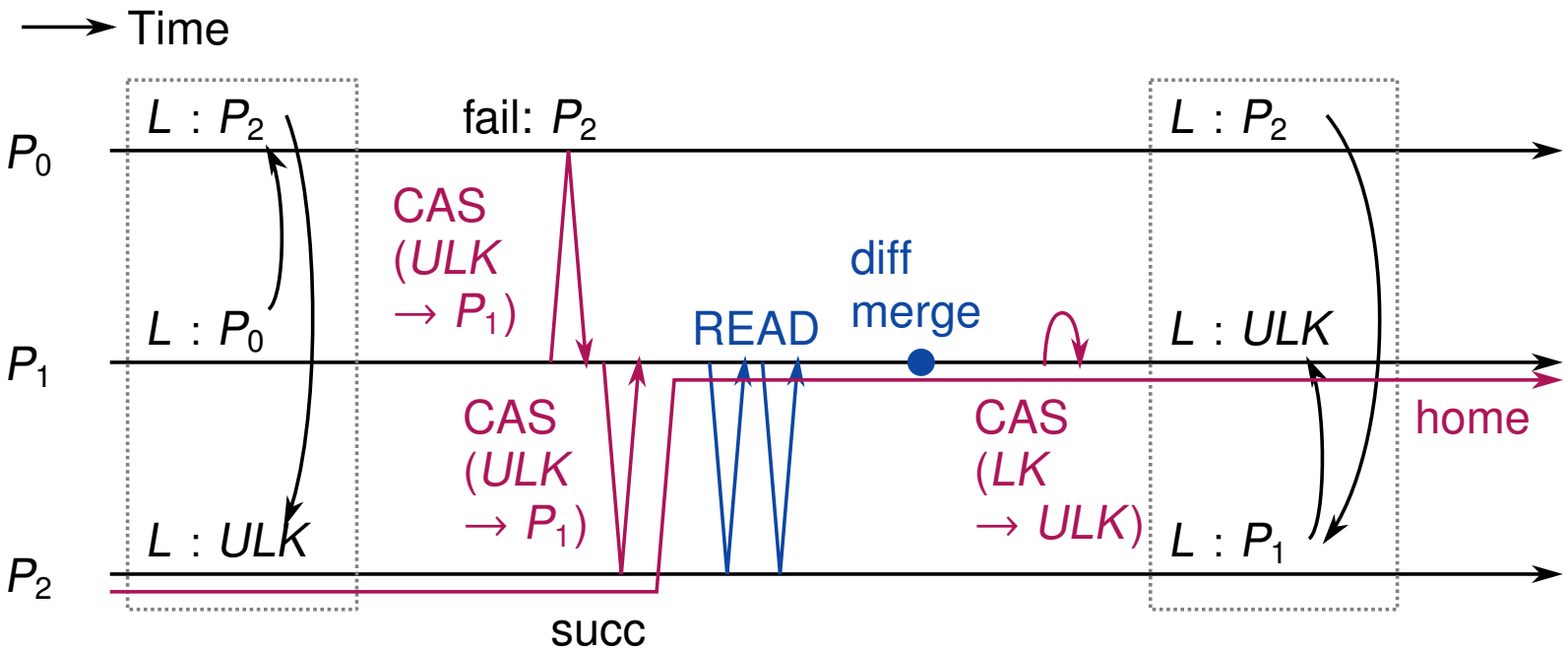


提案手法: ホームマイグレーション (2/2)

- 本手法の利点
  1. 書き込みを頻繁に行うノードにホームが自動的に移動
     - 書き込みの局所性があれば高速化が見込める
  2. RDMA WRITE を細切れに発行する必要がない
- 本手法の欠点
  1. マイグレーションの並列化は困難
     - 必然的にクリティカルセクションで排他制御する必要
     - False sharing 時の性能は，単純なホームベース型より低下
- 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と，
  以下の機能の実装が容易になる:
  1. ホームマイグレーション
  2. diff 併合処理: SIMD による XOR
  3. 論理タイムスタンプの管理
  4. DSM でのアトミック命令の実装

