分散ワークスティーリングと協調するキャッシュ機構を備えたPGAS処理系

第185回 HPC研究会 @ SWoPP 2022

椎名 峻平, 田浦 健次朗

東京大学大学院 情報理工学系研究科

2022.07.29

負荷分散とハードウェア階層

- **負荷分散**は並列計算において極めて重要
  - 特に不規則な並列性を生むプログラム（疎行列，木構造，AMR，...）

ダム崩壊の流体シミュレーション

現代における負荷分散:

- **マルチコア/メニーコア CPU**
  - 増えていくコア数 → より多くの並列性が必要
- **深化するメモリ・キャッシュ階層**（階層キャッシュ，NUMA，マルチソケット）
  - キャッシュ階層の局所性を意識しつつ負荷分散
- **分散メモリ**（クラスタ，スパコン）
  - プログラマによる明示的な通信の記述，通信の集約化

プログラマによる手動負荷分散は十分に複雑，生産性を損ねる
→ **処理系による自動負荷分散** で，これらのハードウェア階層を統一的に扱う

分散メモリ上の自動負荷分散の研究動向

- 共有メモリでは: **ワークスティーリング**等の自動負荷分散を備えた処理系が多く存在
  - Cilk, Intel TBB, OpenMP tasks, MassiveThreads, ...
  - → 分散メモリでは?

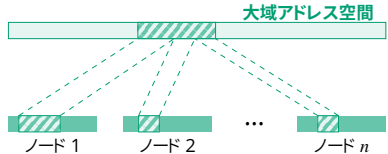
分散メモリにおけるワークスティーリングの研究動向:

- **RDMA** によって分散メモリでも効率的な自動負荷分散(ワークスティーリング)が可能に **[Dinan+, SC '09]**
- Fork-Join への拡張,動的スレッドマイグレーション **[Akiyama and Taura, HPDC '15]**
- Join 処理等の性能改善, 10 万コアでのスケーラビリティ **[Shiina and Taura, Cluster '22] (to appear)**

RDMA の活用により**分散ワークスティーリングは効率的に行えるようになった**
→ 実用的な問題を解くためには**グローバルなデータを効率的に扱う**ことが必要

PGAS によるグローバルデータへのアクセス

前提: 分散ワークスティーリングではタスクが実行されるノードは動的に決まる



- **PGAS: Partitioned Global Address Space**
  - 分散メモリ上にグローバルデータを分割して配置
  - **グローバルポインタ**を介して明示的にグローバルデータへアクセス
  - どのノードからでも同じ方法(API)でグローバルデータにアクセス可能 → **ワークスティーリングとの相性良し?**
- PGAS をワークスティーリングと組み合わせた先行研究も存在
  - Scioto [Dinan+, ICPP '08], HotSLAW [Min+, PGAS '11], Grappa [Nelson+, USENIX ATC '15] など
- **問題点:** ほとんどの処理系は**キャッシュを持たない** → メモリアクセスの局所性を活かせず
  - 特に細粒度なタスク → 個々のタスクが細かい通信を発行
  - 複数のタスクが用いるデータをまとめて通信(集約化)する,などの最適化が困難 (∵ 前提)

→ **本研究では PGAS にソフトウェアキャッシュを実装**

本研究の目標・現状

目標: (Cilk のような) 再帰的な分割統治型の Fork-Join プログラムを分散メモリ上でも効率的に実行
→ **高水準な並列性の記述，自動負荷分散による 生産性の高い並列分散プログラミング**

本発表での現状報告:

- シンプルなソフトウェアキャッシュを PGAS に実装
  - 効率の良いキャッシュ管理のため，PGAS ライブラリに新しい API を導入
  - 分散ワークスティーリングと協調したキャッシュコヒーレンスの管理
- 並列ソート (Cilksort) のベンチマークを評価 → **27,648 コア (576 ノード) までのスケーラビリティ**
  - 並列化効率はあまり良くない (27,648 コアで 542 倍の台数効果) → さらなる性能改善が必要

Outline

背景

提案する PGAS 処理系の設計・実装

実験評価

関連研究

結論・今後の課題

Outline

背景

提案する PGAS 処理系の設計・実装

実験評価

関連研究

結論・今後の課題

タスク並列，分割統治，Fork-Join モデル

- **タスク並列:** 動的に並列タスクを生成し（**fork**），待ち合わせる（**join**）
  - 再帰的にタスクを生成可能 → **分割統治**のプログラムと相性が良い

```
task_group tg;
tg.fork([=] { A(); });
tg.fork([=] { B(); }); // A() と B() が並列に実行される
tg.join(); // A() と B() の実行を待ち合わせる
```

- 多くの並列アルゴリズムを表現できる**汎用的**なモデル
  - 単純な並列 for ループをはじめ，行列演算，FFT，ソート，動的計画法，空間分割木，...
- 高水準で簡潔な並列アルゴリズムの記述が可能 → **高生産性**
- 具体的なコア数を意識せず大量のタスクを生成可能 → **高い移植性**
  - タスク並列処理系が**自動で動的負荷分散**

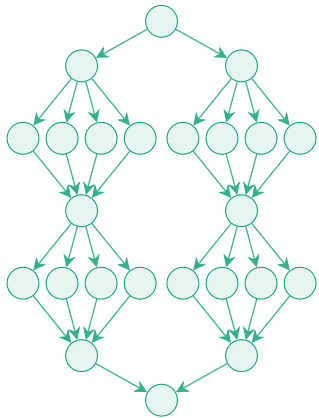
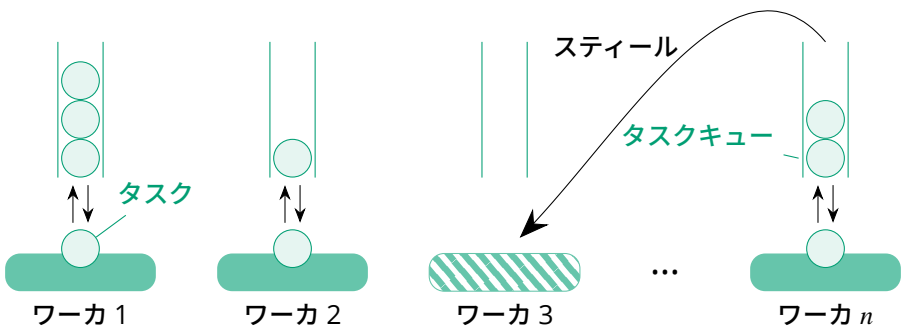


