Almost Deterministic Work Stealing

xSIG 2019

椎名 峻平, 田浦 健次朗

東京大学 田浦研究室

2019.5.27

研究の目的

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
2. 処理系による**動的な負荷分散**
3. メモリアクセスの**局所性**の良いスケジューリング

研究の目的

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
2. 処理系による**動的な負荷分散**
3. メモリアクセスの**局所性**の良いスケジューリング

これらを全て満たす処理系の提案

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

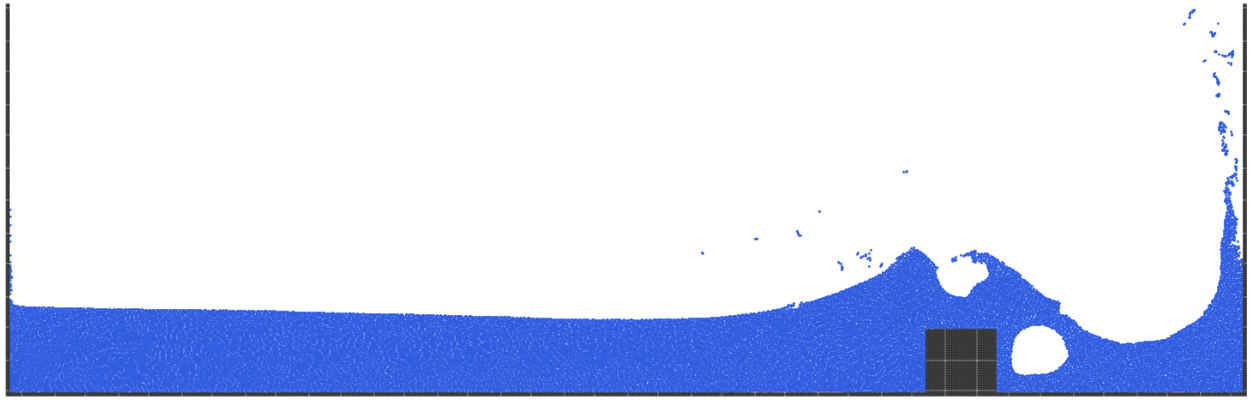
結論

Outline

Outline

（例）粒子法によるシミュレーション

例えば, 以下の計算を並列化したいとする

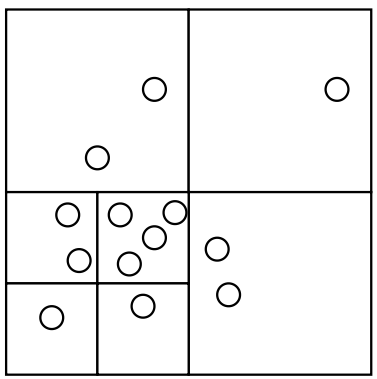


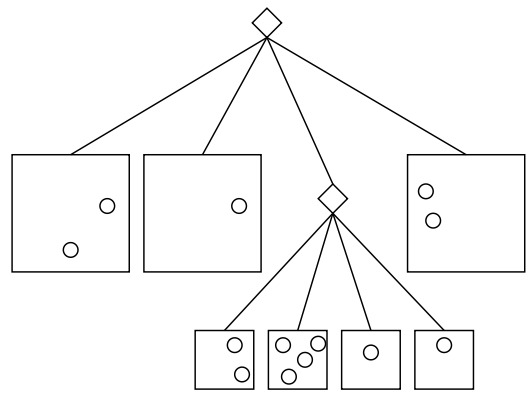
- 例: **Smoothed Particle Hydrodynamics (SPH 法)**
- 流体を粒子の集まりとしてモデル化しシミュレーションする手法
- 近傍粒子のみの相互作用（**短距離力**）のみを計算
- 図は 2D のダム崩壊の例

粒子間相互作用の計算

FDPS[1]（粒子計算フレームワーク）の実装では

- 領域を一定以下の粒子数になるまで再帰的に分割（左下）
- 四分木（3D なら八分木）の形で粒子を管理（右下）





1 M. Iwasawa, A. Tanikawa, N. Hosono, et al., “Implementation and performance of FDPS: A framework for developing parallel particle simulation codes,” Publications of the Astronomical Society of Japan, vol. 68, no. 4, 2016.

再帰を用いた粒子間相互作用の計算

逐次処理であれば以下のように書くのが自然

Traverse the tree

```
particle_interaction(node) {
  if (node is leaf) {
    /* node (leaf) 内の粒子間相互作用の計算 */
  } else {
    // 木を再帰的にたどっていく
    for (child in node.children) {
      particle_interaction(child);
    }
  }
}
```

FDPS における粒子間相互作用の並列化

手順:

1. 四分木を再帰的に辿り, リーフノードを配列に並べ直す
2. ループによる並列化

OpenMP によるループによる並列化

```
#pragma omp parallel for schedule(dynamic, 4)
for (leaf in leaf_list) {
  /* leaf 内の粒子間相互作用の計算 */
}
```

逐次の場合と比べて手間が増える(配列の確保など)

タスク並列モデルによる粒子間相互作用の並列化

一種の分割統治法として記述する例

```
particle_interaction(node) {
  if (node is leaf) {
    /* node (leaf) 内の粒子間相互作用の計算 */
  } else {
    task_group tg;
    for (child in node.children) {
      // 子のノードをタスクとして生成（fork）
      tg.run([=]{ particle_interaction(child); });
    }
    tg.wait(); // タスクの終了を待つ（join）
  }
}
```

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

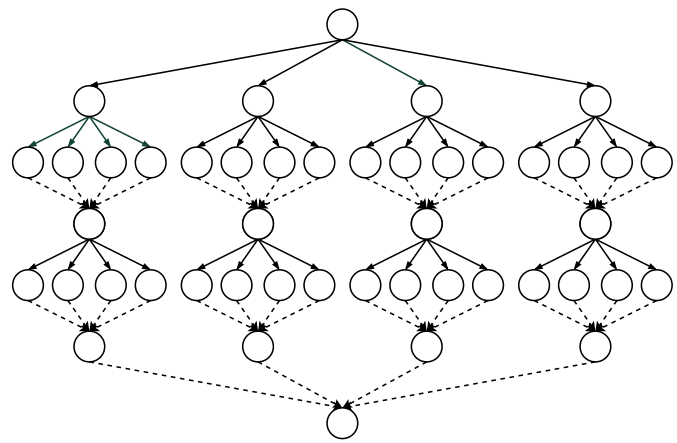
関連研究

結論

タスク並列モデル

- タスク間の依存関係を元に並列実行を行うモデル
- **分割統治法**で書かれたプログラムを簡単に並列化可能
- 左下：タスクを 4 つ並列実行する例（TBB[2]ライクな記法）
- 右下：実行は **Directed Acyclic Graph (DAG)** の形で表現される

```
task_group tg;
tg.run([]{ ... });
tg.run([]{ ... });
tg.run([]{ ... });
tg.run([]{ ... });
tg.wait();
```



2 J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism. ” O’Reilly Media, Inc.”, 2007.

タスク並列の利点

- プログラムの任意の時点でタスクを生成可能

タスク並列の利点

- プログラムの任意の時点でタスクを生成可能
- プロセッサの数以上のタスクを生成可能

タスク並列の利点

- プログラムの任意の時点でタスクを生成可能
- プロセッサの数以上のタスクを生成可能
- 再帰や分割統治で書かれたプログラムを簡単に並列化可能

タスク並列の利点

- プログラムの任意の時点でタスクを生成可能
- プロセッサの数以上のタスクを生成可能
- 再帰や分割統治で書かれたプログラムを簡単に並列化可能

効率的なタスク並列処理系の実装が必要

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

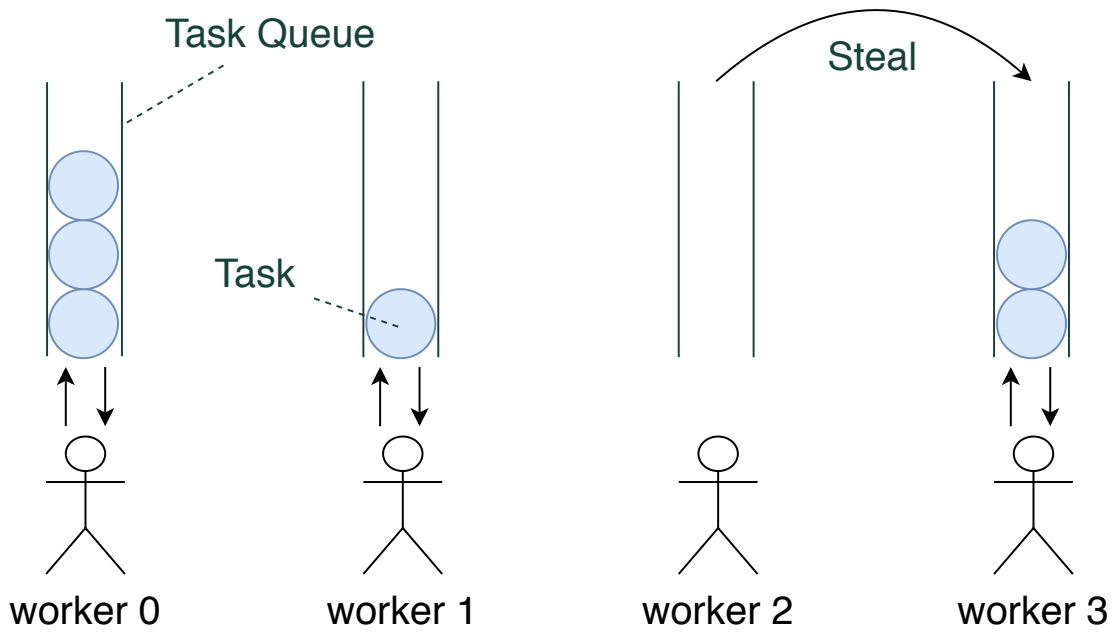
性能評価

関連研究

結論

Work Stealing[3]

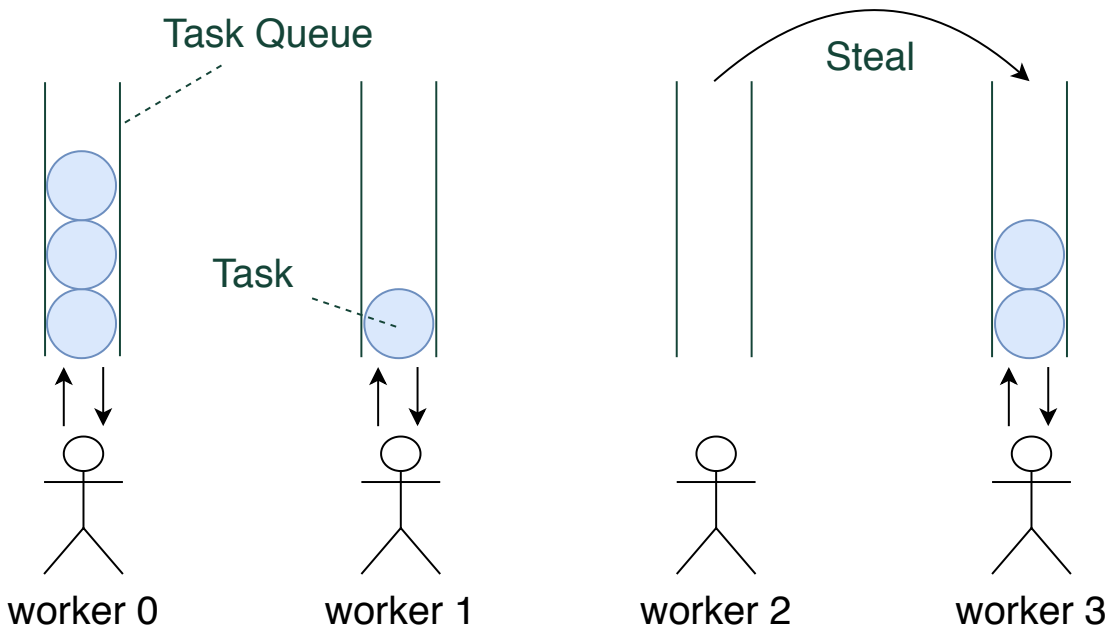
- よく使われるタスク並列スケジューリング手法



3 R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” J. ACM, vol. 46, no. 5, pp. 720–748, 1999.

Work Stealing[3]