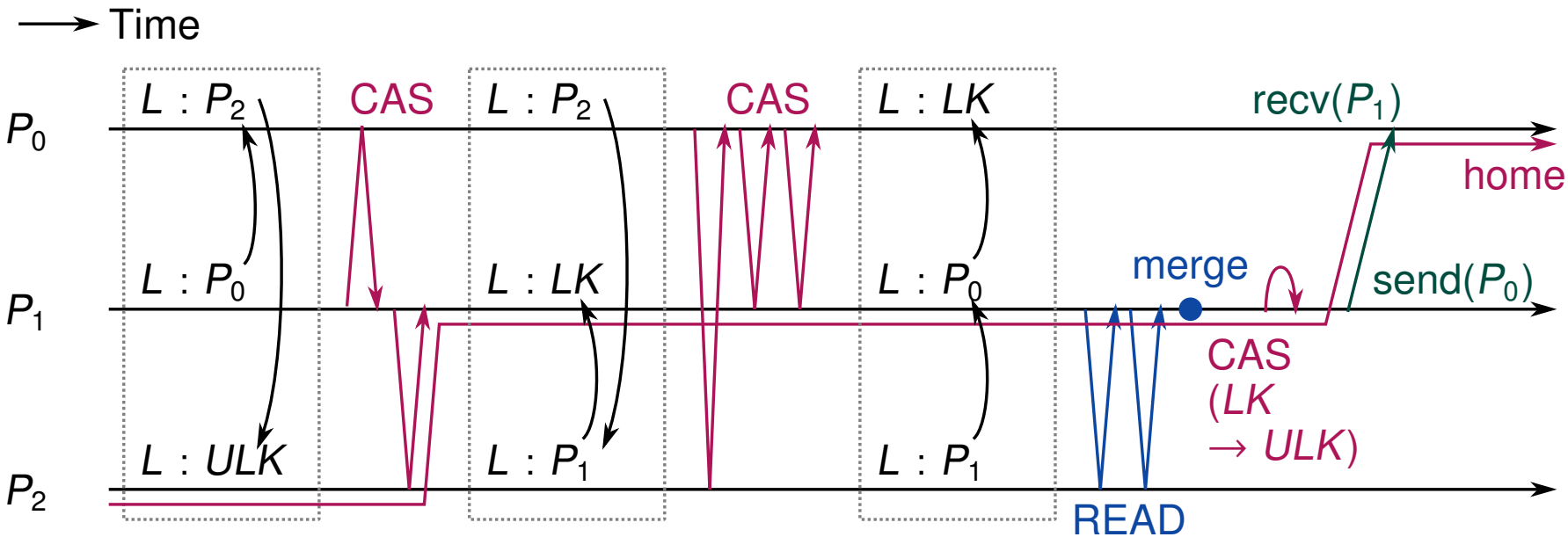
提案手法: ブロック毎のクリティカルセクション (1/2)

- クリティカルセクションの実行の流れ
  1. Probable owner を RDMA CAS で辿ってホームを検索 & ロック
  2. 古いホームから RDMA READ でタイムスタンプとデータを読み取る
  3. ローカルに diff 併合を行う
  4. RDMA CAS でアンロック



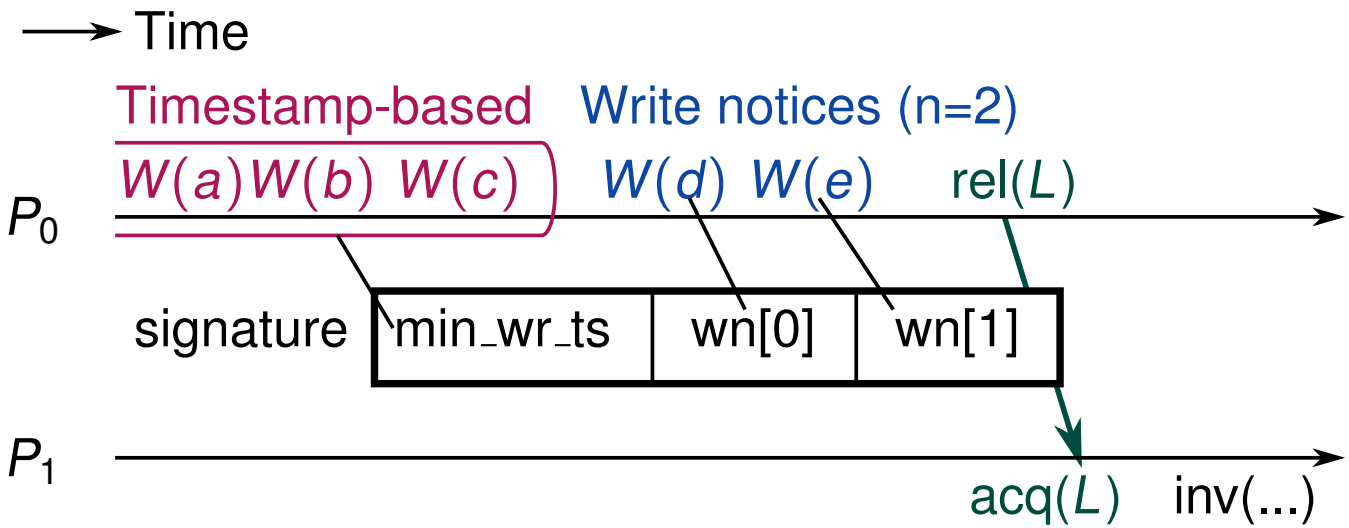
提案手法: ブロック毎のクリティカルセクション (2/2)

