分散タスク並列処理系 Itoyori における局所性に配慮した大域アドレス空間およびスケジューリング

第 190 回 HPC 研究会 @ SWoPP 2023

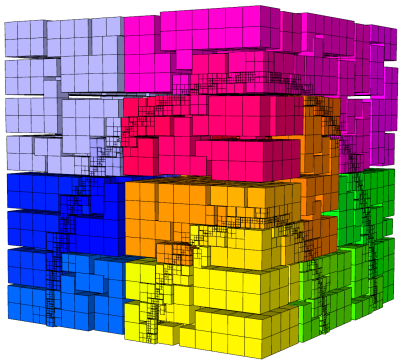
椎名 峻平、田浦 健次朗

東京大学大学院 情報理工学系研究科

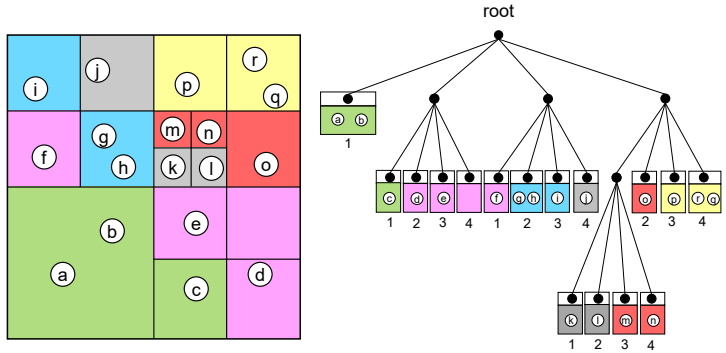
2023.08.04

“市場に並んだイトヨリダイ” by Totti, CC BY-SA 4.0.

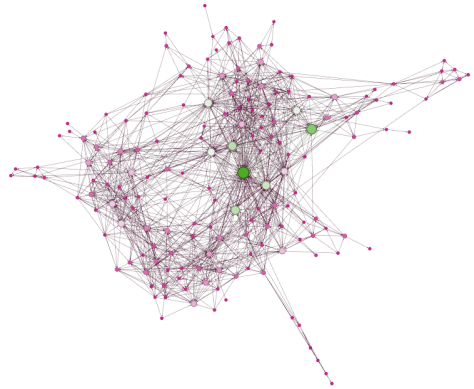
負荷分散の難しいアプリケーションを分散メモリでどう扱うか?



AMRの領域分割[1]



木構造による粒子の管理[2]



グラフ解析[3]

- 不規則な計算の負荷分散は難しい
  - 不均一なデータ構造、動的な並列性、入力データ依存の計算負荷
- プログラマが手動で負荷分散を記述することも可能だが、
  生産性を大きく損なう

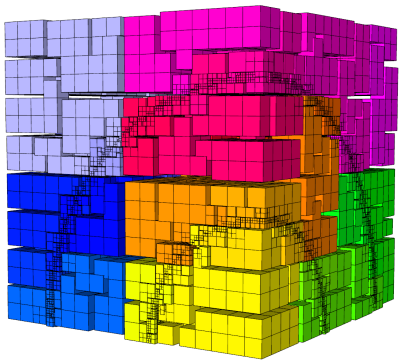
---

1 https://glvis.org/img/gallery/partition-2048-a.png

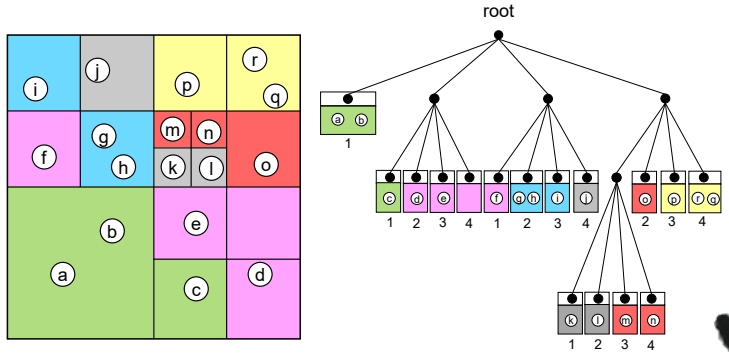
2 Satori Tsuzuki and Takayuki Aoki. “Effective Dynamic Load Balance Using Space-Filling Curves for Large-Scale SPH Simulations on GPU-Rich Supercomputers”. In: ScalA ’16. 2016

3 https://gephi.org/images/screenshots/layout1.png

負荷分散の難しいアプリケーションを分散メモリでどう扱うか?