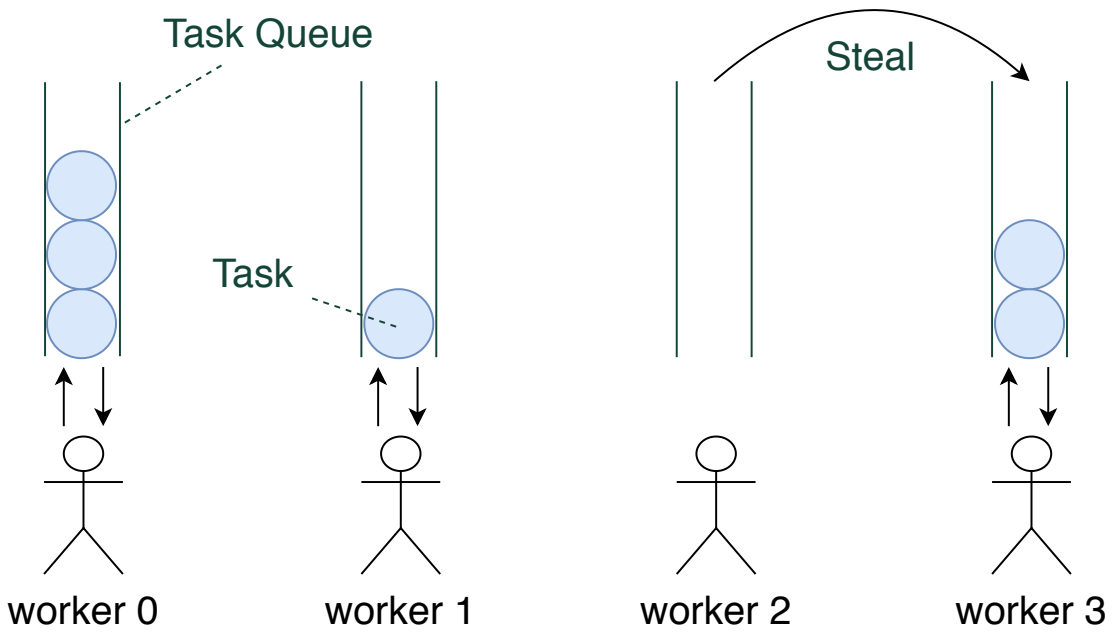
- よく使われるタスク並列スケジューリング手法
- 各 worker がそれぞれタスクキューを持つ



3 R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” J. ACM, vol. 46, no. 5, pp. 720–748, 1999.

Work Stealing[3]

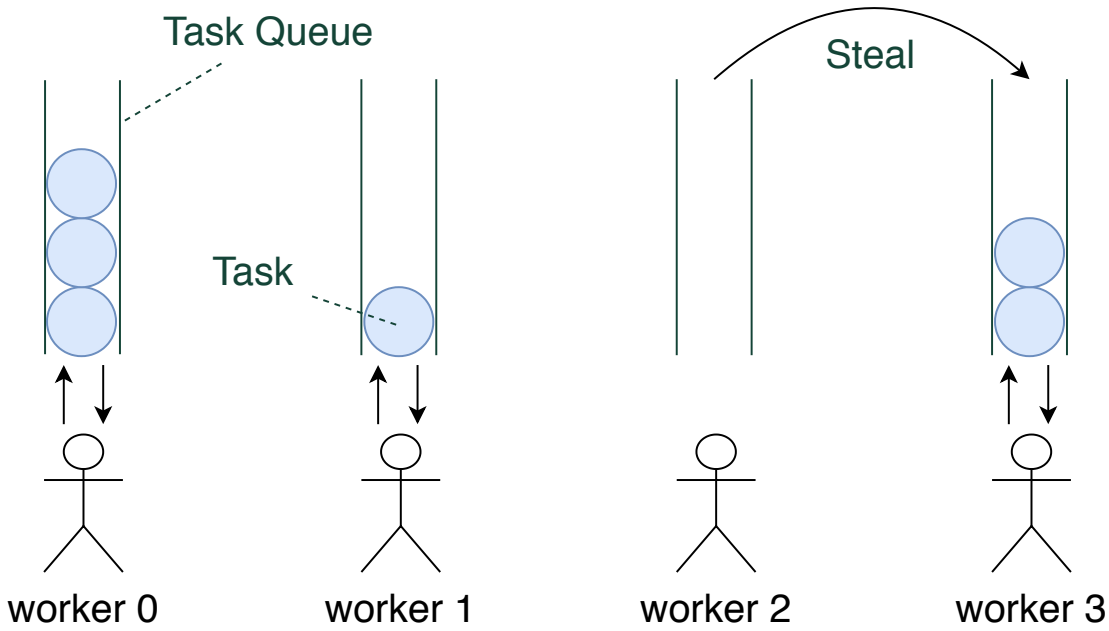
- よく使われるタスク並列スケジューリング手法
- 各 worker がそれぞれタスクキューを持つ
- 各 worker は自身のタスクキューからタスクを取り出し, 実行する



3 R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” J. ACM, vol. 46, no. 5, pp. 720–748, 1999.

Work Stealing[3]

- よく使われるタスク並列スケジューリング手法
- 各 worker がそれぞれタスクキューを持つ
- 各 worker は自身のタスクキューからタスクを取り出し, 実行する
- 自身のタスクキューが空になった worker は
  ランダムに worker を選び, タスクを盗む（**steal**）



3 R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” J. ACM, vol. 46, no. 5, pp. 720–748, 1999.

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

結論

タスク並列におけるメモリアクセスの局所性

- DAG において近いタスクは近いメモリを触る傾向にある（**メモリアクセスの局所性**）
- できるだけ飛び飛びの位置を実行しないようにしたい
- NUMA では局所性の問題が深刻
  - worker {0, 1}, {2, 3}が同一の socket に属していたら?

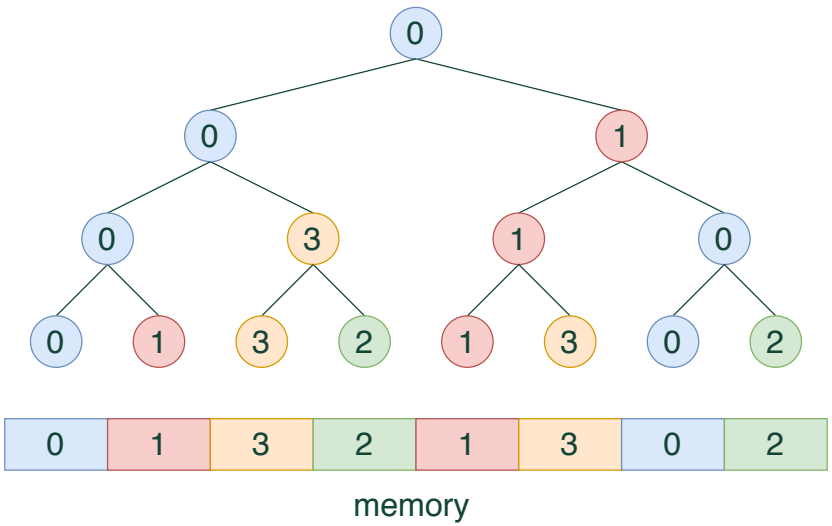


Fig. 局所性が悪い例

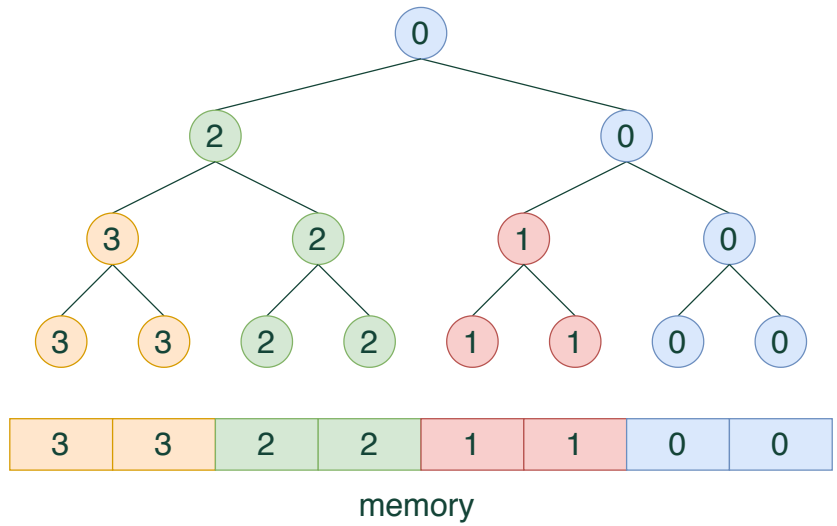
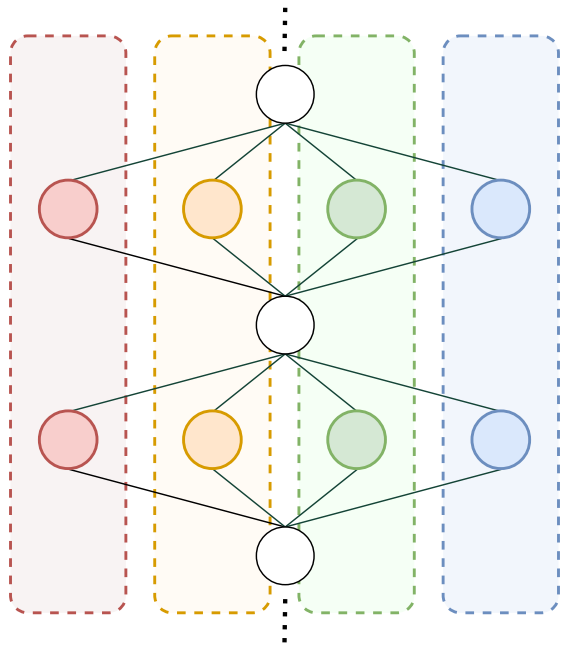


Fig. 局所性が良い例

iterative なプログラムにおける局所性

- **iterative なプログラム**: 似た計算を繰り返し行うプログラム
- iteration 毎にだいたい同じ DAG の形状になる
- iteration 毎に DAG の同じ位置のタスクを実行すると局所性が良い



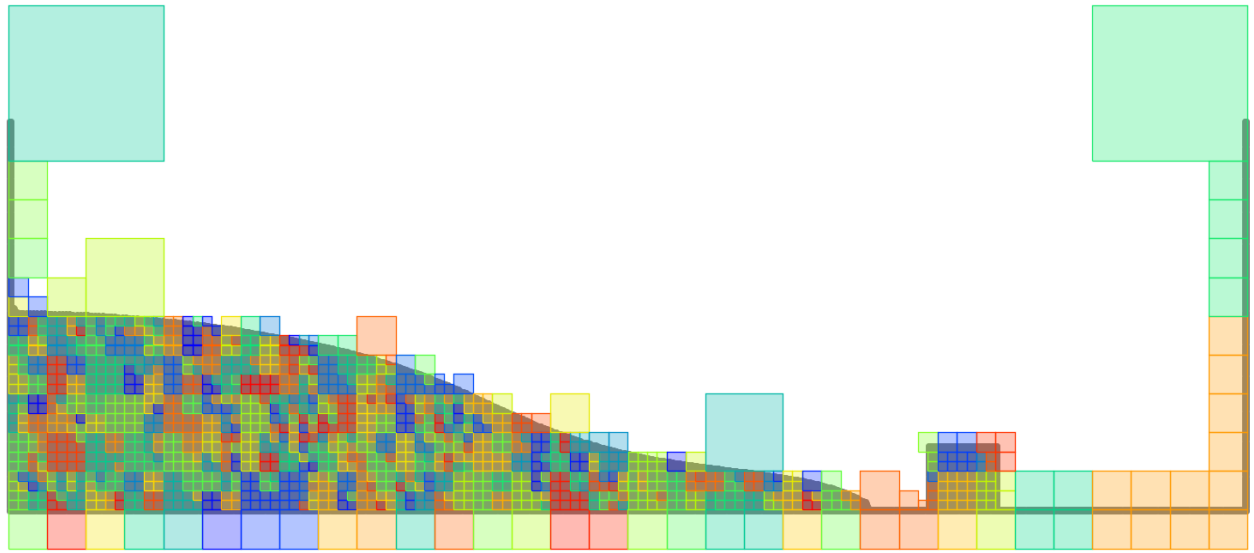
これまでの Work Stealing の局所性

- ランダムに worker を選択して steal を行うため,
  メモリアクセスの**局所性**を損なう場合がある

これまでの Work Stealing の局所性

- ランダムに worker を選択して steal を行うため,
  メモリアクセスの**局所性**を損なう場合がある
- 似た計算を繰り返し行う **iterative なプログラム**では,
  iteration ごとに負荷分散の様子が異なり
  キャッシュの再利用ができない

(参考)実際の負荷分散の様子



- **近い worker が近い領域を計算しない**
  - メモリ階層(NUMA 等)により worker 間の通信コストは異なる
- **iteration 毎に全く異なる負荷分散になる**
  - L1, L2 などのキャッシュが再利用できない

研究の目的（再掲）

我々の求めるもの：

研究の目的(再掲)

我々の求めるもの:

1. **分割統治法**で書かれたプログラムを簡単に並列化できる

研究の目的(再掲)