- ロックのスケーラビリティ
  - MPI で RMA を同じプロセスに繰り返すと，送信先の MPI progress を圧迫して，ライブロックに近い状態になる
- **キューロック**によるスケーラビリティ向上
  - MCS ロックに近い形で実装
  - ロック待機中のスレッドをリストとして保持
  - Probable owner と自然に組み合わせることが可能



提案手法: キャッシュ無効化 (1/4)

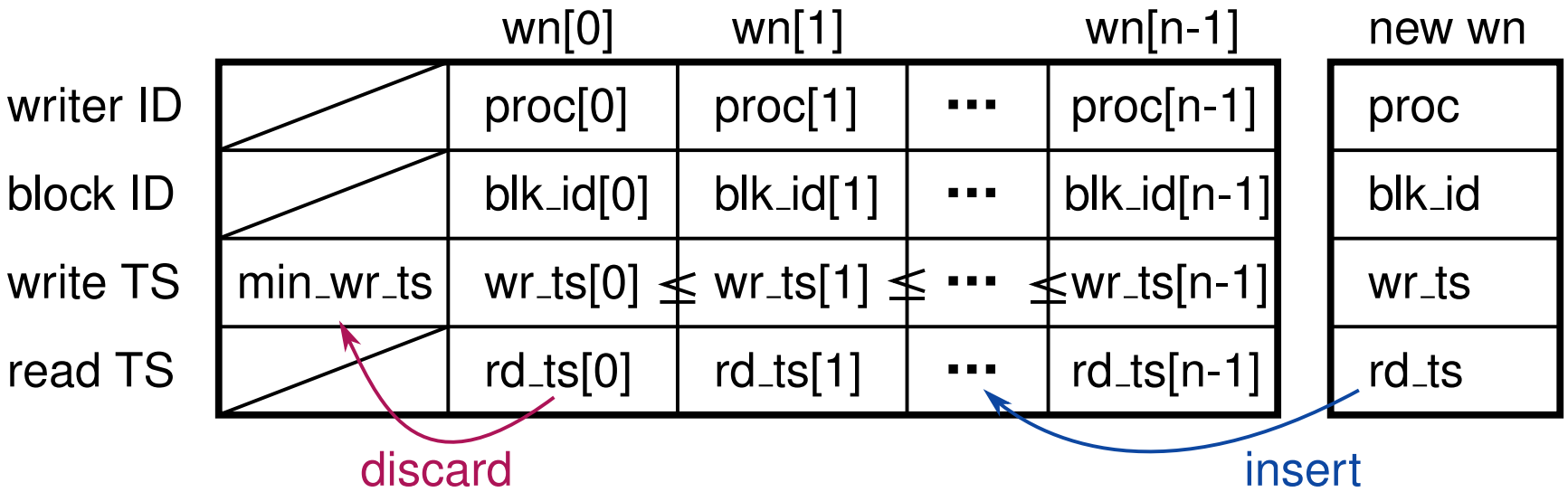
- 提案手法 (2): 論理タイムスタンプと Write notice の併用
  - 新しい書き込みを一定数だけ Write notice で無効化
  - 古い書き込みを論理タイムスタンプで無効化



- 併用すること自体は新規ではない
  - e.g. 物理タイムスタンプ & ブルームフィルタ [Yao et al. '16]
  - 論理タイムスタンプと Write notice の相性の良さに着目したのは, 本研究がおそらく初