Fig. タスクの依存関係のグラフ

ワークスティーリング [Blumofe and Leiserson, JACM '99]

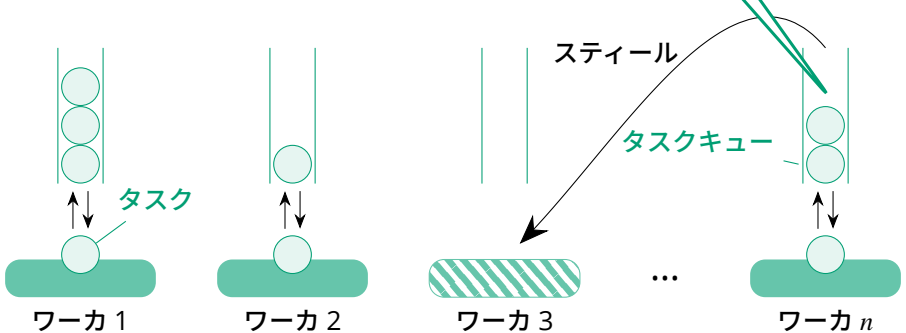
- 広く用いられているタスク並列プログラムのスケジューリング手法
- 各**ワーカ**(CPU コア等)はそれぞれ独立の**タスクキュー**を持つ
- 基本的にワーカは自身のタスクキュー(の下側)からタスクを出し入れして実行
- タスクキューが空になると他のワーカのタスクキュー(の上側)から**スティール**を試みる
  - スティールされるワーカは一様ランダムに選択される(ランダムワークスティーリング)



ワークスティーリング [Blumofe and Leiserson, JACM '99]

- 広く用いられているタスク並列プログラムのスケジューリング手法
- 各**ワーカ**(CPU コア等)はそれぞれ独立の**タスクキュー**を持つ
- 基本的にワーカは自身のタスクキュー(の
- タスクキューが空になると他のワーカのタ
  - スティールされるワーカはランダムに選択

タスクキュー内で最も古いタスク
(タスクツリーにおいて根に近い部分木)が盗まれる
→ **大きい粒度で負荷分散が行われる**

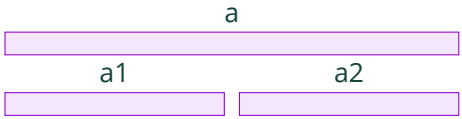


例: 並列マージソート

```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a1, a2] = a.divide_two();
    task_group tg;
    tg.fork([=] { merge_sort(a1); });
    tg.fork([=] { merge_sort(a2); });
    tg.join();
    merge(a1, a2);
  }
}
```

- マージソートを再帰的に並列に行う例
- コード例では配列の範囲(span)を用い記述
  - Span: ポインタと要素数の組

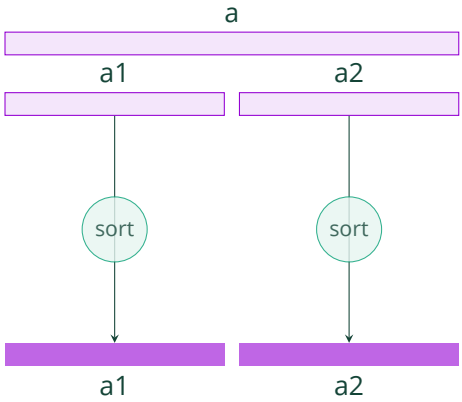
例: 並列マージソート




```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a1, a2] = a.divide_two();
    task_group tg;
    tg.fork([=] { merge_sort(a1); });
    tg.fork([=] { merge_sort(a2); });
    tg.join();
    merge(a1, a2);
  }
}
```

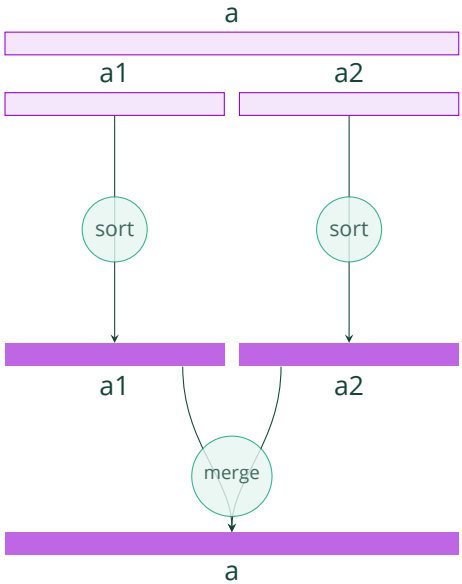
例: 並列マージソート

```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a1, a2] = a.divide_two();
    task_group tg;
    tg.fork([=] { merge_sort(a1); });
    tg.fork([=] { merge_sort(a2); });
    tg.join();
    merge(a1, a2);
  }
}
```



例: 並列マージソート

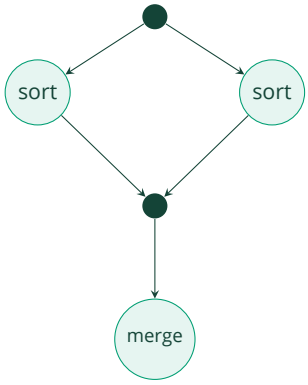
```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a1, a2] = a.divide_two();
    task_group tg;
    tg.fork([=] { merge_sort(a1); });
    tg.fork([=] { merge_sort(a2); });
    tg.join();
    merge(a1, a2);
  }
}
```



例: 並列マージソート

```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a1, a2] = a.divide_two();
    task_group tg;
    tg.fork([=] { merge_sort(a1); });
    tg.fork([=] { merge_sort(a2); });
    tg.join();
    merge(a1, a2);
  }
}
```