我々の求めるもの:

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
    - タスク並列モデルが有望

研究の目的(再掲)

我々の求めるもの:

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - タスク並列モデルが有望
2. 処理系による**動的な負荷分散**

研究の目的(再掲)

我々の求めるもの:

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
    - タスク並列モデルが有望
2. 処理系による**動的な負荷分散**
    - Work Stealing

研究の目的（再掲）

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - タスク並列モデルが有望
2. 処理系による**動的な負荷分散**
   - Work Stealing
3. メモリアクセスの**局所性**の良いスケジューリング

研究の目的（再掲）

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - タスク並列モデルが有望
2. 処理系による**動的な負荷分散**
   - Work Stealing
3. メモリアクセスの**局所性**の良いスケジューリング
   - しかし Work Stealing は局所性が悪い

研究の目的(再掲)

我々の求めるもの:

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - タスク並列モデルが有望
2. 処理系による**動的な負荷分散**
   - Work Stealing
3. メモリアクセスの**局所性**の良いスケジューリング
   - しかし Work Stealing は局所性が悪い

1. と 2. の要素を損なわず, 3. の改善を目指す

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

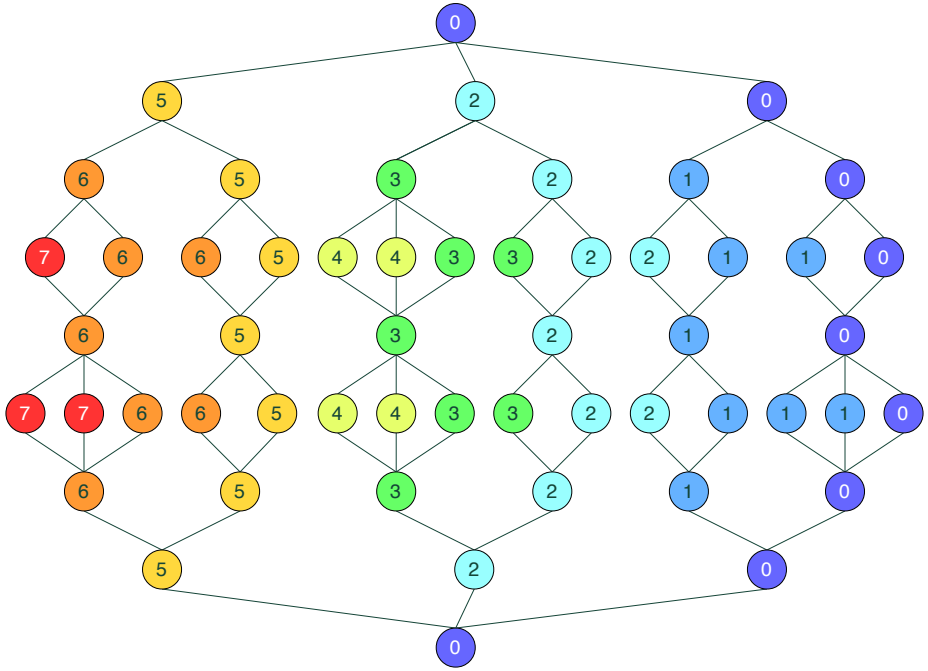
- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

結論

Almost Deterministic Work Stealing (ADWS)



- 各 worker の仕事量がほぼ等しくなるように（**負荷分散**）
- ある worker, ある socket 内の worker が近い位置のタスクを実行するように（**メモリアクセスの局所性**）
- ほぼ決定的にスケジューリングを行う
  （**iterative なプログラムにおける局所性**）

プログラマへの要求

ADWS の制約として,

- プログラマは明示的に**各タスクの仕事量**を与える必要がある
  - 絶対値でなく相対値で良い（`w_all` と `w_1`, ..., `w_4` の比）
  - アプリケーション固有の情報であり, ハードウェア依存ではない

```
task_group tg(w_all);
tg.run([]{ ... }, w_1);
tg.run([]{ ... }, w_2);
tg.run([]{ ... }, w_3);
tg.run([]{ ... }, w_4);
tg.wait();
```

例: 粒子間相互作用の仕事量

粒子数を仕事量として与える例

```
particle_interaction(node) {
  if (node is leaf) {
    /* node (leaf) 内の粒子間相互作用の計算 */
  } else {
    task_group tg(node.n_particles);
    for (child in node.children) {
      tg.run([=]{ particle_interaction(child); },
             child.n_particles);
    }
    tg.wait();
  }
}
```

ADWS の 2 つの要素

- **Deterministic Task Allocation**
  - 各タスクの仕事量を元に**決定的**にタスクを振り分ける
  - **分割統治法**に対する静的負荷分散手法とも言える
- **Hierarchical Localized Work Stealing**
  - 実行中に生じた実行時間のばらつきを動的に埋め合わせる
  - 局所性に配慮した **Work Stealing** に基づくアルゴリズム

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

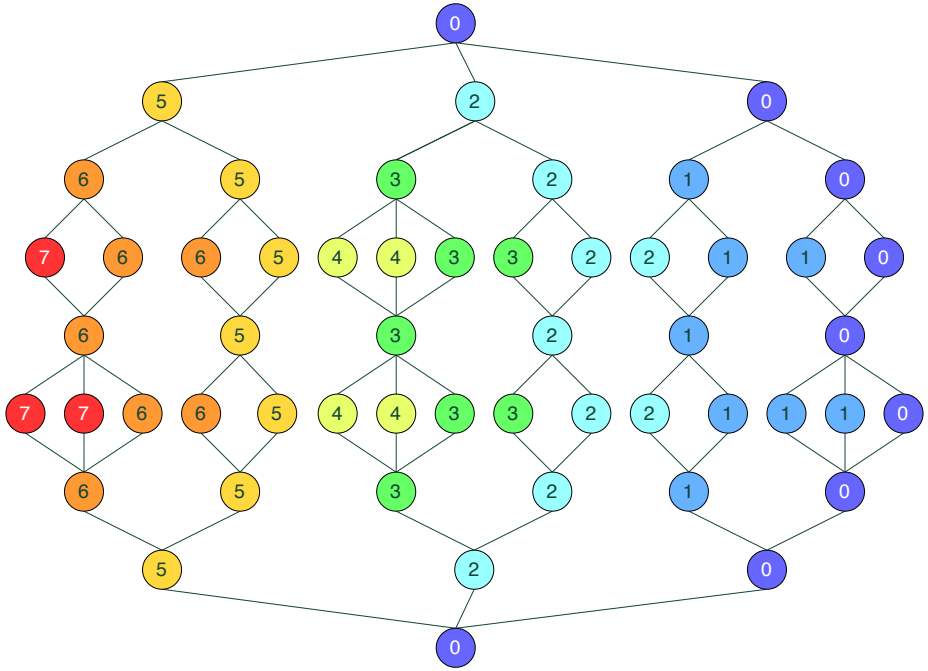
- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

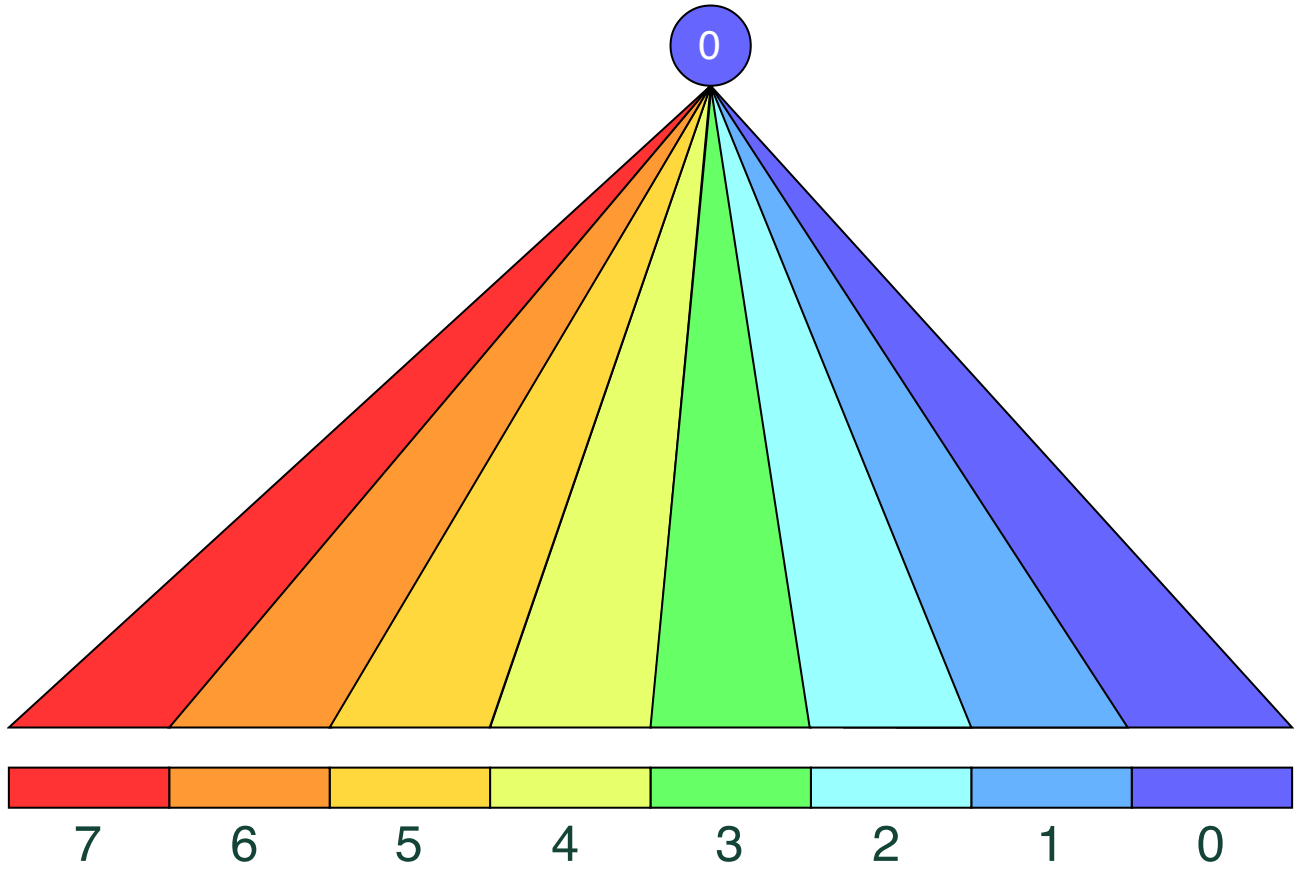
結論

Deterministic Task Allocation



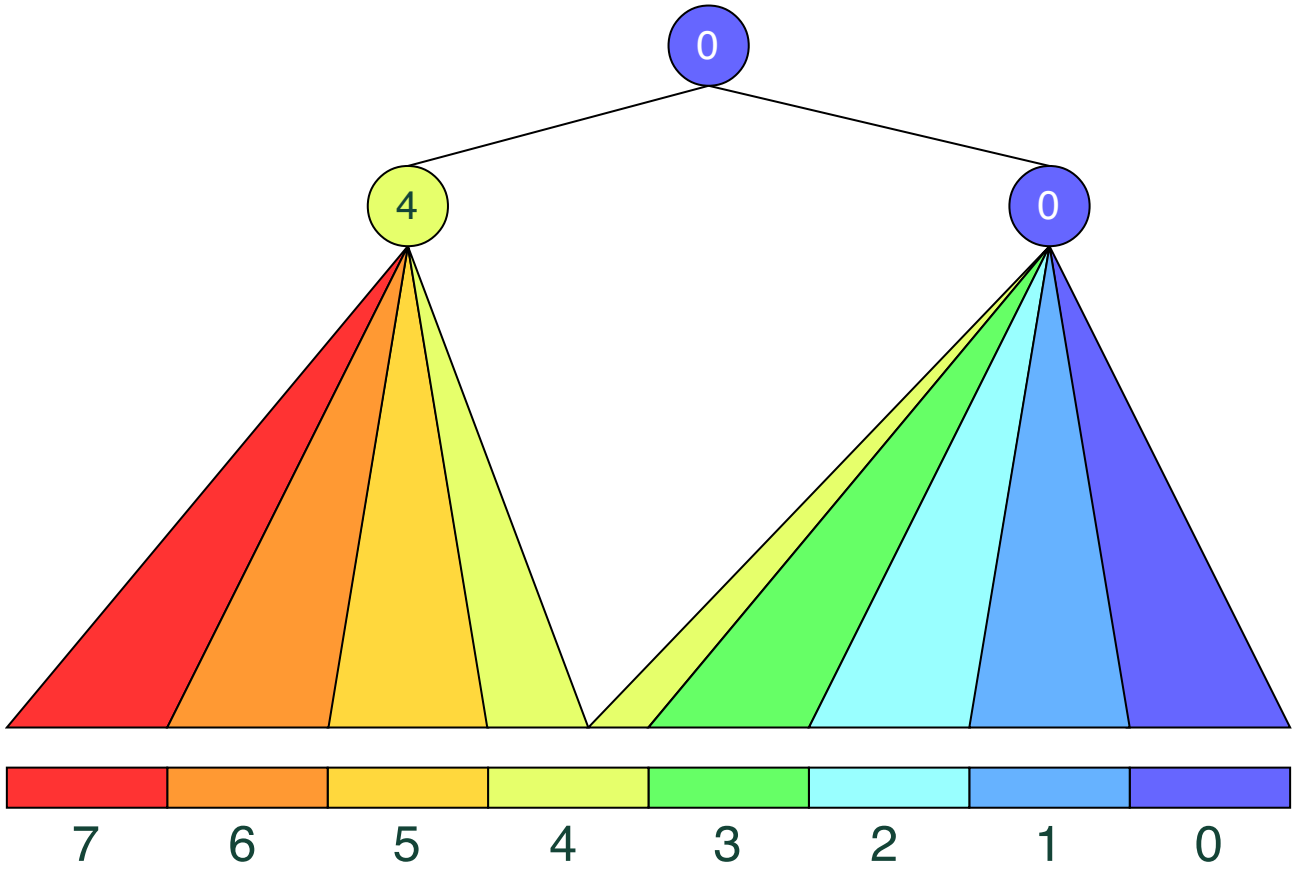
- プログラマから与えられた各タスクの仕事量を元に
  決定的なタスクの配置を行う
- 右から左に順番に worker が並ぶようにする
  - こう並べると嬉しいことがある
- 近い位置の worker が近い位置のタスクを実行
  - 近い位置の worker が近い番号を振られているという仮定