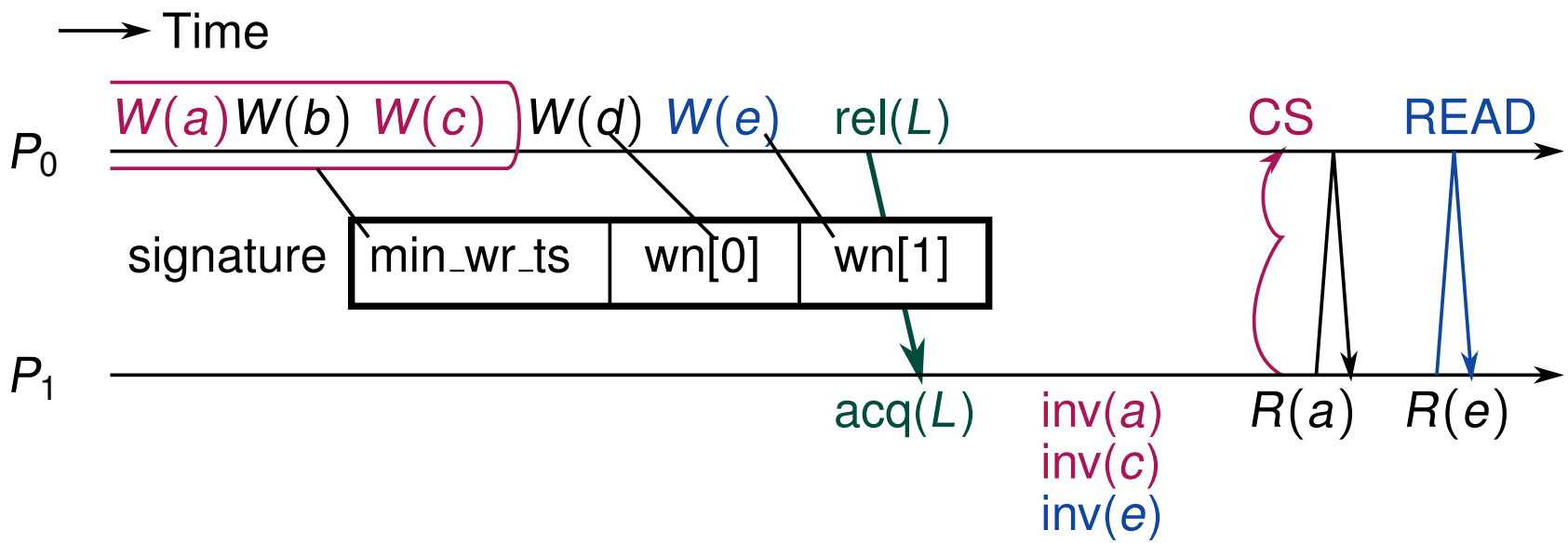
提案手法: キャッシュ無効化 (2/4)

- “signature”: 「先行する全書き込みの集合」を意味
  - write notice の配列 + “min_wr_ts” の 2 つで構成
- 書き込みの度に write notice を追加
  - 上限数に達した場合，**write timestamp が最小の write notice を破棄**
  - この時，捨てる write notice に関して min_wr_ts を更新
    min_wr_ts := max(min_wr_ts, wr_ts[0])



提案手法: キャッシュ無効化 (3/4)

- 論理タイムスタンプで無効化されたブロック
  - どのプロセスが書き込んだかの情報が失われている
  - Probable owner を辿ってホームを検索し,
    クリティカルセクションを使って read timestamp を更新
- Write notice で無効化されたブロック
  - 書き込んだプロセスの ID / read timestamp などの情報も付与
  - 読み込み時に最新のホームを検索する必要がない
  - **書き込んだプロセスから RDMA READ を 1 回行うだけで読み込める**



提案手法: キャッシュ無効化 (4/4)

- 全プロセスでのバリアの実装
  1. 全プロセスが release バリアを発行
     - 全ての書き込みがホームに反映される
     - この時，前述のクリティカルセクションを実行
  2. signature を `MPI_Allgather()` で交換
     - 現在の実装方法では $O(N_P)$ の時間・空間計算量を必要とする
     - **signature 同士の和集合**を計算すれば，
       “reduction” として並列化することは原理的に可能
  3. 全プロセスが acquire バリアを実行
     - 全プロセスの signature に基づいて全プロセスが無効化
- アトミック命令やスレッド移動時の処理
  1. release 側が release バリアを発行
  2. （何らかの形で）signature を転送する
  3. acquire 側が acquire バリアを発行
  4. **signature 同士の和集合**を計算
     - happens-before 半順序に従うために必要