AMRの領域分割[1]



木構造による粒子の管理[2]

グラフ解析[3]

- 不規則な計算の負荷分散は難しい
  - 不均一なデータ構造、動的な並列性、入力デー
- プログラマが手動で負荷分散を記述するこ
  生産性を大きく損なう

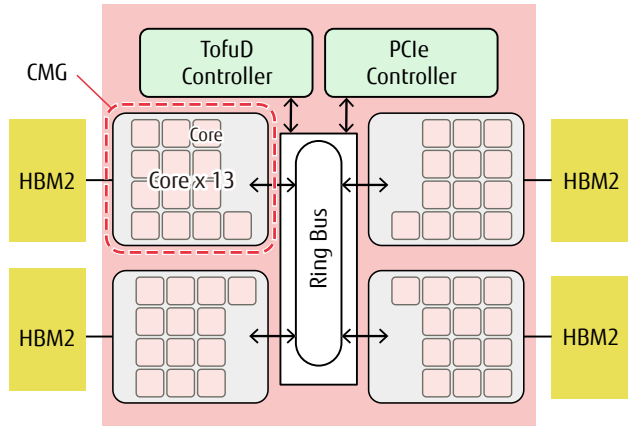
1 https://glvis.org/img/gallery/partition-2048-a.png

2 Satori Tsuzuki and Takayuki Aoki. “Effective Dynamic Load Balance Using Space-Filling Curves for Large-Scale SPH Simulations on GPU-Rich Supercomputers”. In: ScalA ’16. 2016

3 https://gephi.org/images/screenshots/layout1.png

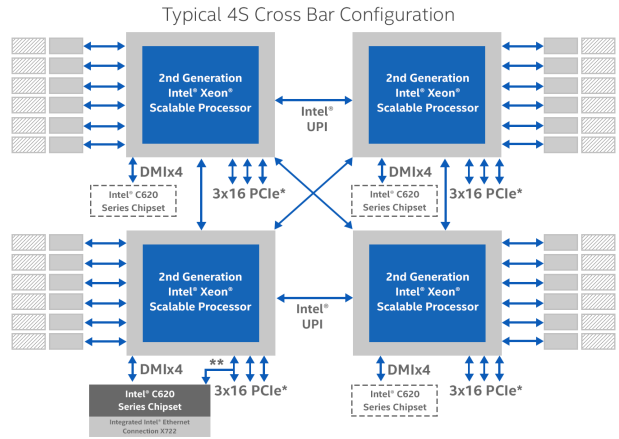
さらに、ノード内のマルチコア、深いメモリ階層

- CPUのコア数は増加傾向
- 1ノード(**共有メモリ**)内のメモリ階層:
  - Non-Uniform Memory Access (NUMA)
  - マルチソケット
  - 階層キャッシュ
- **プログラミングコストは増加する一方**
  - 分散/共有メモリの区別(**MPI+X** モデル)



HBM2: High Bandwidth Memory 2

富士通 A64FX(48コア)[1]



Intel Xeonプロセッサ (4ソケット)[2]

[1] https://www.fujitsu.com/global/documents/about/resources/publications/technicalreview/2020-03/article03.pdf

[2] https://www.intel.co.jp/content/www/jp/ja/products/platforms/details/cascade-lake.html

さらに、ノード内のマルチコア、深いメモリ階層

- CPUのコア数は増加傾向
- 1ノード(**共有メモリ**)内のメモリ階層:
  - Non-Uniform Memory Access (NUMA)
  - マルチソケット
  - 階層キャッシュ
- **プログラミングコストは増加する一方**
  - 分散/共有メモリの区別(**MPI+X** モデル)