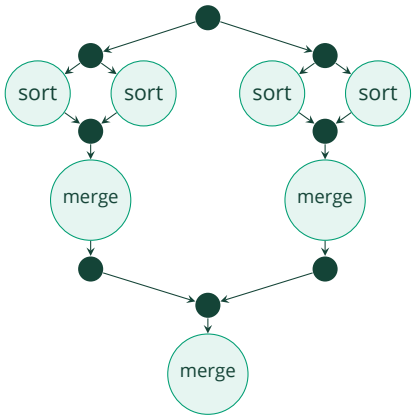
タスク間の依存関係は有向非巡回グラフ(Directed Acyclic Graph; DAG)として表される



例: 並列マージソート

```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a1, a2] = a.divide_two();
    task_group tg;
    tg.fork([=] { merge_sort(a1); });
    tg.fork([=] { merge_sort(a2); });
    tg.join();
    merge(a1, a2);
  }
}
```

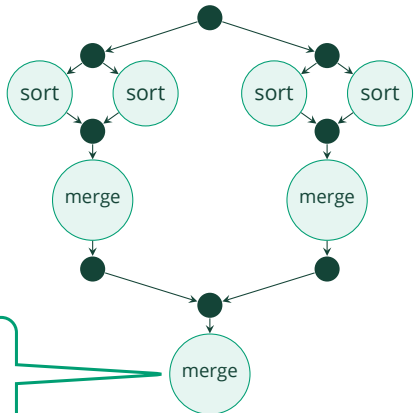
タスク間の依存関係は有向非巡回グラフ(Directed Acyclic Graph; DAG)として表される



例: 並列マージソート

```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a1, a2] = a.divide_two();
    task_group tg;
    tg.fork([=] { merge_sort(a1); });
    tg.fork([=] { merge_sort(a2); });
    tg.join();
    merge(a1, a2);
  }
}
```

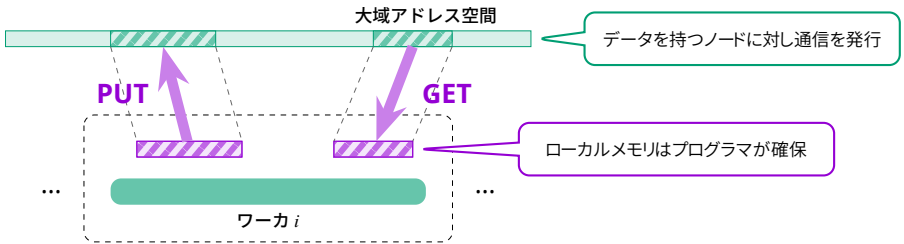
タスク間の依存関係は有向非巡回グラフ(Directed Acyclic Graph; DAG)として表される



実験に用いる Cilksort では
マージ処理も内部で並列化される

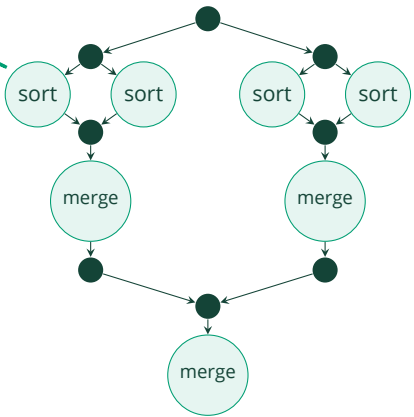
PGAS ライブラリ

- 数多くの PGAS 言語が存在するが，今回はライブラリ実装に焦点を当てる
  - Global Arrays [Nieplocha+, '96]，OpenSHMEM [Chapman+, PGAS '10]，UPC++ [Zheng+, IPDPS '14]
  - 多くは SPMD 的なプログラミングモデルを仮定 → 負荷分散はプログラマの責任
- 主要な操作（API）：
  - **malloc/free:** グローバルメモリの確保/解放．大きい配列はノード間で分割配置される
  - **GET/PUT:** グローバルメモリの読み込み/書き込み
- malloc で返却された**グローバルポインタ**を用い，範囲を指定して GET/PUT で読み書き



タスク並列 + PGAS = ?

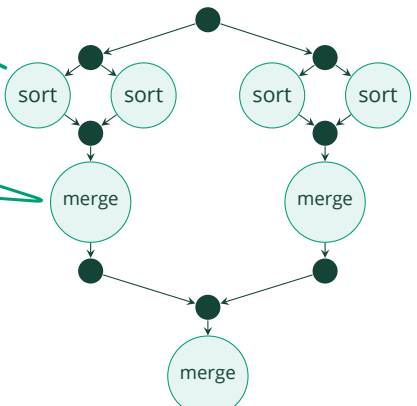
各タスクでグローバル配列に **GET/PUT** でアクセス



タスク並列 + PGAS = ?

各タスクでグローバル配列に **GET/PUT** でアクセス

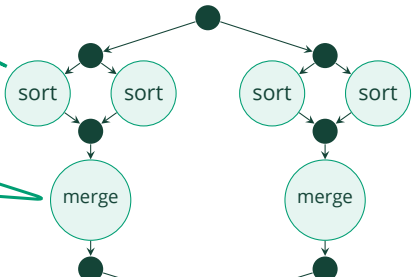
近いタスクではしばしば同じデータを扱う
→ データを再利用して通信を削減したい



タスク並列 + PGAS = ?