負荷分散アルゴリズムの概略 (1/4)



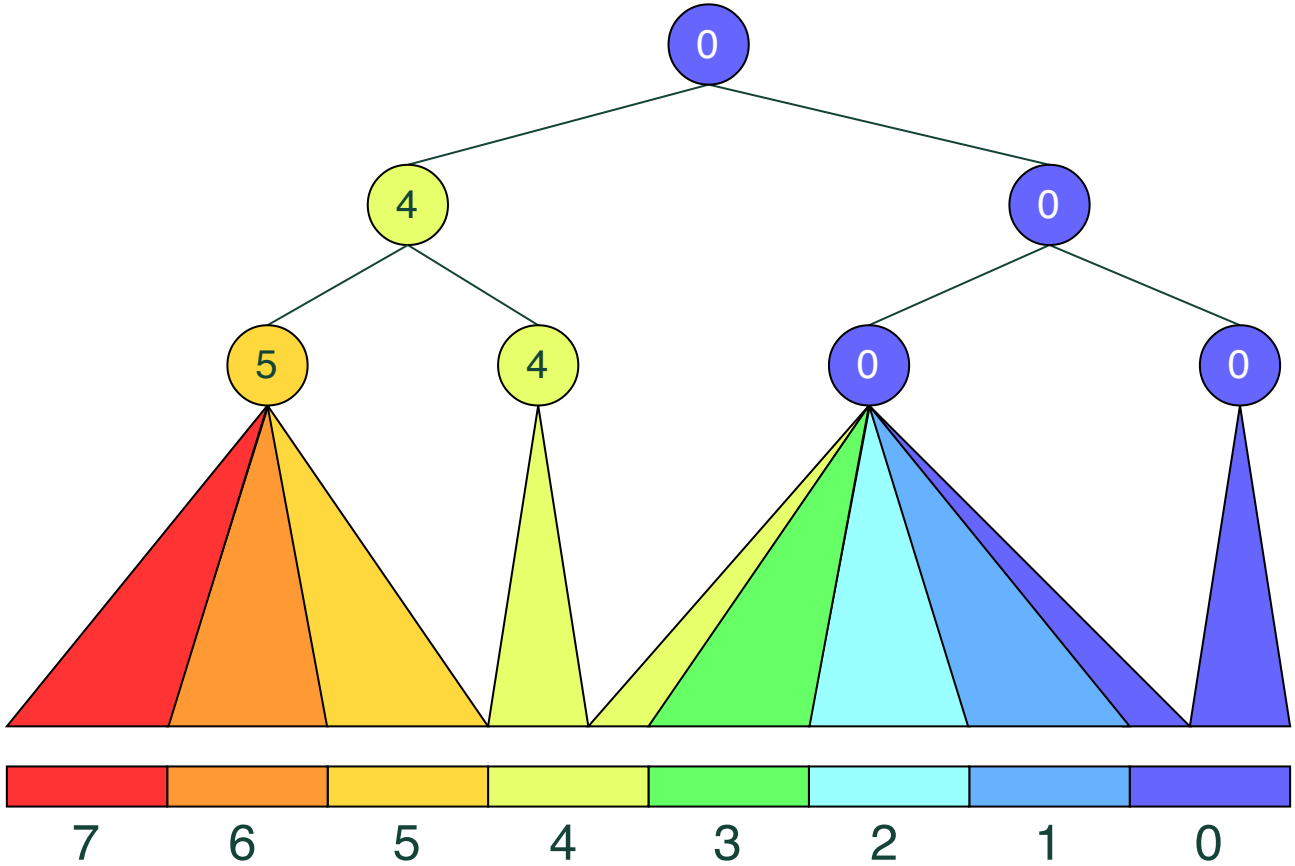
- 8 つの worker で分割を行う例
- 下の四角形が worker の範囲を表す
- 初期状態では 1 つのタスクのみが存在

負荷分散アルゴリズムの概略 (2/4)



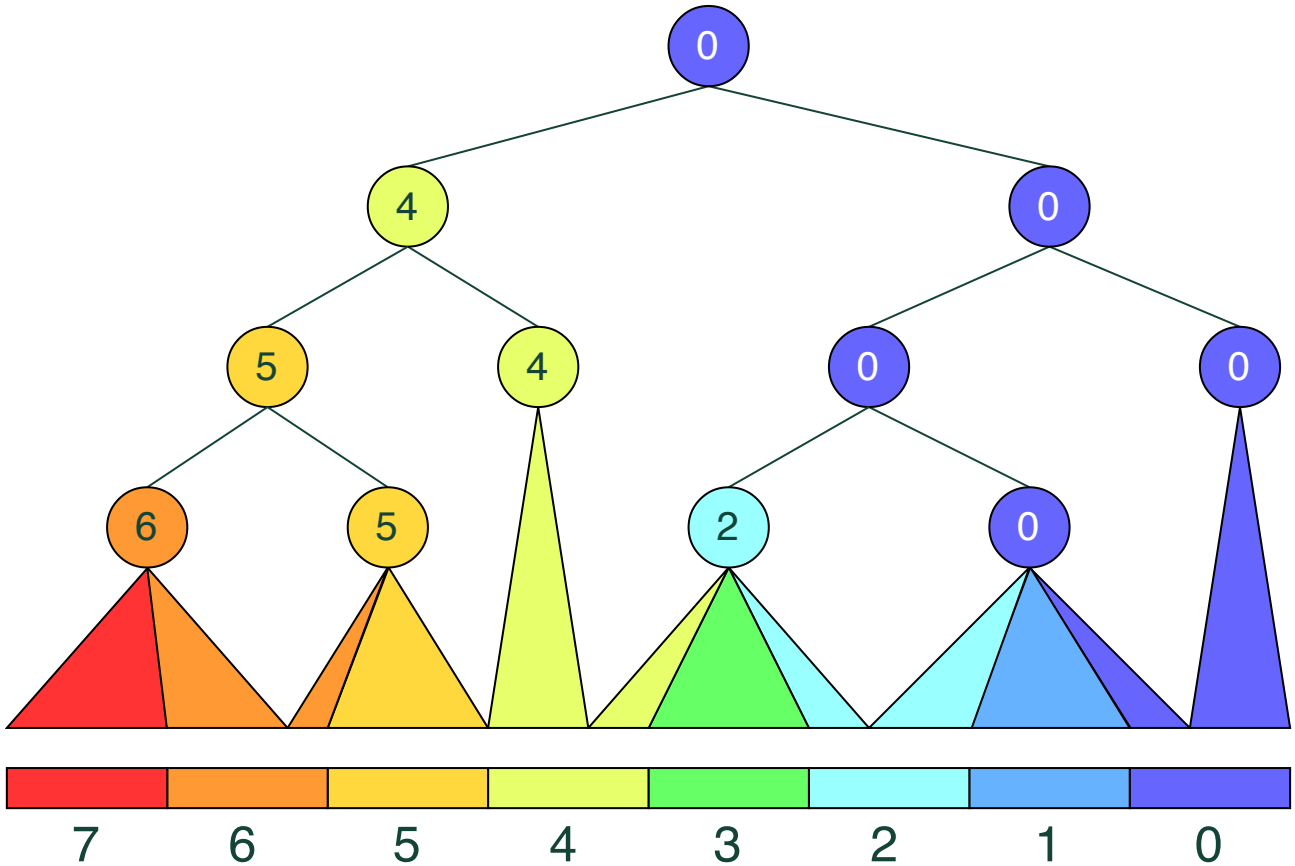
- 新しくタスクが生成された場合
- 左が生成されたタスク, 右は継続
- 生成されたタスクの仕事量を元に worker の範囲を分割

負荷分散アルゴリズムの概略 (3/4)



- タスクが生成される度に再帰的に分割を繰り返していく
- 複数の worker を持つタスクは,
  その中で最も番号の小さい worker が担当

負荷分散アルゴリズムの概略 (4/4)



- 実際にタスクを実行する過程で負荷分散を行っていく
- 並列にタスク配置を決定するため, オーバーヘッドが少ない

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

結論

Hierarchical Localized Work Stealing

- **Deterministic Task Allocation** では
  プログラマ指定の仕事量を元に負荷分散

Hierarchical Localized Work Stealing

- **Deterministic Task Allocation** では
  プログラマ指定の仕事量を元に負荷分散
- **大雑把な仕事量**の指定でも上手くやってほしい
  - e.g. 仕事量は必ずしも粒子数に比例しない

Hierarchical Localized Work Stealing

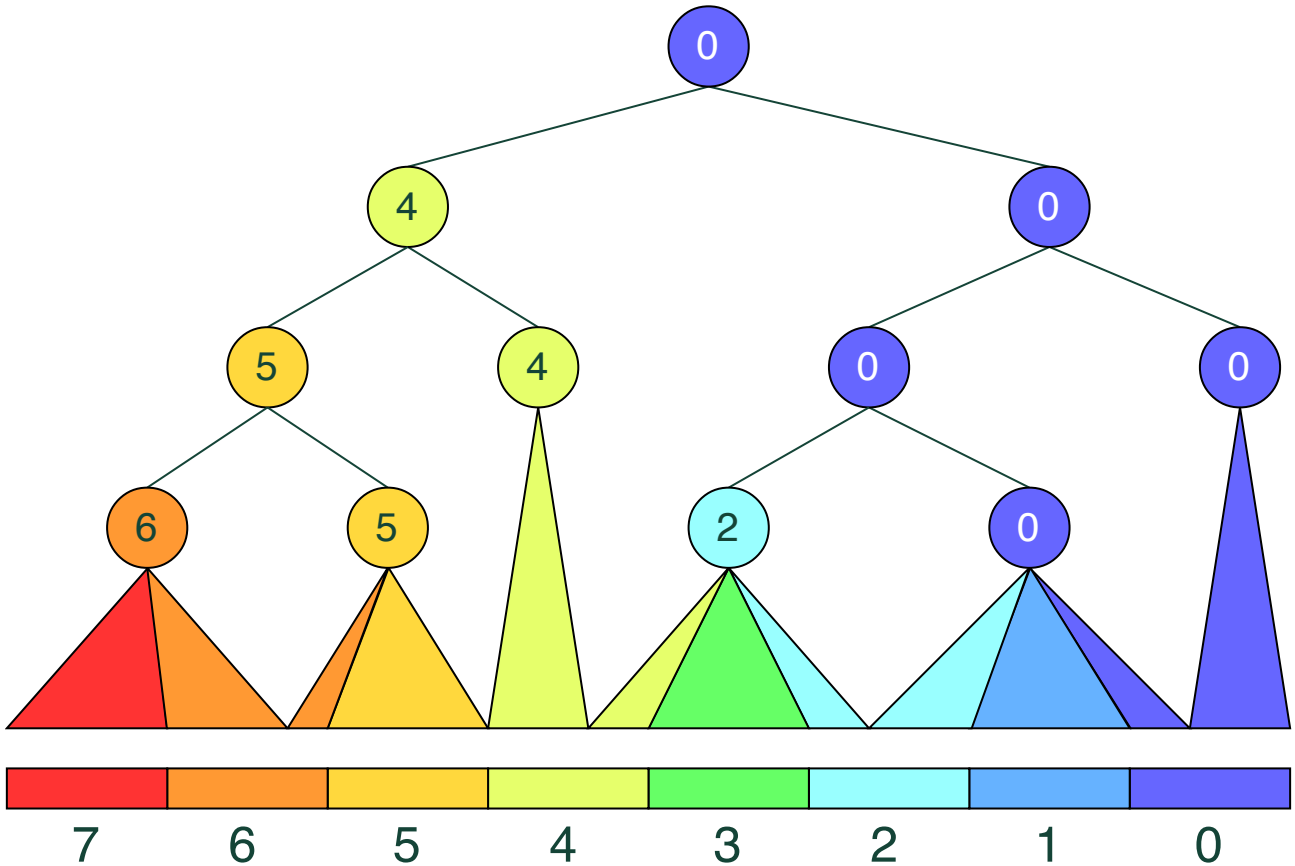
- **Deterministic Task Allocation** では
  プログラマ指定の仕事量を元に負荷分散
- **大雑把な仕事量**の指定でも上手くやってほしい
  - e.g. 仕事量は必ずしも粒子数に比例しない
- 仕事量が正確でも**実行時のノイズ**で実行時間は変化し得る
  - e.g. OS ノイズ, CPU の周波数スケーリング