[1] https://www.fujitsu.com/global/documents/about/resources/publications/technicalreview/2020-03/article03.pdf

[2] https://www.intel.co.jp/content/www/jp/ja/products/platforms/details/cascade-lake.html

提案 ▷ 分散タスク並列処理系 Itoyori[1]

高水準なグローバルビュー タスク並列モデルによる
高い生産性と性能の両立の実現

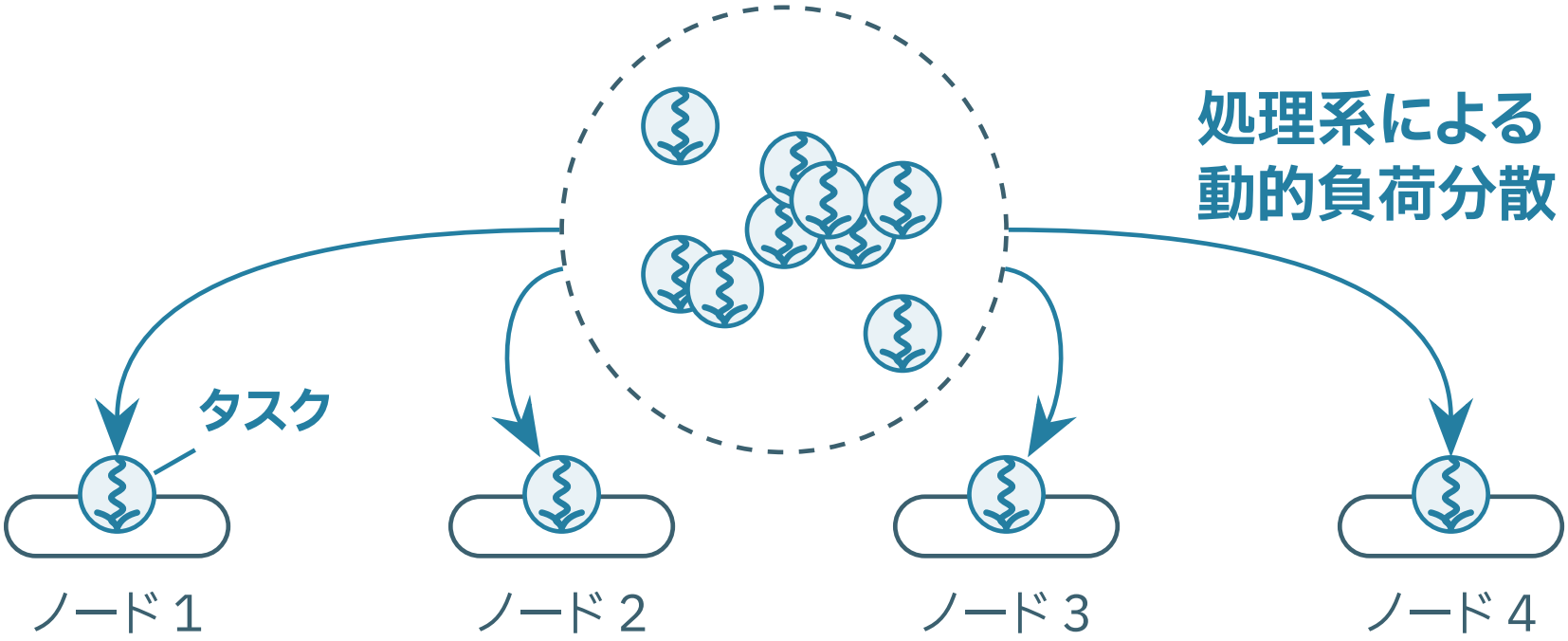
高生産性:

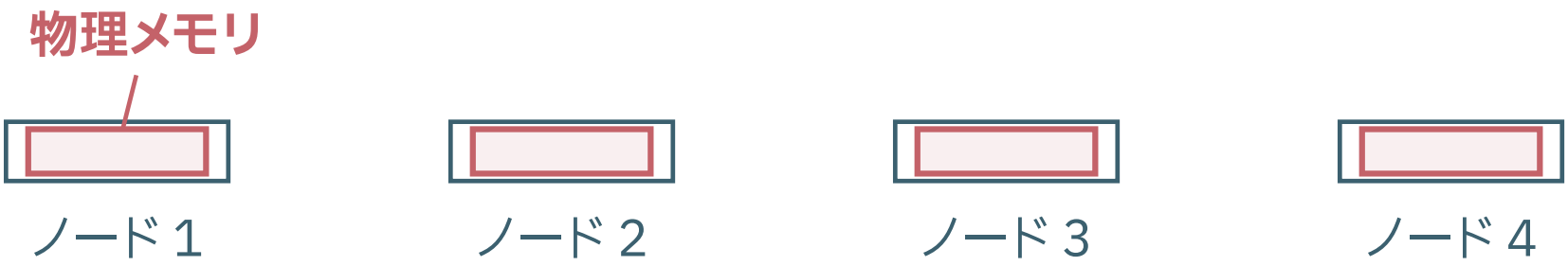
- 直観的でシンプルなグローバルビューに基づく**タスク並列モデル**
- 具体的なハードウェア階層、負荷分散、通信を忘れてプログラミング

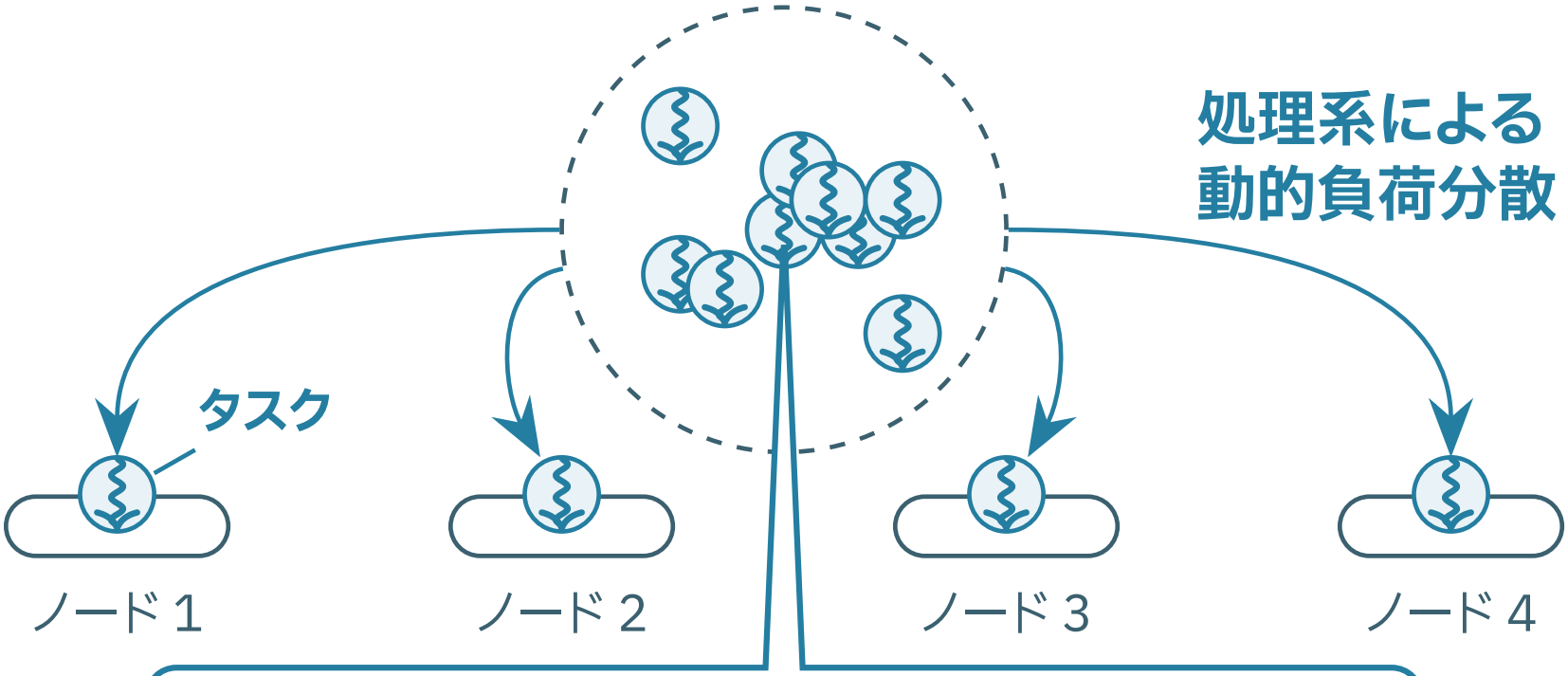
高性能:

- 動的負荷分散単体の評価では **10 万コア以上**のスケーラビリティ **[Shiina and Taura, Cluster '22]**
- 現実的なアプリでも MPI で手動で最適化された実装に比肩する性能

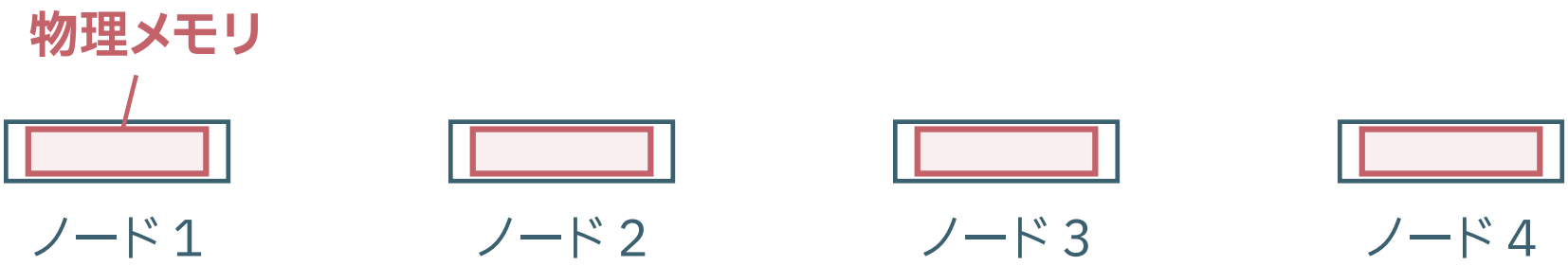
[1] 実装はオープンソースで公開: `https://github.com/itoyori/itoyori`