各タスクでグローバル配列に **GET/PUT** でアクセス

近いタスクではしばしば同じデータを扱う
→ データを再利用して通信を削減したい



タスクを実行するワーカは動的に決まる
→ **GET/PUT** に用いるローカルメモリをタスク間で再利用することは困難
→ 処理系による**キャッシュが必要**

Outline

背景

提案する PGAS 処理系の設計・実装

実験評価

関連研究

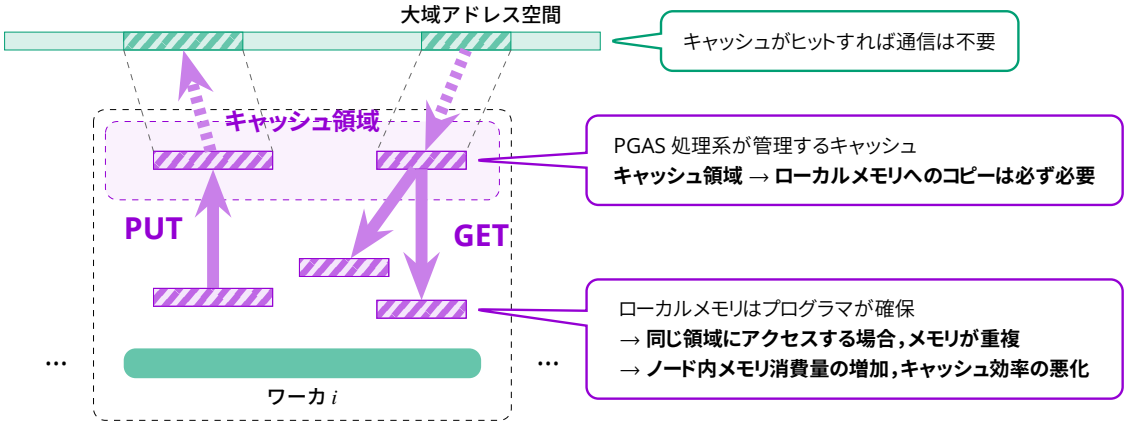
結論・今後の課題

PGAS へのソフトウェアキャッシュの実装 – 概要

- 既存の PGAS 処理系は基本的にキャッシュを提供しない
  - SPMD に基づく PGAS: 通信の集約化 → プログラマの責任
  - いくつか例外的にキャッシュを持つ PGAS も存在するが，**ノードをまたぐ動的負荷分散を対象としていない**
    - Chapel **[Ferguson and Buettner, PGAS '15]**，CLaMPI **[Girolamo+, IPDPS '17]**，GAM **[Cai+, VLDB '18]**
- **GET/PUT API** は本質的にキャッシュと相性が良くない
  - ローカルメモリをユーザ側で管理するため → 次スライド
- 本研究では新たに **Checkout/Checkin API** を導入
  - 処理系側がローカルメモリの管理を行うための API
  - → **ワークスティーリングに適したキャッシュ実装**が可能に

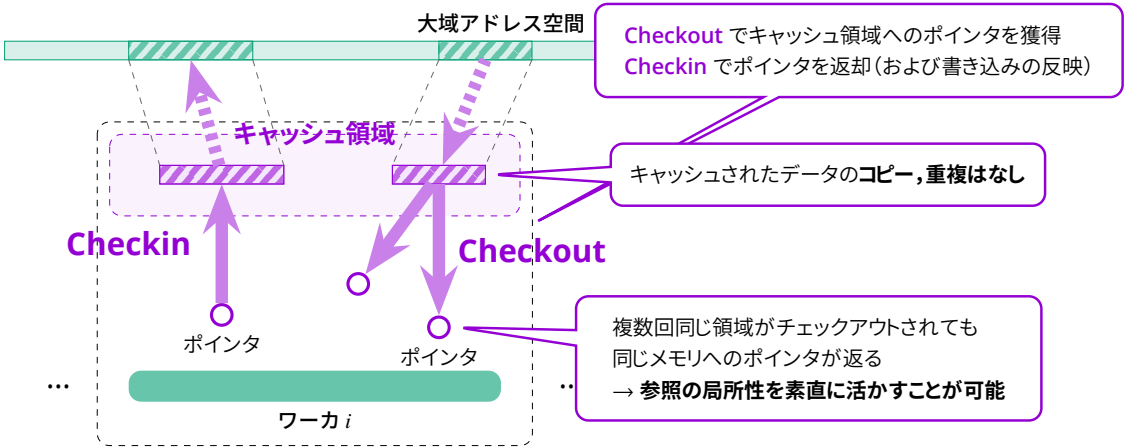
GET/PUT + キャッシュ の問題点

既存の PGAS ライブラリに素直にキャッシュを実装する場合



設計方針: 提案する Checkout/Checkin API

本研究では GET/PUT の問題点を踏まえ，新たに API を導入



Checkout/Checkin API の詳細

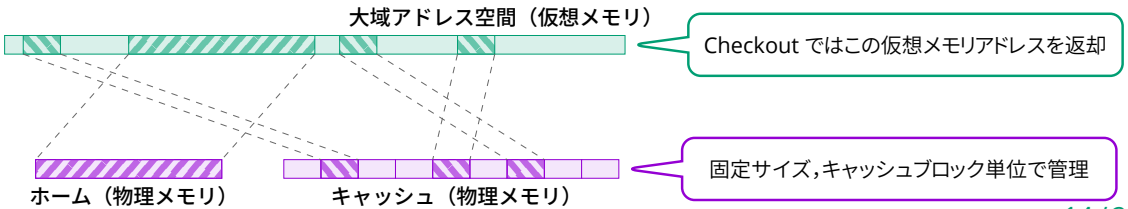
```
void mergesort(Span a) {
  if (a.size() < cutoff) {
    auto a_ = checkout(a, mode::
         ReadWrite);
    quicksort_serial(a_);
    checkin(a_);
  } else {
    ... // same as before
  }
}
```

タスク並列プログラムでは主に
カットオフ後の逐次実行で呼ばれる

- **checkout:** 指定された領域のキャッシュへのポインタを獲得
  - グローバルポインタ → ローカルポインタ への変換
  - アクセスモード（read/write/read+write）を同時に指定
  - 領域が既にキャッシュされていれば通信を省略可
- **checkin:** チェックアウトした領域を処理系に返却
  - read-only でなければ領域全体に書き込みがあったと見なす
- チェックアウトされている領域はキャッシュから追い出されない
- ある領域がチェックアウトされている間に Fork/Join を呼んではいけない
- （CRL [Johnson+, SOSP '95] と似た API だが，本提案は大規模な配列を扱うことを想定した設計）