Hierarchical Localized Work Stealing

- **Deterministic Task Allocation** では
  プログラマ指定の仕事量を元に負荷分散
- **大雑把な仕事量**の指定でも上手くやってほしい
  - e.g. 仕事量は必ずしも粒子数に比例しない
- 仕事量が正確でも**実行時のノイズ**で実行時間は変化し得る
  - e.g. OS ノイズ, CPU の周波数スケーリング

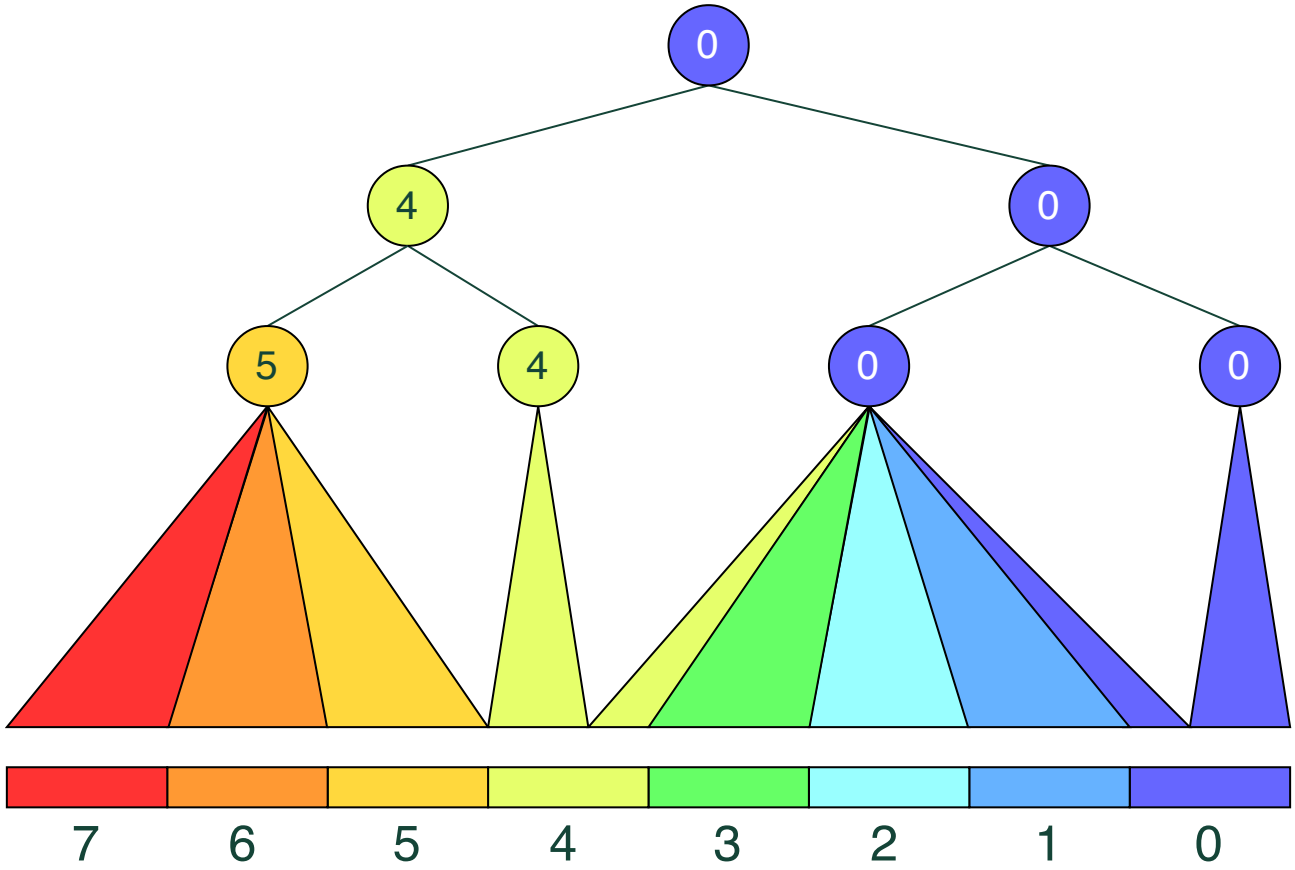
局所性に配慮し, かつ動的な負荷分散を行う手法

steal アルゴリズムの概略 (1/4)



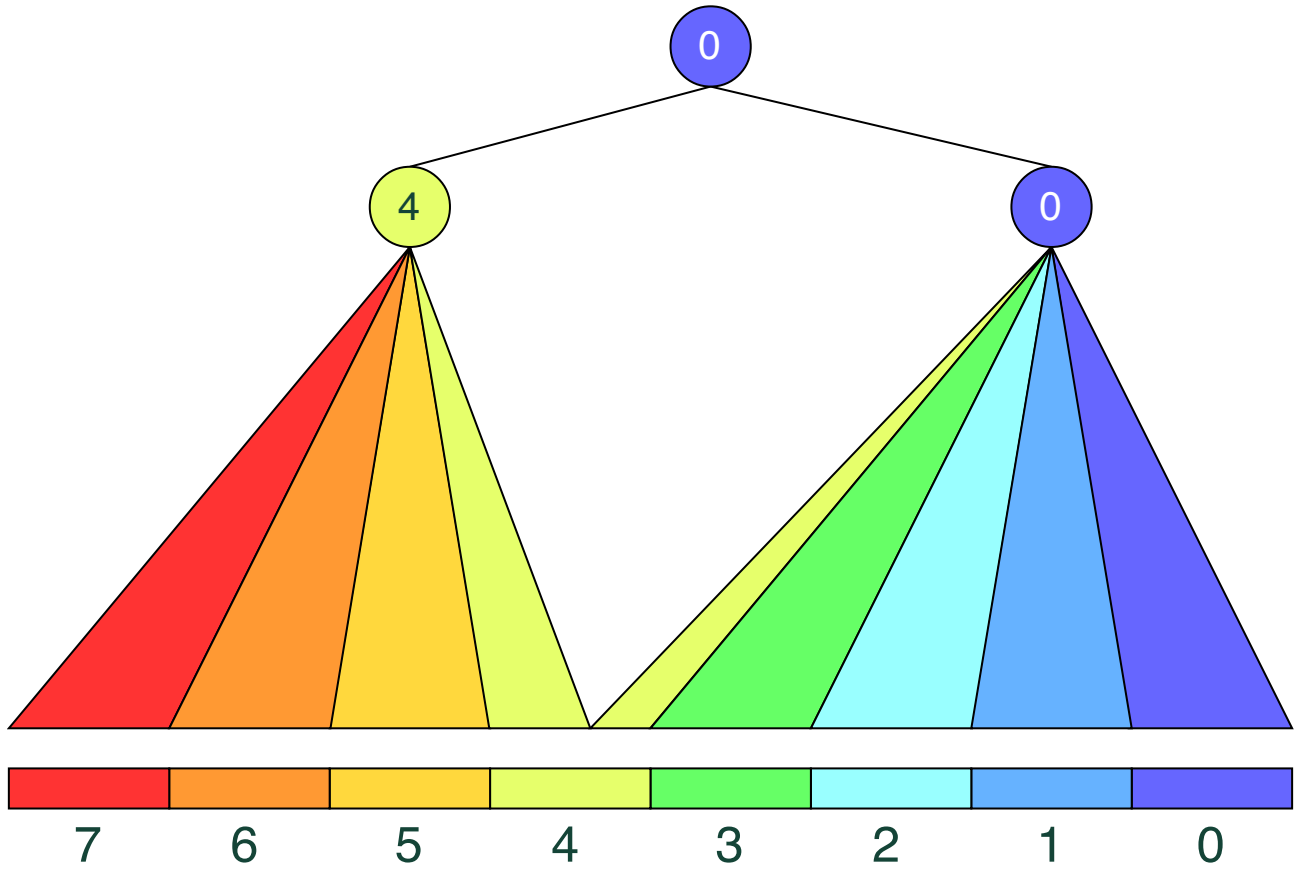
- **Deterministic Task Allocation** で行った分割がベース
- steal する範囲を自分の属するグループに制限

steal アルゴリズムの概略 (2/4)



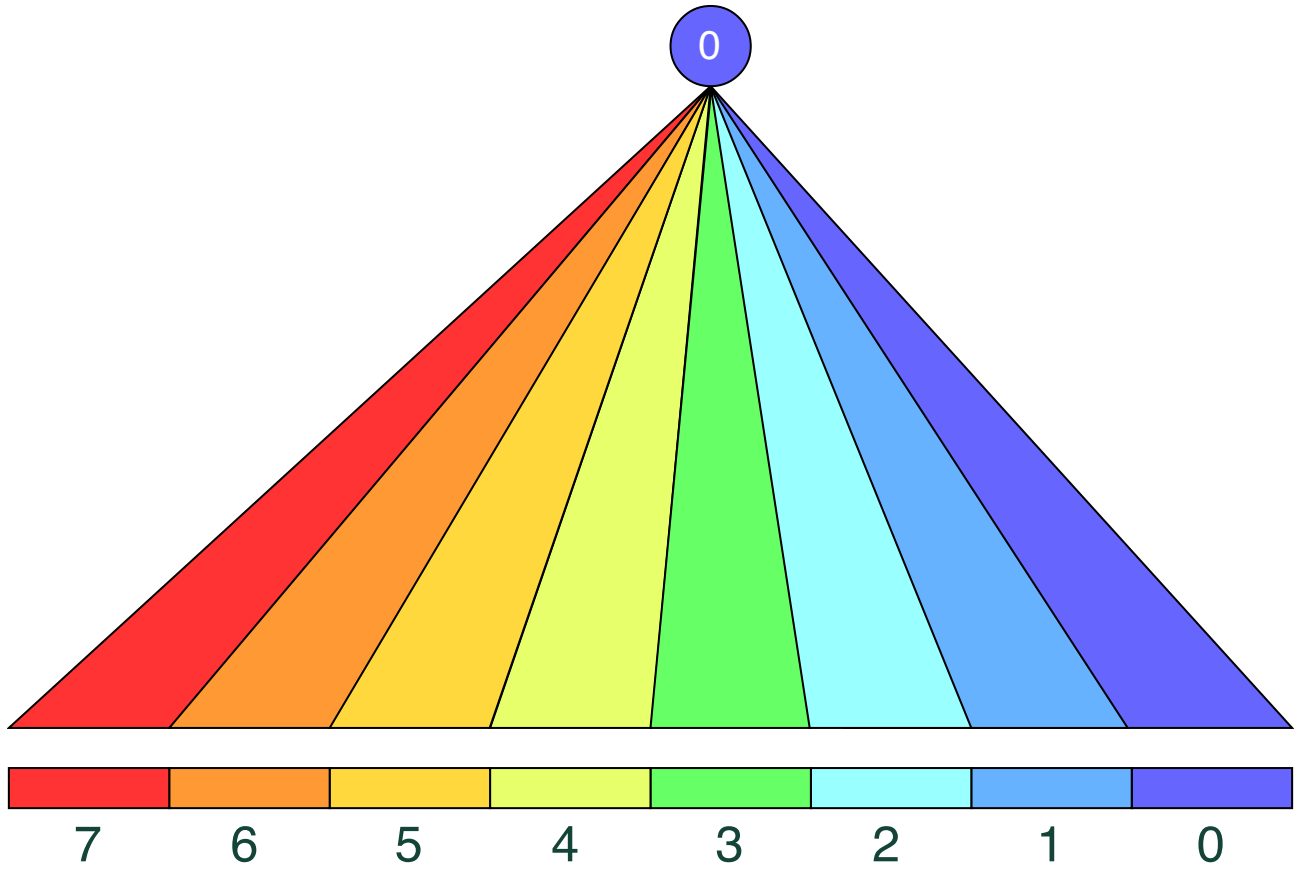
- 現在のグループにタスクがなくなれば親のグループに移行

steal アルゴリズムの概略 (3/4)



- タスクが少なくなると steal する worker の範囲が広くなる
- Deterministic Task Allocation の分割をボトムアップにたどる

steal アルゴリズムの概略 (4/4)



- 最終的にはただのランダムな **Work Stealing** と同じになる
- 理想的には, この時点で残存タスクはほぼ存在しない

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

結論

性能評価

MassiveThreads[4] 上に ADWS を実装し,
既存の Random Work Stealing と性能を比較

測定環境:

- 1 ノード内での測定（64 コア）
- 4 socket の NUMA 構成
- 16 cores / socket (Xeon Gold 6130, Skylake)

[4] J. Nakashima and K. Taura, “MassiveThreads: A thread library for high productivity languages,” in Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday. Springer Berlin Heidelberg, 2014, pp. 222–238.