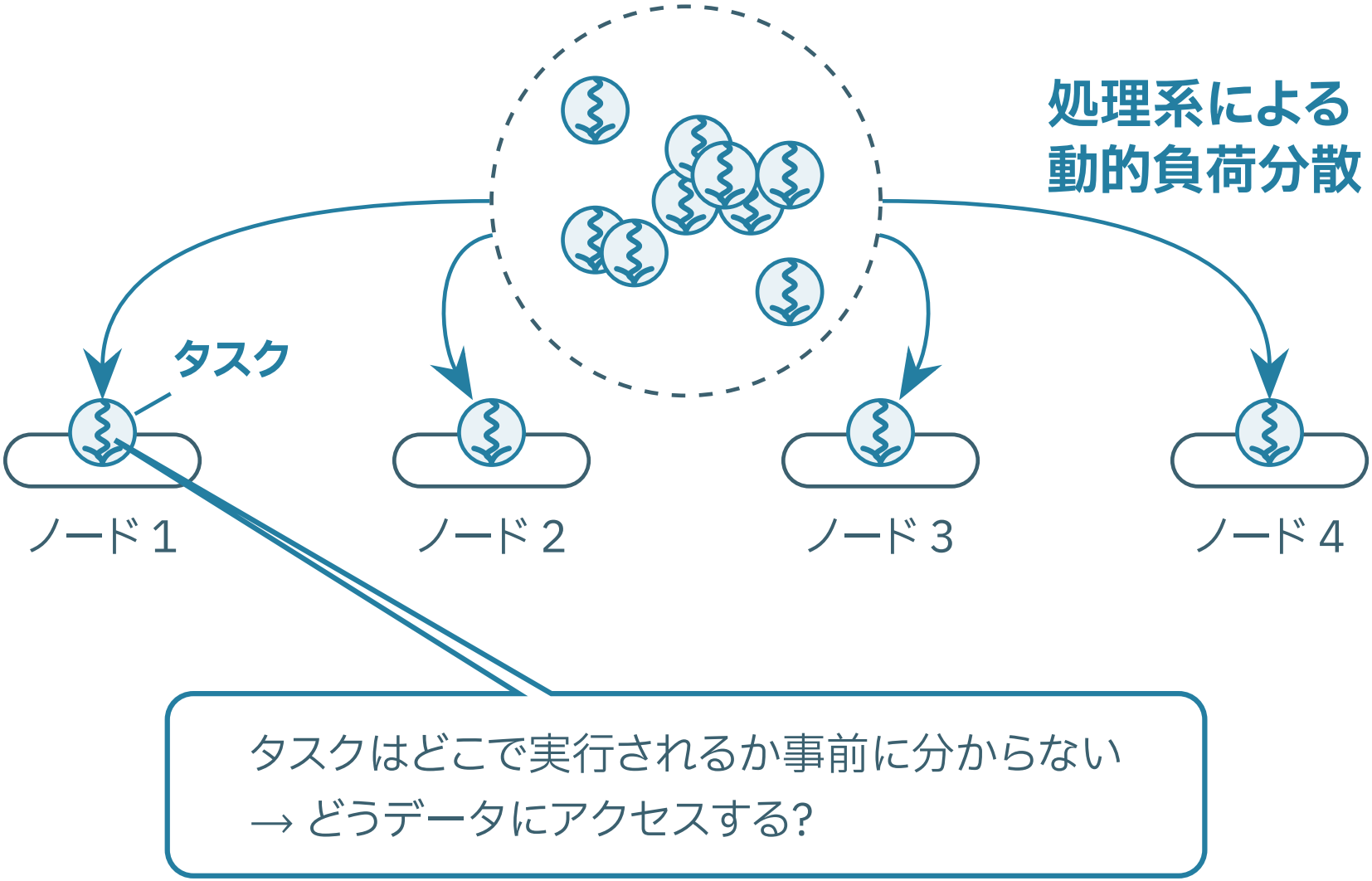

処理系による
動的負荷分散
タスク
ノード 1
ノード 2
ノード 3
ノード 4


物理メモリ
ノード 1
ノード 2
ノード 3
ノード 4

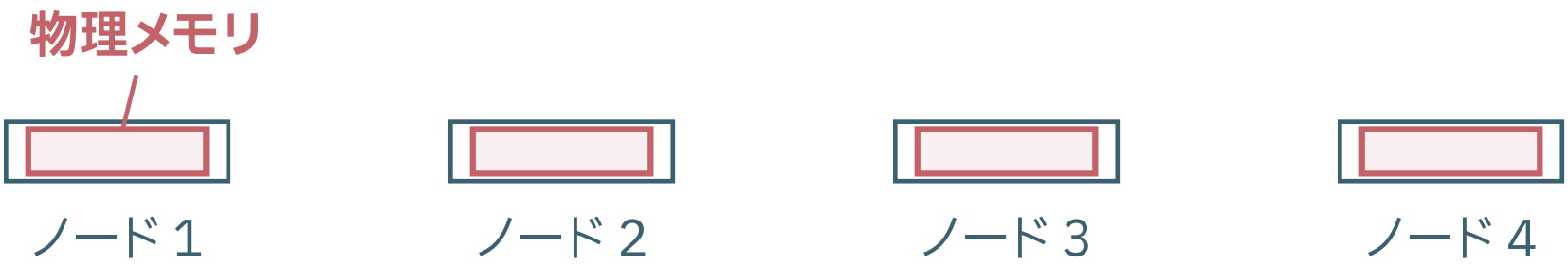