ソフトウェアキャッシュの実装

- 各ワーカで独立，キャッシュサイズは固定，**キャッシュブロック**単位で管理
- **キャッシュブロック:** キャッシュが管理される単位（実験では 64 KB = A64FX のページサイズ）
- **ホーム:** グローバルメモリのある領域が割り当てられたノード．malloc 時に決定される
  - 現状ではブロック分割のみ実装
- 固定サイズの物理メモリを `shm_open()` 等で確保しておき，`mmap()` で仮想メモリを動的に張り替え
  - オーバーラップした領域がチェックアウトされることもある
    → 連続な仮想メモリアドレスがあることが望ましい



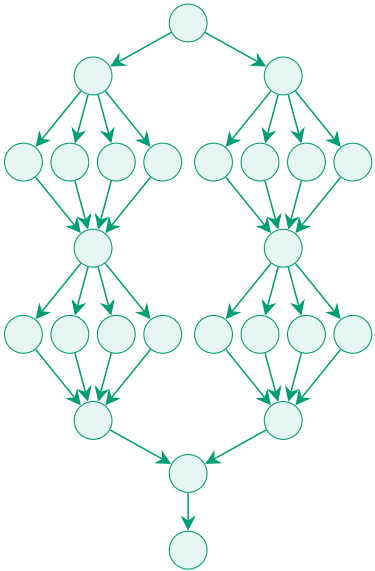
現状のコヒーレントキャッシュの実装

各 API におけるキャッシュに対する操作（**コヒーレンスプロトコル**）:

- **Checkout:** キャッシュされていない領域を**キャッシュブロックの粒度で読み込み**
  - **Read（読み込み）**操作に相当
  - キャッシュが満杯の場合はランダムにキャッシュブロックを追い出す ← 将来的には LRU?
- **Checkin:**（read-only でない場合）チェックインされた領域のみ**ホームに書き込み**
  - **Write（書き込み）**操作に相当
  - 現状はシンプルな**ライトスルー型**（毎回の write でホームに書き込み）
  - （キャッシュブロック単位ではなくバイト単位で書き込む必要あり）

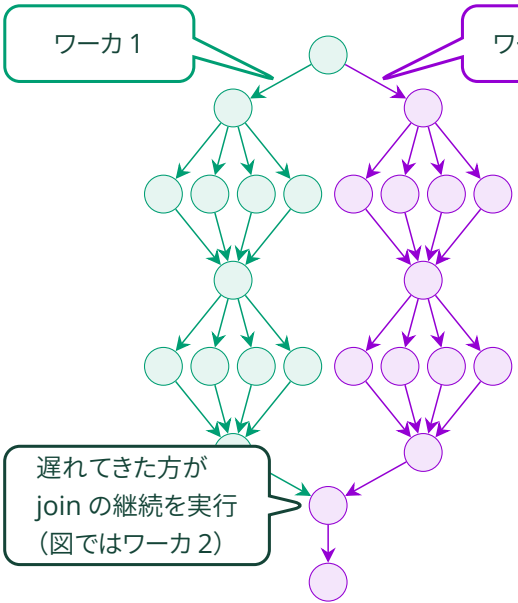
他のワーカによって書き込まれた新しい値を正しく読み込むため，
どこかのタイミングでキャッシュを無効にする必要がある
→ 正しく**キャッシュコヒーレンス**を保つには？

ワークスティーリングにおけるキャッシュコヒーレンスの管理



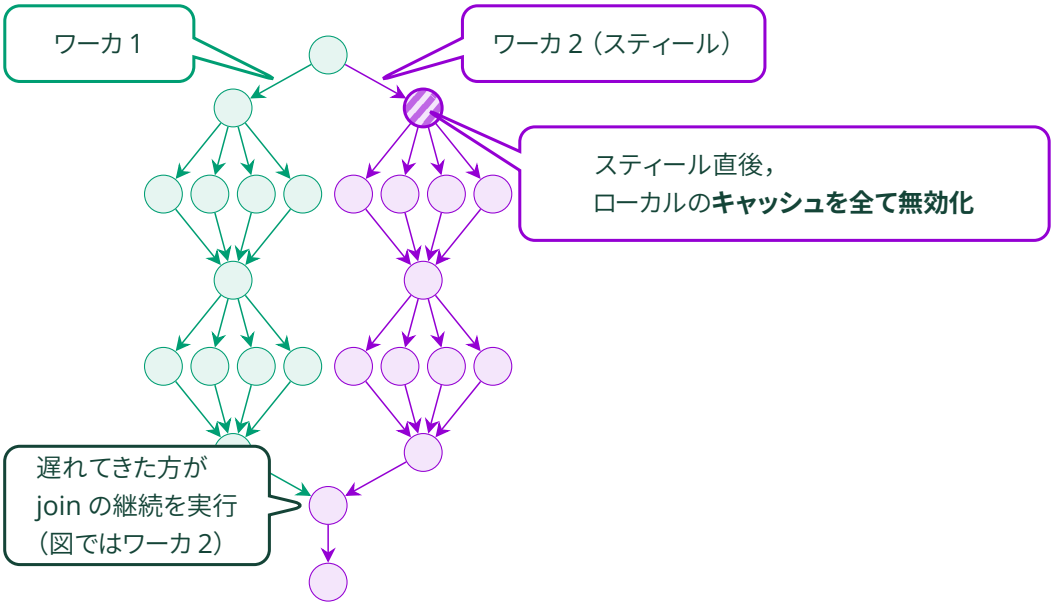
- 依存関係のあるタスク間の読み書き順序を保証
  - DAG consistency [Blumofe+, SPAA '98]
  - プログラムは**データレースフリー**であると仮定
- 適切なタイミングでキャッシュを無効化し，最新の値をフェッチする必要がある
- 問題設定:
  - タスクを実行するワーカは動的に決定される
  - ワーカはそれぞれ独立のキャッシュを持つ
  - 書き込みは checkin でホームに反映される