粒子法における負荷分散の様子（既存手法）

64 core で負荷分散した様子を可視化（青: 0 → 赤: 63）

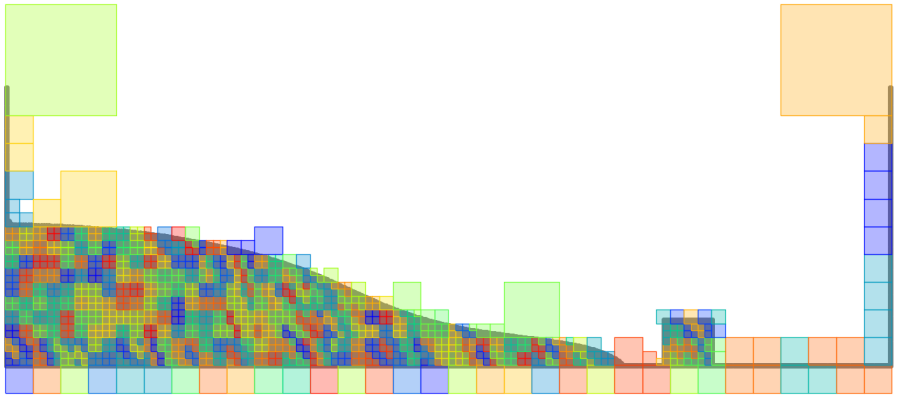


Fig. OpenMP dynamic (original)

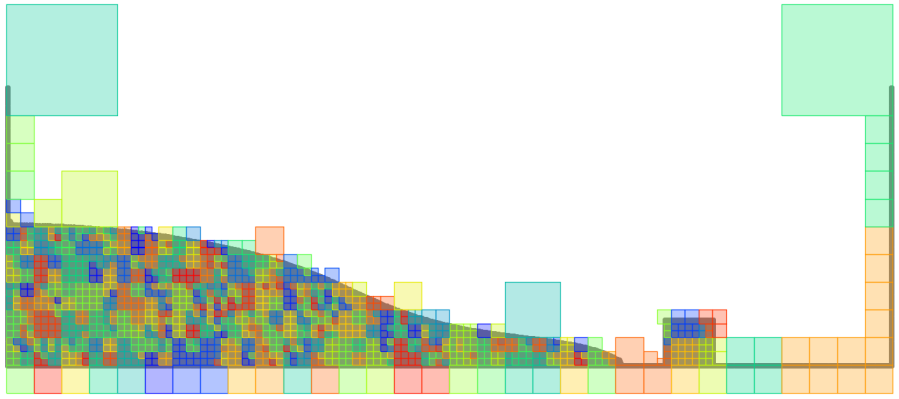


Fig. Random Work Stealing

- 負荷の偏りが生じやすいため, **動的負荷分散**が望ましい
- **FDPS** では OpenMP dynamic を使用
- 同一 NUMA ノード内の worker の担当領域が飛び飛びで
  局所性が良くない

粒子法における負荷分散の様子（提案手法）

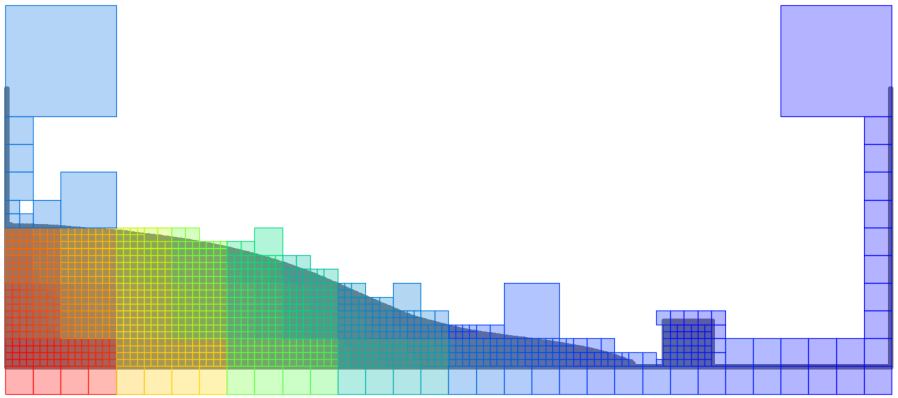


Fig. ADWS (no steal)

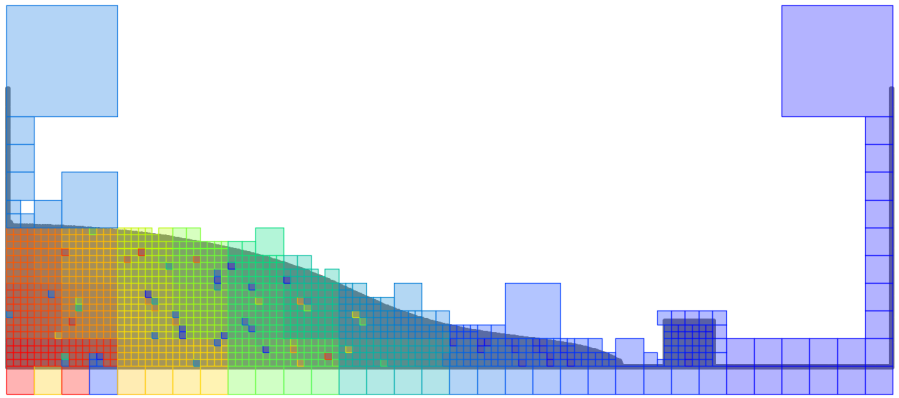
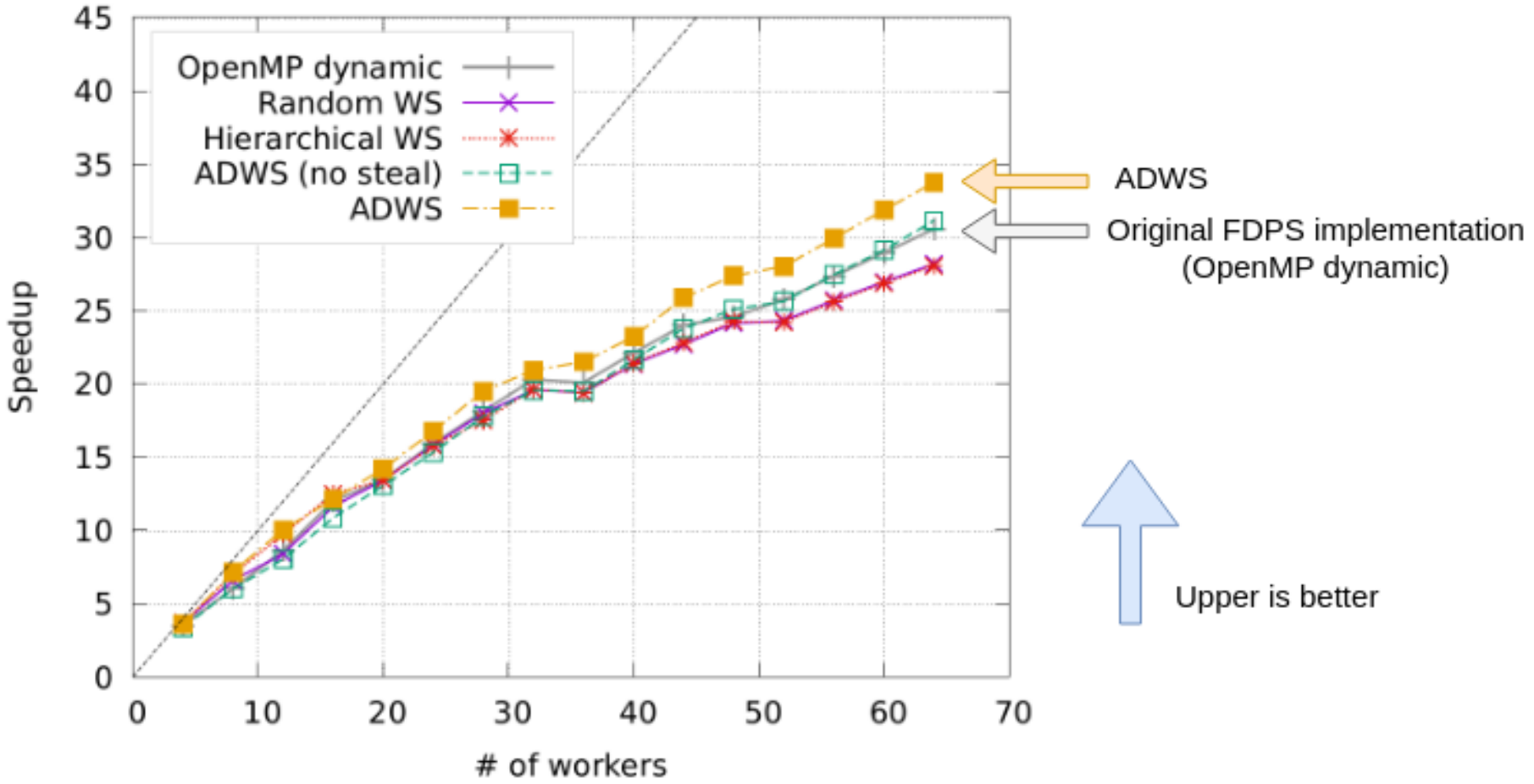


Fig. ADWS

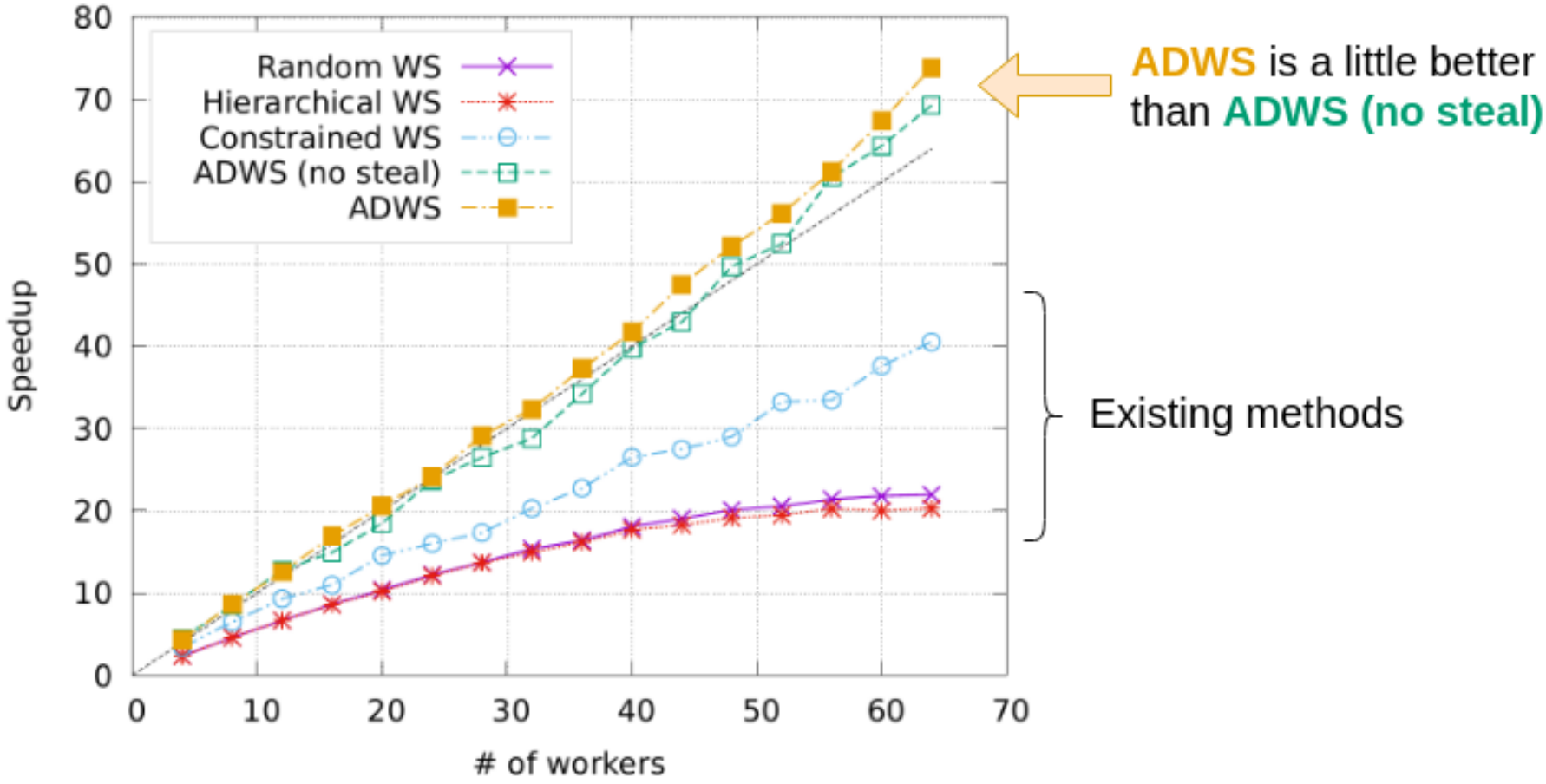
- Deterministic Task Allocation のみを行った場合（左）では
  局所性は良いが**負荷に偏り**が生じる
- 動的な負荷分散を加えた場合（右）では
  だいたいの局所性を確保しつつ動的に負荷分散を行う