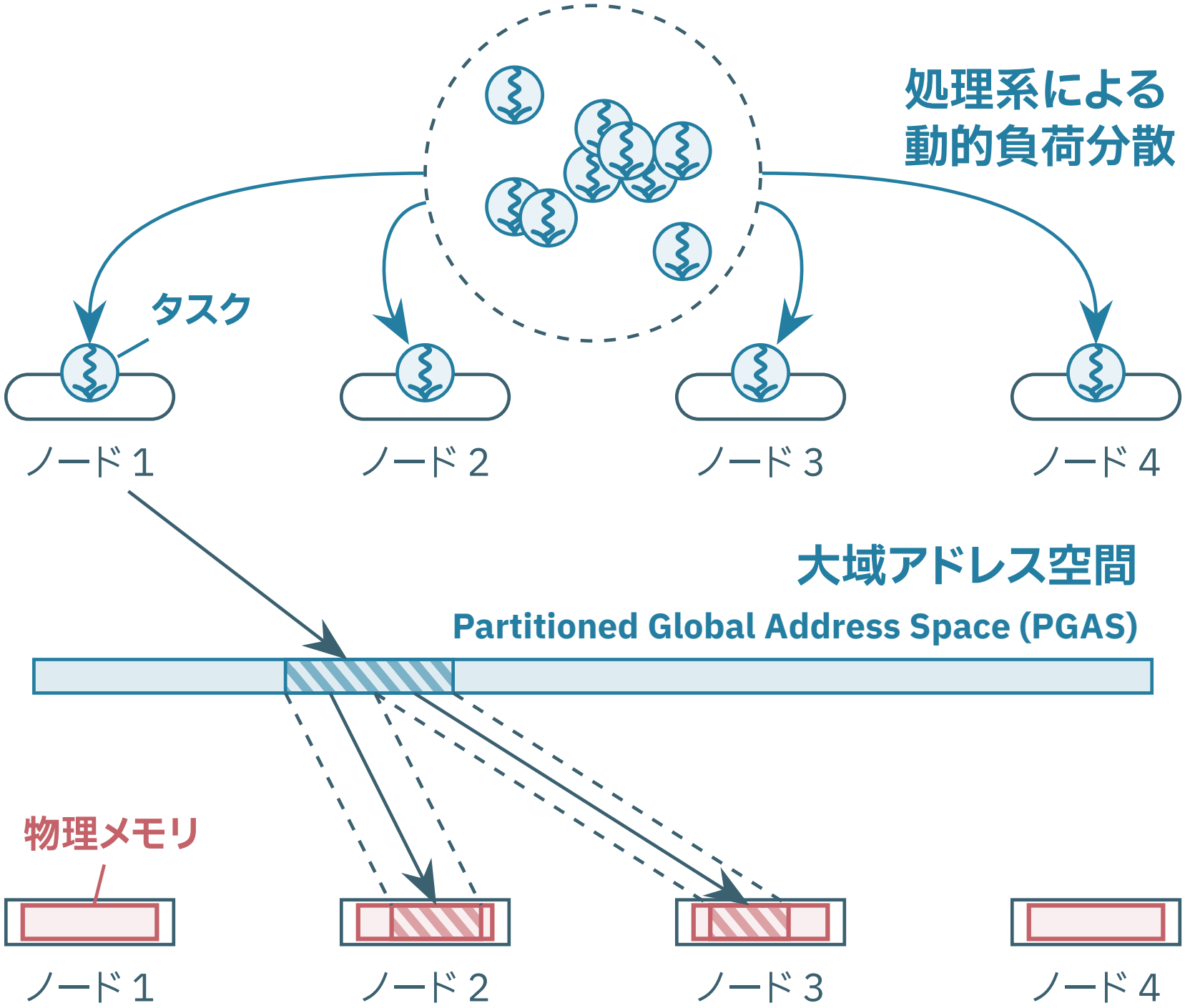
- プログラマは論理的な並列性を**タスク**として表現
- 任意のタイミングで動的にタスク生成、同期可能
  （動的な**タスク並列モデル**）