ワークスティーリングにおけるキャッシュコヒーレンスの管理

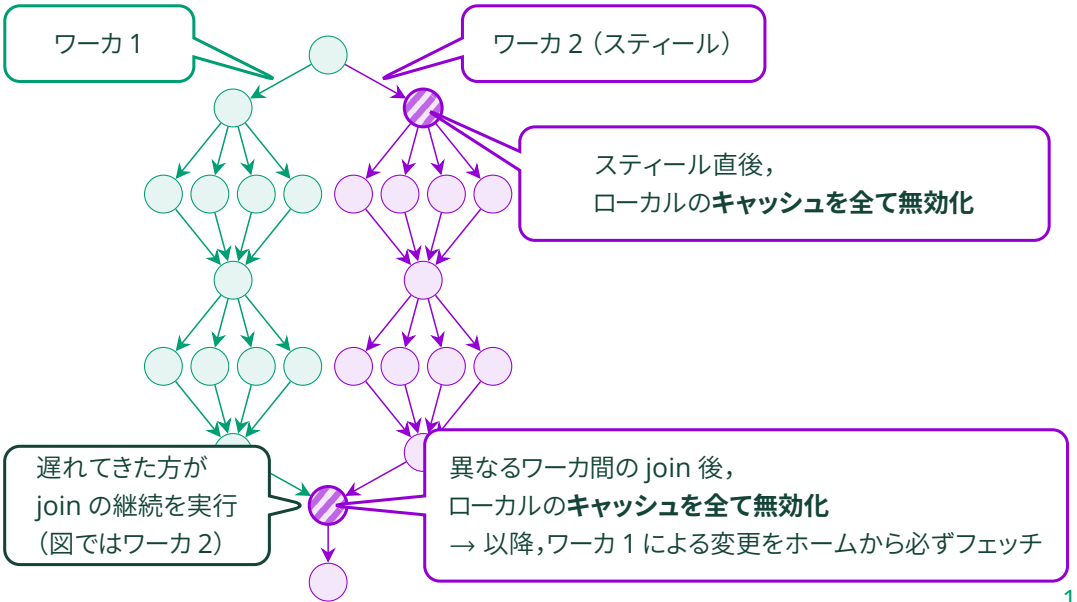


- 依存関係のあるタスク間の読み書き順序を保証
  - DAG consistency [Blumofe+, SPAA '98]
  - プログラムは**データレースフリー**であると仮定
- 適切なタイミングでキャッシュを無効化し，
  最新の値をフェッチする必要がある
- 問題設定:
  - タスクを実行するワーカは動的に決定される
  - ワーカはそれぞれ独立のキャッシュを持つ
  - 書き込みは checkin でホームに反映される

ワークスティーリングにおけるキャッシュコヒーレンスの管理



ワークスティーリングにおけるキャッシュコヒーレンスの管理



Outline

背景

提案する PGAS 処理系の設計・実装

実験評価

関連研究

結論・今後の課題

実験評価

まず分散ワークスティーリング単体の性能を評価し、次に PGAS と組み合わせた場合の性能を見る

ソフトウェア:

- 分散ワークスティーリング: **MassiveThreads/DM** [Akiyama and Taura, HPDC '15] を A64FX 上で MPI-3 RMA を用いて動作するよう改善したものを使用
  - （その他の性能改善・評価については Cluster '22 で発表予定）
- PGAS: MPI-3 RMA を用い C++ ライブラリとして実装（非常にシンプルなコヒーレントキャッシュ実装）

実験環境:

- 東大のスパコン **Wisteria/BDEC-01 Odyssey**（最大 2,304 ノード）
- CPU: 富士通 A64FX (48 コア/ノード)
- 通信インターコネクト: Tofu-D
- MPI: 富士通 MPI 4.0.1
- コンパイラ: 富士通 コンパイラ 4.8.0（`-O3 -Nclang`）

分散ワークスティーリングの性能

- **Unbalanced Tree Search (UTS)** ベンチマークを用いて評価
  - 動的負荷分散の性能評価でよく用いられる単純なベンチマーク．PGAS を用いる必要はない．
- **X 軸:** コア数（最大 110,592 コア; 2,304 ノード）
- **Y 軸:** スループット（1 秒間に数えたノード数）
- MassiveThreads/DM は **110,592 コアで 96.4% の並列化効率**を達成

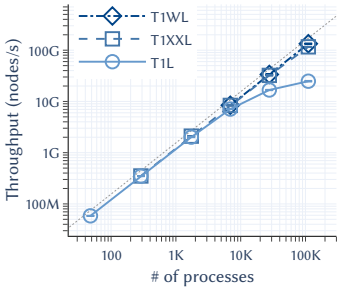


Fig. MassiveThreads/DM

実験設定:

- UTS: バランスの悪い木の全ノード数を並列にカウント
- 木のサイズ: T1L < T1XXL < T1WL
- 1 プロセス/コア

分散ワークスティーリング単体の性能は良好
→ PGAS を組み合わせた場合の性能は?

提案 PGAS 処理系上での並列ソート（Cilksort）の評価

- キャッシュを実装 → **最大で 2.4 倍**の性能向上
  - 27,648 コアでは 1.3 倍 → ノード数が多い場合キャッシュの効果は小さい?
- 少なくとも**ノード数を増やせば性能は向上**
- **台数効果（27,648 コア; 576 ノード）:** 542 倍，`std::sort()` と比較しても 412 倍
  - 並列化効率はあまり良くないが，現在のキャッシュ実装には大いに改善の余地あり

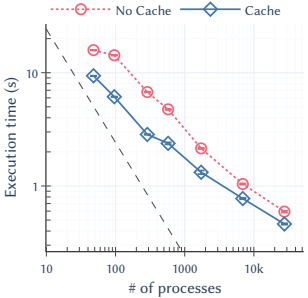


Fig. 実行時間

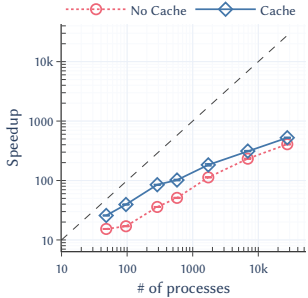


Fig. 台数効果

実験設定:

- 10 億要素（32 bit float; 一様乱数）
- 逐次実行へのカットオフ: 16K 要素 （64 KB）
- **No Cache:** キャッシュを無効化した PGAS
- **Cache:** キャッシュあり（32 MB/コア）
- 点線の直線: 理想的な性能
- 理想的な実行時間 = 逐次実行時間 / コア数