実装概要

- DSM ライブラリと（単純な）OpenMP ランタイムを実装
- DSM ライブラリ
  - ノード間通信には MPI を使用
  - RDMA 前提の通信にも MPI-3 RMA を使用
- OpenMP ランタイム
  - GCC の OpenMP ランタイムをリンク時に乗っ取る
    - コンパイラ改変なしで，OpenMP プログラムがそのまま動作
  - 現在は static scheduling での parallel for などのみ実装
  - reduction や threadprivate などは実装できていない
    - GCC がプロセッサのアトミック命令を出力するため
- ノード内のスレッドスケジューリングには MassiveThreads を使用
  - ノードあたり 1 プロセスで実行することを想定

評価手法

- 評価には NAS Parallel Benchmark を使用
  - 公式版は Fortran だったため，非公式な C & OpenMP 移植版を使用
    - DSM の仕組み自体は言語非依存であり，Fortran 対応も今後検討
  - 今回は NAS EP, CG, BT の 3 つを試した
  - サポートできていない OpenMP の機能は書き換え
    - reduction は逐次ループ化
    - グローバルな threadprivate 変数は手動でコピー
- 東大情報基盤センターの ReedBush-U で実験

CPU	Intel® Xeon® E5-2695 v4 2.1 GHz (max. 3.3 GHz with Turbo boost) 18 cores × 2 sockets / node
Memory	256GB / node
Interconnect	Mellanox® Connect-IB® dual port InfiniBand EDR 4x
OS	Red Hat® Enterprise Linux® 7.2
Compiler	GCC 4.8.5 (with the option “-O3”)
MPI	MVAPICH 2.2

評価結果: NAS EP

- NAS EP に関してのみ，複数ノードを使用した時の性能向上を達成できた
  - Embarrassingly-Parallel なベンチマークであるため，
    DSM の性能はあまり影響しない
  - 完全にスケールしているようにも見えない
    - EP も開始時・終了時には DSM のアクセスが発生

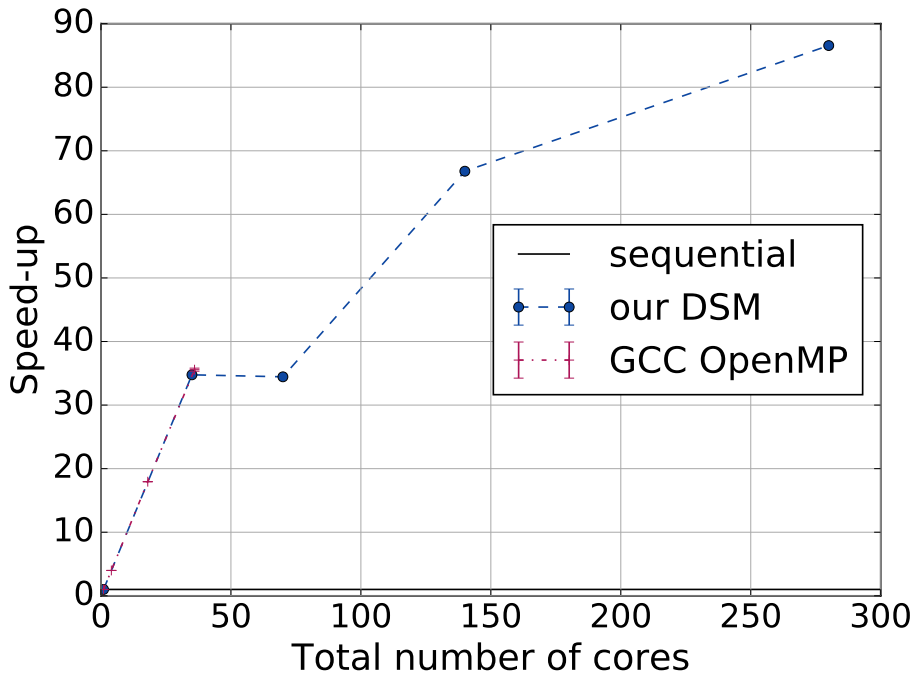


Figure 1: NAS EP (CLASS=C)