粒子間相互作用の性能評価



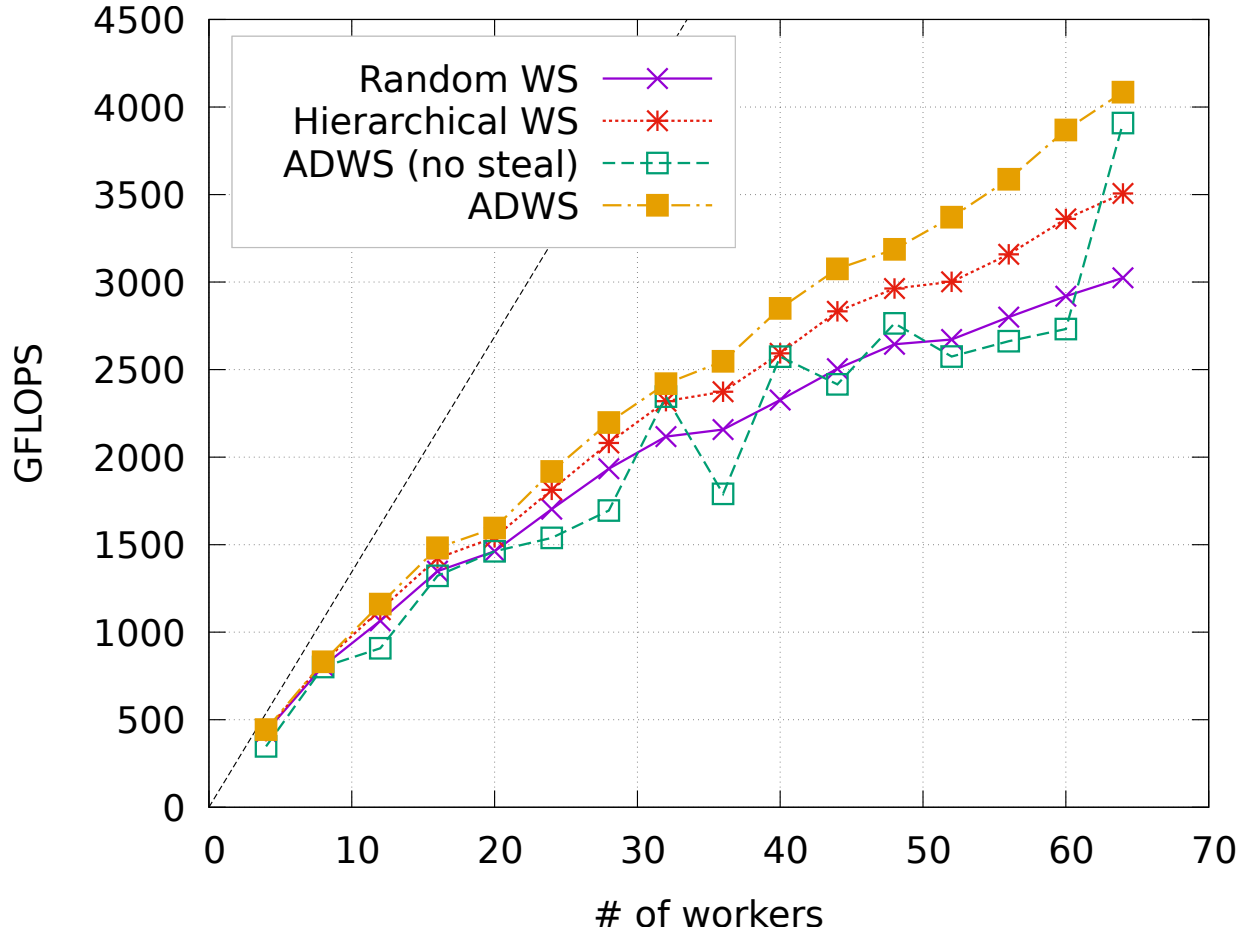
- 粒子数 138968, 2 次元のダム崩壊の計算
- 既存の FDPS の実装と比較しても良い性能を示している

2 次元熱伝導の性能評価



- 4096x4096 のサイズ, 非常にメモリ律速なアプリケーション
- 既存手法と比べ高い性能向上を達成

行列積の性能評価



- 4096x4096, 計算カーネルを SIMD 命令を用いて最適化
- 簡潔な記述ながら理論性能の半分近くを達成

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

結論

関連研究: iterative なプログラムに限定するもの

- 前回の iteration の実行の情報を元に
  動的に実行を最適化する[5][6][7]
- 本研究は iterative なプログラムに限定しない

5 U. A. Acar, G. E. Blelloch, and R. D. Blumofe, “The data locality of work stealing,” in Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2000, pp. 1–12.

6 J. Lifflander, S. Krishnamoorthy, and L. V. Kale, “Optimizing data locality for fork/join programs using constrained work stealing,” in SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 857–868.

7 Q. Chen, M. Guo, and H. Guan, “LAWS: Locality-aware work-stealing for multi-socket multi-core architectures,” in Proceedings of the 28th ACM International Conference on Supercomputing, ACM, 2014, pp. 3–12.

関連研究: 局所性のヒントを必要とするもの

- SLAW[8], NUMA-WS[9]など
- ハードウェアに基づくヒントを要求するものが多い
  - アーキテクチャの構造をある程度理解する必要
  - 移植性が良くない
- 本研究ではハードウェアに依存しないヒントを必要とする

8 Y. Guo, J. Zhao, V. Cave, et al., "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–12.

9 J. Deters, J. Wu, Y. Xu, et al., "A NUMA-aware provably-efficient task-parallel platform based on the work-first principle," in 2018 IEEE International Symposium on Workload Characterization (IISWC), 2018, pp. 59–70.

Outline

研究の背景

- Case Study: 粒子法シミュレーションの並列化
- タスク並列モデルと分割統治法
- Work Stealing による動的負荷分散
- メモリアクセスの局所性

提案手法: Almost Deterministic Work Stealing (ADWS)

- Deterministic Task Allocation
- Hierarchical Localized Work Stealing

性能評価

関連研究

結論

結論

我々の求めるもの：

結論

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる

結論

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
    - 仕事量を与えることは要求するものの，
      従来のタスク並列モデルのように見通しよく記述可能

結論

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - 仕事量を与えることは要求するものの，
     従来のタスク並列モデルのように見通しよく記述可能
2. 処理系による**動的な負荷分散**

結論

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
    - 仕事量を与えることは要求するものの，
      従来のタスク並列モデルのように見通しよく記述可能
2. 処理系による**動的な負荷分散**
    - Hierarchical Localized Work Stealing

結論

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - 仕事量を与えることは要求するものの，
     従来のタスク並列モデルのように見通しよく記述可能
2. 処理系による**動的な負荷分散**
   - Hierarchical Localized Work Stealing
3. メモリアクセスの**局所性**の良いスケジューリング

結論

我々の求めるもの：

1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - 仕事量を与えることは要求するものの，
     従来のタスク並列モデルのように見通しよく記述可能
2. 処理系による**動的な負荷分散**
   - Hierarchical Localized Work Stealing
3. メモリアクセスの**局所性**の良いスケジューリング
   - ほぼ決定的なスケジューリングと，
     局所性に配慮した Work Stealing

結論

我々の求めるもの：

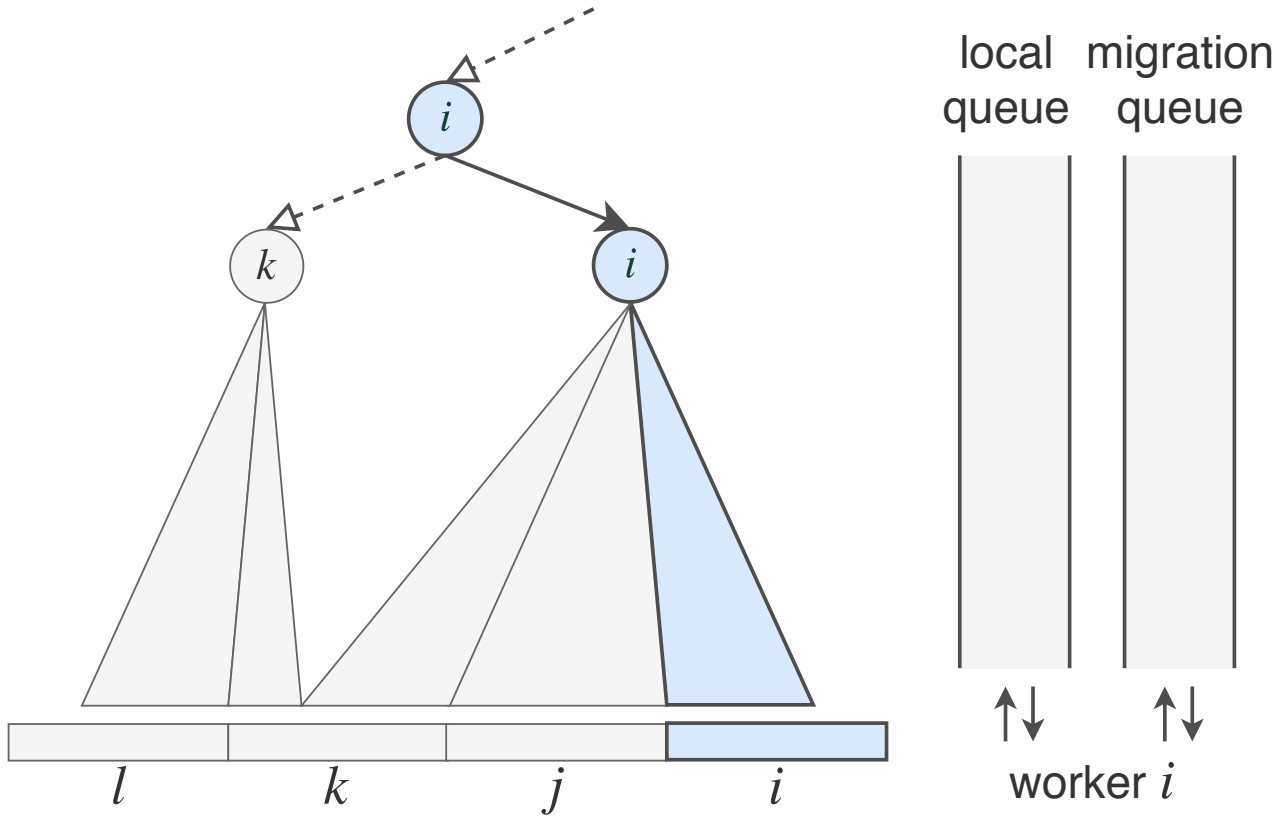
1. **分割統治法**で書かれたプログラムを簡単に並列化できる
   - 仕事量を与えることは要求するものの，
     従来のタスク並列モデルのように見通しよく記述可能
2. 処理系による**動的な負荷分散**
   - Hierarchical Localized Work Stealing
3. メモリアクセスの**局所性**の良いスケジューリング
   - ほぼ決定的なスケジューリングと，
     局所性に配慮した Work Stealing