Outline

背景

提案する PGAS 処理系の設計・実装

実験評価

関連研究

結論・今後の課題

関連研究

- **ソフトウェア分散共有メモリ**（Software Distributed Shared Memory; DSM）
  - グローバルポインタとローカルの生ポインタが区別されない（PGAS では両者を明確に区別）
    - メモリ保護を活用した仮想メモリ → ほとんどの処理系がコヒーレントキャッシュを持つ
  - 1990 年代に盛んに研究 → 2000 年代には下火に
    - TreadMarks **[Keleher+, WTEC '94]**，Midway **[Bershad+, '93]**，DAG-consistency **[Blumofe+, SPAA '96]** など
  - 近年の DSM 処理系（通信インターコネクトの相対的な性能向上に着目）
    - Argo **[Kaxiras+, HPDC '15]**，Popcorn **[Chuang+, SYSTOR '20]**，MENPS **[Endo+, IPDRM '20]** など
  - **性能の出しやすさの観点から，本研究では PGAS 的なアプローチを採用**
- **タスクとデータを明示的に紐付ける**処理系
  - Tascell **[Hiraishi+, PPoPP '09]**，Legion **[Bauer+, SC '12]**，PaRSEC **[Bosilca+, '13]** など
  - タスクの移動とともに用いる入出力データをローカルにコピーして実行
  - **問題点:** 一部のデータしか使わない場合でも，タスクに紐付いたデータ**全体**を**実行前に**コピーする
    - 問題になる例: **巨大な配列に対する二分探索** → 実際は一部のデータしか使わない
    - 1 ノードのメモリより大きいデータを扱うことが難しいという問題も

Outline

背景

提案する PGAS 処理系の設計・実装

実験評価

関連研究

結論・今後の課題

まとめ

結論:

- シンプルなソフトウェアキャッシュを PGAS に実装，分散ワークスティーリングと協調して動作
- 並列ソート（Cilksort）のベンチマークを評価 → **27,648 コア（576 ノード）までのスケーラビリティ**
  - 並列化効率はあまり良くない（27,648 コアで 542 倍の台数効果） → さらなる性能改善が必要

今後の課題:

- コヒーレンスプロトコルの改善（ライトバック型など）
  - スティールされるワーカ（victim）に対し何らかの割り込みが必要
- プリフェッチの導入
- ノード内のマルチコアでキャッシュを共有 ← スケジューラの改善なしでは効果は小さい?
- メモリ階層を意識したスケジューラを実装し，局所性のさらなる改善
  - ADWS [Shiina and Taura, SC '19] 等の深いメモリ階層を想定したスケジューラが有望

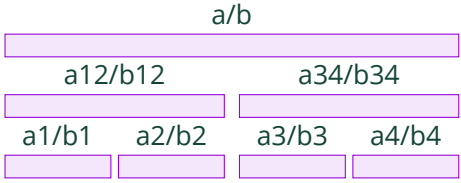
予備資料: Cilksort

```
void cilksort(Span a, Span b) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a12, a34] = a.divide_two();
    auto [b12, b34] = b.divide_two();
    auto [a1, a2] = a12.divide_two();
    auto [a3, a4] = a34.divide_two();
    auto [b1, b2] = b12.divide_two();
    auto [b3, b4] = b34.divide_two();
    task_group tg;
    tg.fork([=] { cilksort(a1, b1); });
    tg.fork([=] { cilksort(a2, b2); });
    tg.fork([=] { cilksort(a3, b3); });
    tg.fork([=] { cilksort(a4, b4); });
    tg.join();
    tg.fork([=] { cilkmerge(a1, a2, b12); });
    tg.fork([=] { cilkmerge(a3, a4, b34); });
    tg.join();
    cilkmerge(b12, b34, a);
  }
}
```

- Cilk 言語の examples に収録
- マージソートを再帰的に行うような並列ソート
- コード例では配列の範囲（span）を用い記述
  - Span: ポインタと要素数の組
- 範囲 a をソート（b は一時的なバッファ）

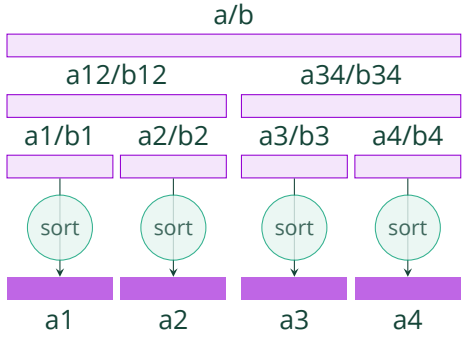
予備資料: Cilksort

```
void cilksort(Span a, Span b) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a12, a34] = a.divide_two();
    auto [b12, b34] = b.divide_two();
    auto [a1, a2] = a12.divide_two();
    auto [a3, a4] = a34.divide_two();
    auto [b1, b2] = b12.divide_two();
    auto [b3, b4] = b34.divide_two();
    task_group tg;
    tg.fork([=] { cilksort(a1, b1); });
    tg.fork([=] { cilksort(a2, b2); });
    tg.fork([=] { cilksort(a3, b3); });
    tg.fork([=] { cilksort(a4, b4); });
    tg.join();
    tg.fork([=] { cilkmerge(a1, a2, b12); });
    tg.fork([=] { cilkmerge(a3, a4, b34); });
    tg.join();
    cilkmerge(b12, b34, a);
  }
}
```