評価結果: NAS CG, BT

- NAS CG, BT では，GCC の OpenMP ランタイムより高速化できていない
  - 現状ではマルチノード化している意味がない
  - BT に関しては，逐次性能よりも遥かに遅い
    - 正確な原因は現在調査中

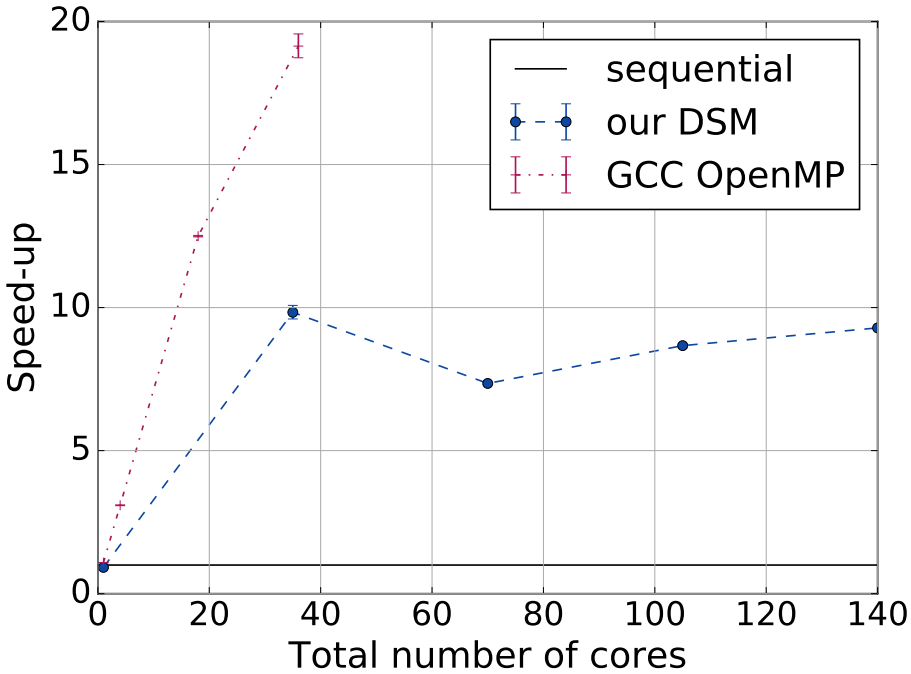


Figure 2: NAS CG (CLASS=C)