生産性を保ったまま高い性能向上を達成

今後の課題

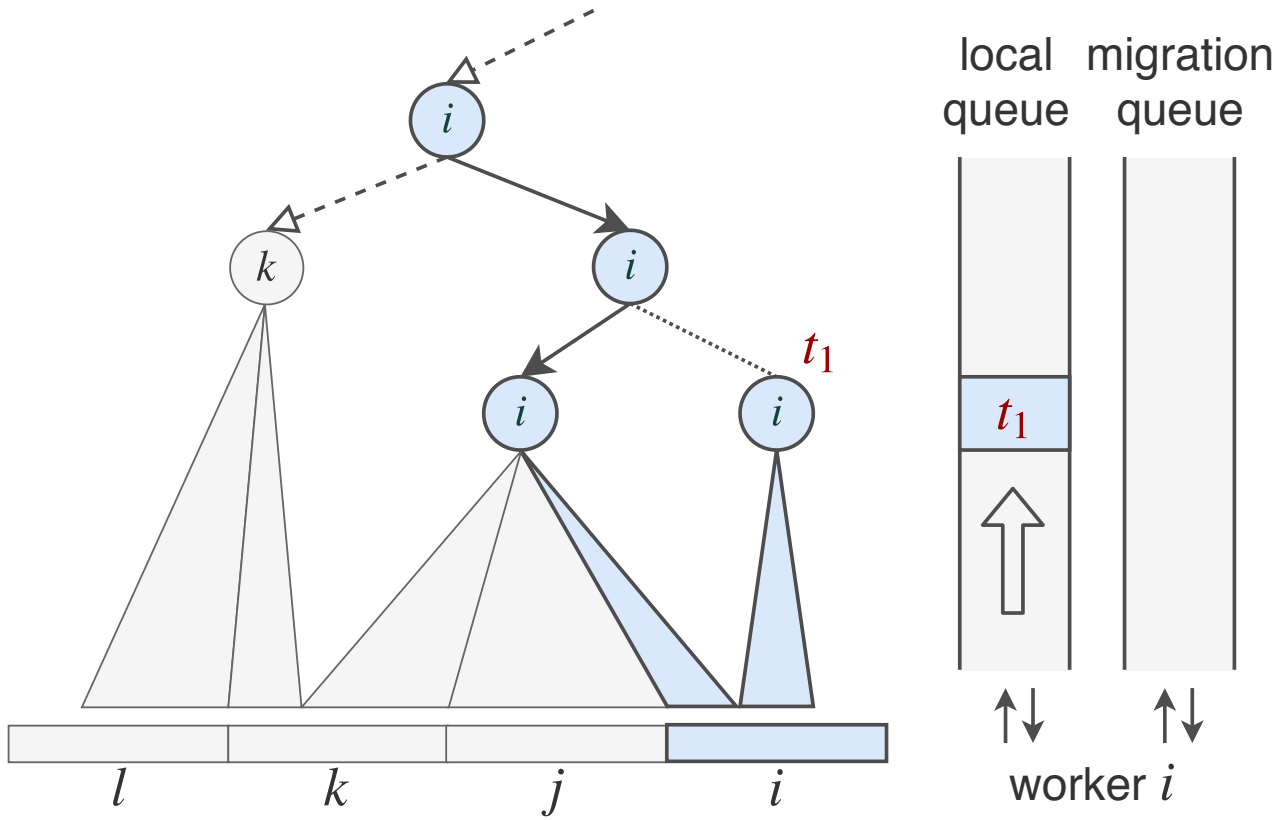
- **各タスクの仕事量の自動チューニング**
  - 今はプログラマに仕事量を与えることを要求しているが,
    iterative なプログラムに限定することで自動化できそう
- **分散環境における ADWS**
  - ADWS は任意のメモリ階層に対応可能
  - もちろん困難が伴うことも推して知るべし

負荷分散アルゴリズム (1/3)



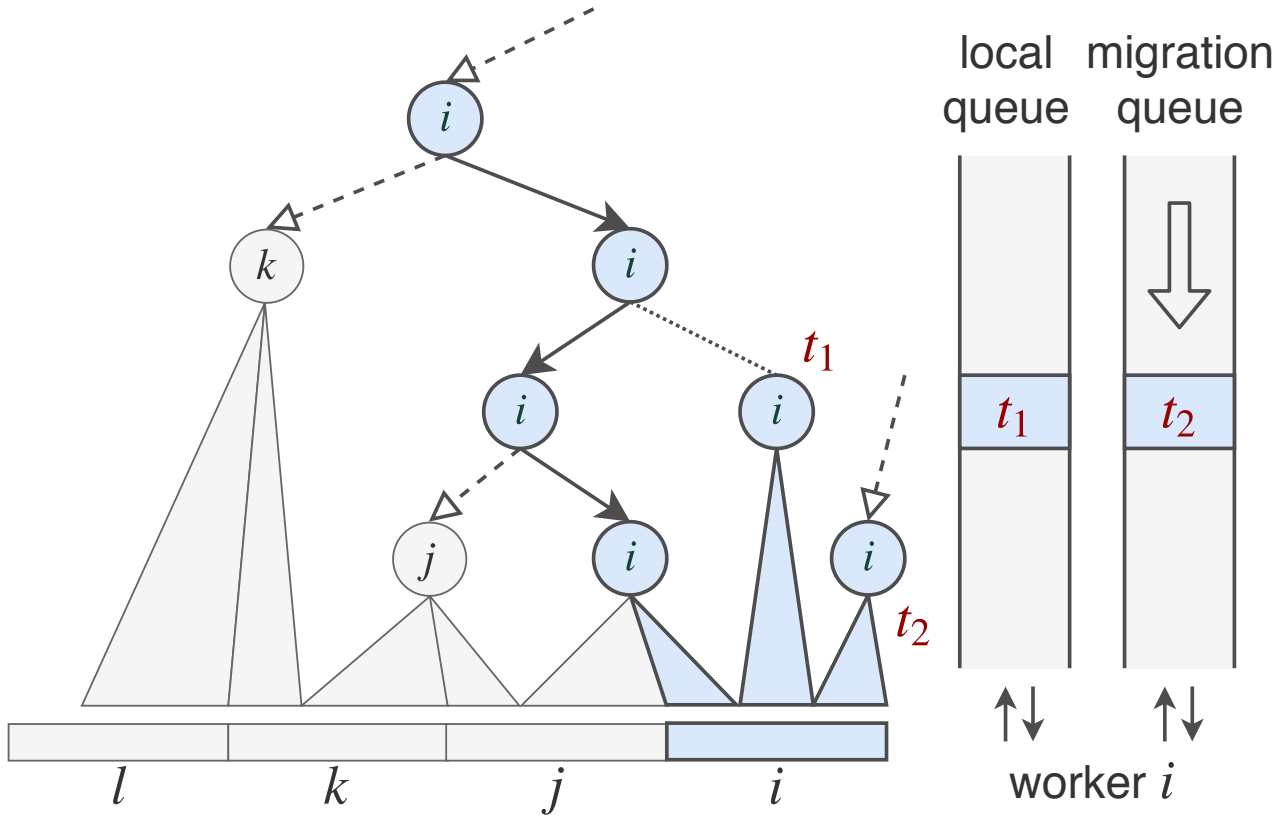
- 1 つの worker (i) の動きに着目
- worker は自身の担当領域の左側の境界を探索する (**search**)
- タスクの生成時, worker の範囲を分割して
  分割点にいた worker (k) にタスクを渡す (**migration**)

負荷分散アルゴリズム (2/3)



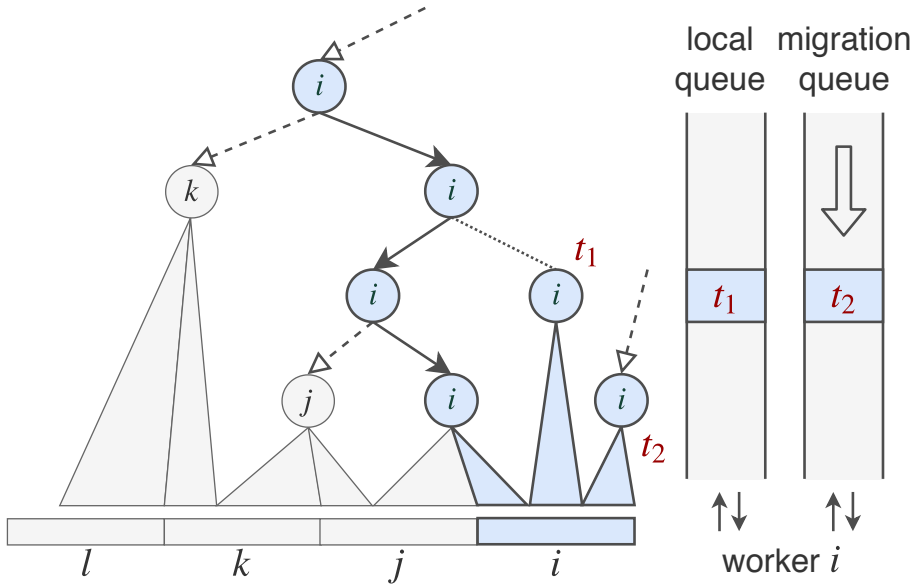
- 自身の領域で worker の範囲が分割されれば,
  生成されたタスクを実行する(左側へ進む)
- 継続(右側のノード; t_1)は local queue に push される

負荷分散アルゴリズム (3/3)



- 他の worker からタスク (t_2) が migrate される場合, migration queue にタスクが追加される

Deterministic Task Allocation の性質

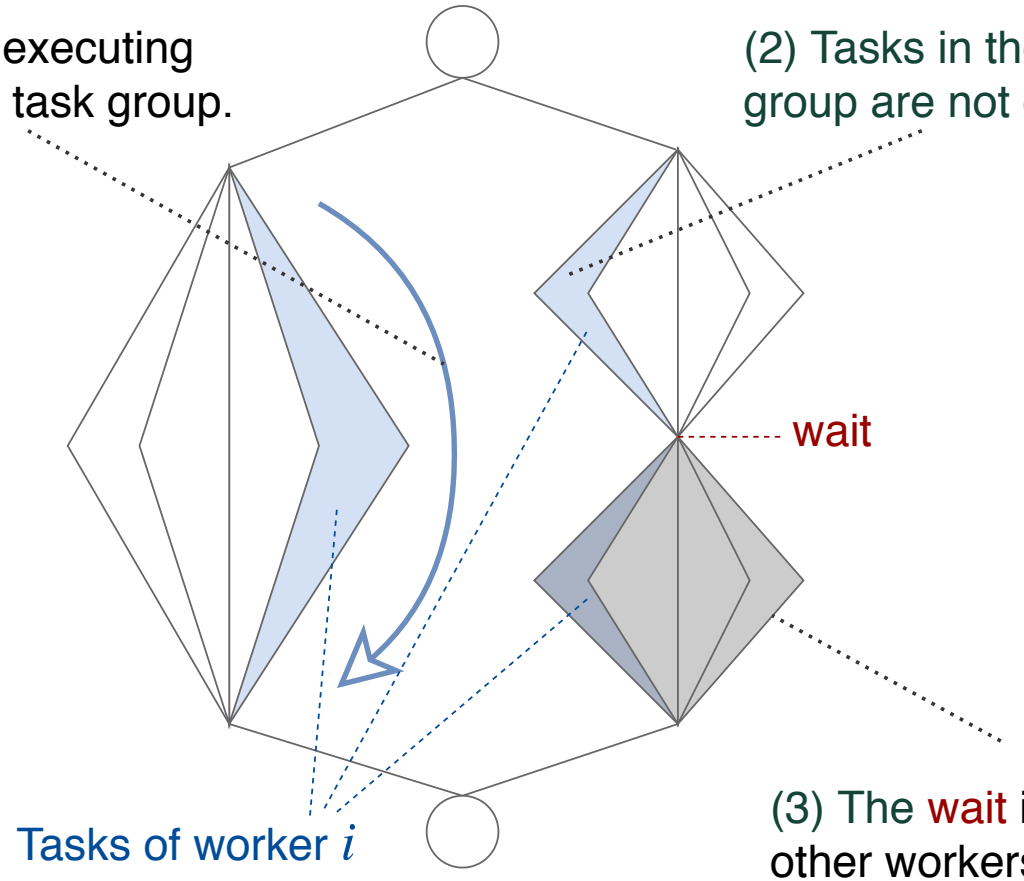


- タスクは左から右に実行される
  - search -> local queue -> migration queue
  - つまり**逐次の場合の実行順序が保たれる**
- migration 操作が **lock-free** である
  - 同時にある worker に対して migration が起きない（証明略）
  - search 中に lock contention が生じない

Deterministic Task Allocation の問題点

(1) First, worker i is executing tasks in the left-side task group.

(2) Tasks in the right-side task group are not executed yet.



(3) The wait is not resolved and other workers are forced to idle.