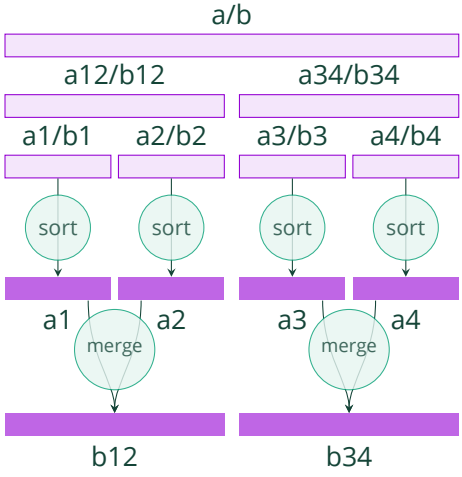
予備資料: Cilksort

```
void cilksort(Span a, Span b) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a12, a34] = a.divide_two();
    auto [b12, b34] = b.divide_two();
    auto [a1, a2] = a12.divide_two();
    auto [a3, a4] = a34.divide_two();
    auto [b1, b2] = b12.divide_two();
    auto [b3, b4] = b34.divide_two();
    task_group tg;
    tg.fork([=] { cilksort(a1, b1); });
    tg.fork([=] { cilksort(a2, b2); });
    tg.fork([=] { cilksort(a3, b3); });
    tg.fork([=] { cilksort(a4, b4); });
    tg.join();
    tg.fork([=] { cilkmerge(a1, a2, b12); });
    tg.fork([=] { cilkmerge(a3, a4, b34); });
    tg.join();
    cilkmerge(b12, b34, a);
  }
}
```



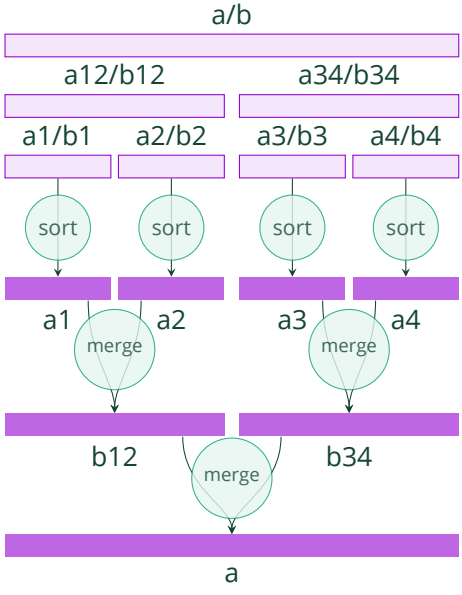
予備資料: Cilksort

```
void cilksort(Span a, Span b) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a12, a34] = a.divide_two();
    auto [b12, b34] = b.divide_two();
    auto [a1, a2] = a12.divide_two();
    auto [a3, a4] = a34.divide_two();
    auto [b1, b2] = b12.divide_two();
    auto [b3, b4] = b34.divide_two();
    task_group tg;
    tg.fork([=] { cilksort(a1, b1); });
    tg.fork([=] { cilksort(a2, b2); });
    tg.fork([=] { cilksort(a3, b3); });
    tg.fork([=] { cilksort(a4, b4); });
    tg.join();
    tg.fork([=] { cilkmerge(a1, a2, b12); });
    tg.fork([=] { cilkmerge(a3, a4, b34); });
    tg.join();
    cilkmerge(b12, b34, a);
  }
}
```



予備資料: Cilksort

```
void cilksort(Span a, Span b) {
  if (a.size() < cutoff) {
    quicksort_serial(a);
  } else {
    auto [a12, a34] = a.divide_two();
    auto [b12, b34] = b.divide_two();
    auto [a1, a2] = a12.divide_two();
    auto [a3, a4] = a34.divide_two();
    auto [b1, b2] = b12.divide_two();
    auto [b3, b4] = b34.divide_two();
    task_group tg;
    tg.fork([=] { cilksort(a1, b1); });
    tg.fork([=] { cilksort(a2, b2); });
    tg.fork([=] { cilksort(a3, b3); });
    tg.fork([=] { cilksort(a4, b4); });
    tg.join();
    tg.fork([=] { cilkmerge(a1, a2, b12); });
    tg.fork([=] { cilkmerge(a3, a4, b34); });
    tg.join();
    cilkmerge(b12, b34, a);
  }
}
```



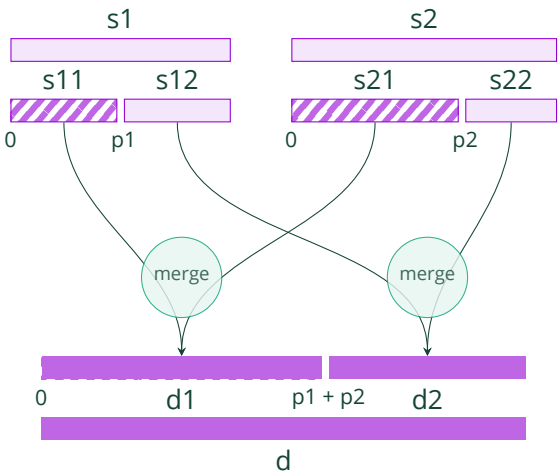
予備資料: Cilksort – マージ処理(Cilkmerge)

```
void cilkmerge(Span s1, Span s2, Span d) {
  if (s1.size() < cutoff) {
    merge_serial(s1, s2, d);
  } else {
    size_t p1 = (s1.size() + 1) / 2;
    size_t p2 = binary_search(s2, s1[p1 - 1]);

    auto [s11, s12] = s1.divide_at(p1);
    auto [s21, s22] = s2.divide_at(p2);
    auto [d1 , d2 ] = d.divide_at(p1 + p2);

    task_group tg:
    tg.fork([=] { cilkmerge(s11, s21, d1); });
    tg.fork([=] { cilkmerge(s12, s22, d2); });
    tg.join();
  }
}
```

マージ処理も再帰的に並列化可能



予備資料: Cilksort – マージ処理(Cilkmerge)

```
void cilkmerge(Span s1, Span s2, Span d) {
  if (s1.size() < cutoff) {
    merge_serial(s1, s2, d);
  } else {
    size_t p1 = (s1.size() + 1) / 2;
    size_t p2 = binary_search(s2, s1[p1 - 1]);

    auto [s11, s12] = s1.divide_at(p1);
    auto [
    auto [

    task_group tg:
    tg.fork([=] { cilkmerge(s11, s21, d1); });
    tg.fork([=] { cilkmerge(s12, s22, d2); });
    tg.join();
  }
}
```

配列の分割点(p2)を決めるため二分探索

マージ処理も再帰的に並列化可能