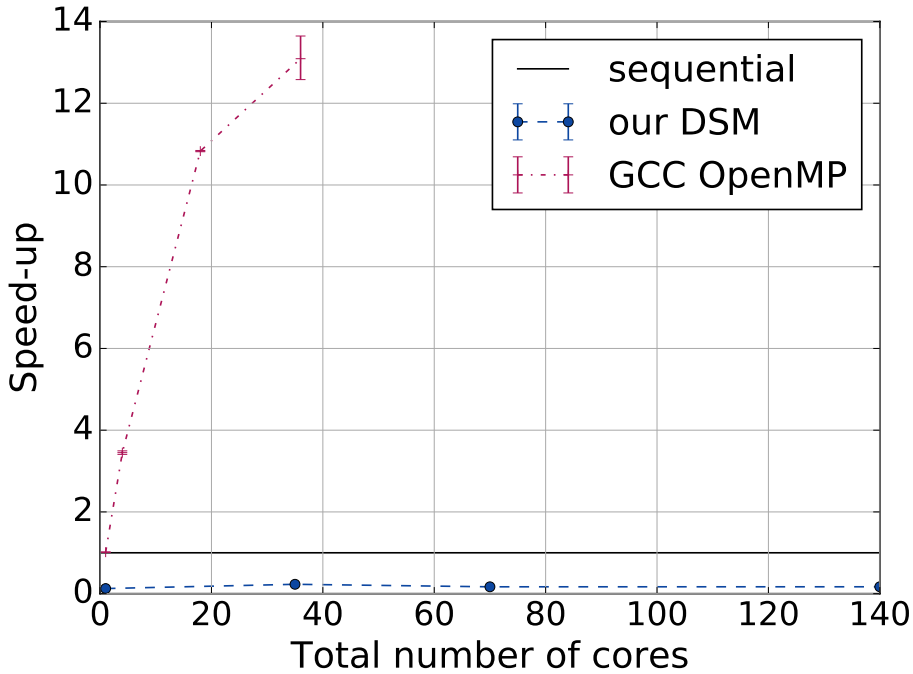
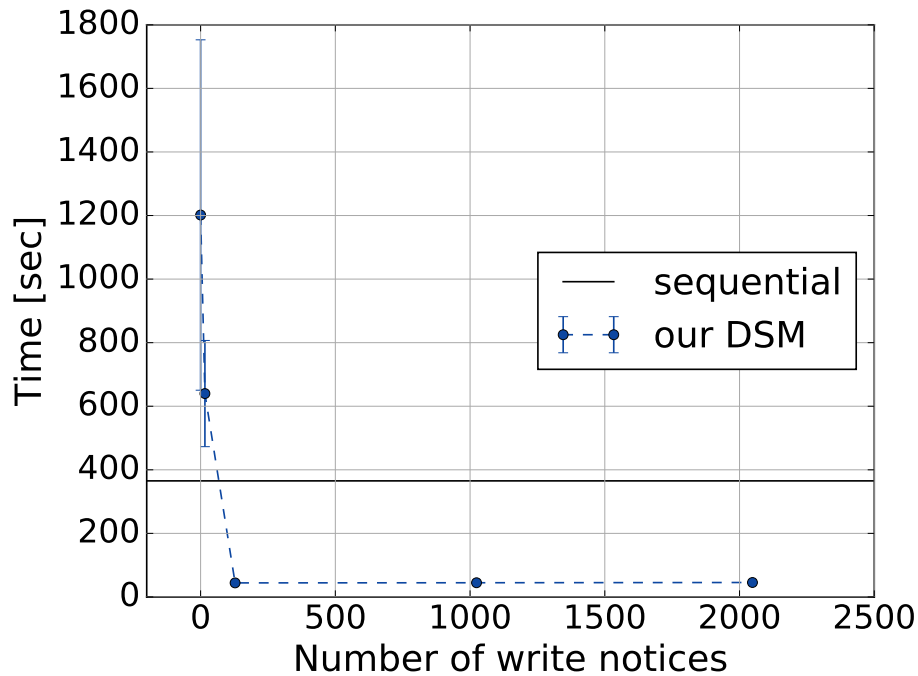


Figure 3: NAS BT (CLASS=A)

評価結果: Write notice 配列長の影響

- 予稿提出後に ReedBush-H で実験
- signature 内の write notice 配列長を変化させる
  - デフォルトでは 1024
  - 0 にすると，論理タイムスタンプのみによる無効化になる
- NAS CG を 4 ノードで動かした時の実行時間
  - ある長さ以下になると，急激に時間が増大
  - 論理タイムスタンプの無効化のみでは，クリティカルセクション処理が多発して性能低下



実装上の問題

- シグナルハンドラ内のコンテキストスイッチでのバグ
  - キャッシュミス時のノード間通信の通信遅延を隠蔽するため，シグナルハンドラ内で（ユーザレベル）スレッドを切り替えたい
  - コンテキストスイッチを有効にすると，NAS BT で結果が不正になるバグが発生
  - 根本的な原因が不明で，そもそも Linux カーネルが正常実行できるパターンなのかも不明
  - 今回はコンテキストスイッチを全て無効にして評価
    - 以前提案した通信のソフトウェアオフローディング機構とも相性がよくない
- MPI のマルチスレッド時のバグ
  - MVAPICH を用いて MPI_THREAD_MULTIPLE で実行すると，RMA アトミックの結果が不正になることがある
  - 現状は MPI 関数呼び出しを全てスピンロックで逐次化して対応

結論

- 分散共有メモリの実装手法を提案
  - ホームマイグレーションを常に行う方式
  - 論理タイムスタンプに基づくキャッシュ無効化
  - 状態遷移の改良（発表では省略）
- NPB での性能評価結果
  - NAS EP 以外では依然として低速
- 今後の課題
  - NAS BT などでの性能低下の原因究明
  - 各機能のマイクロベンチマーク
  - 他処理系 (DSM, MPI, PGAS) との性能比較
  - シグナルハンドラとコンテキストスイッチ併用時の問題
  - プリフェッチャなどの高速化機構の導入
  - 最適な lease 値の予測機構の導入
  - DSM 上でのワークスティーリングの（再）実装