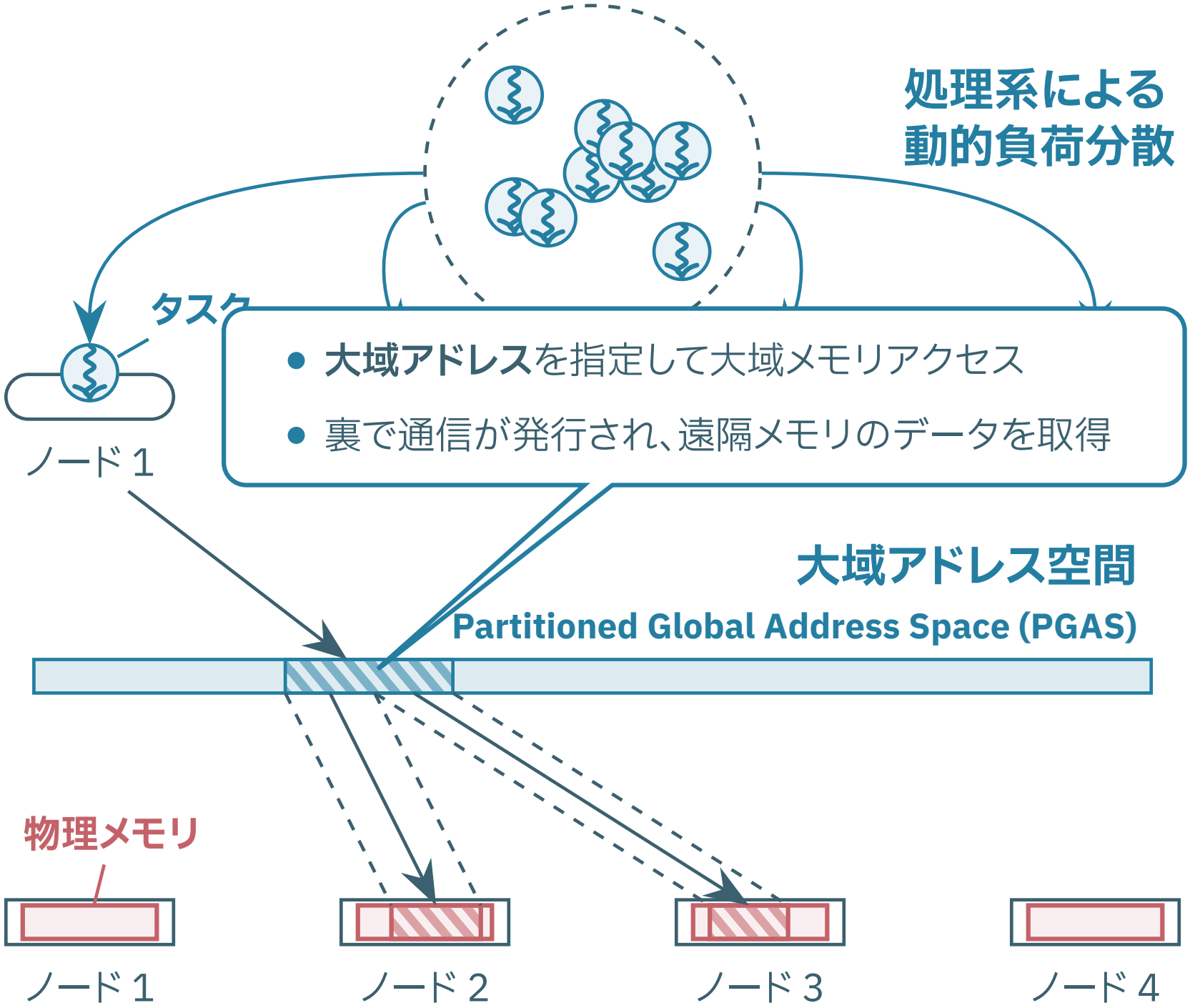




タスクはどこで実行されるか事前に分からない
→ どうデータにアクセスする?




処理系による
動的負荷分散
タスク
ノード 1
ノード 2
ノード 3
ノード 4
大域アドレス空間
Partitioned Global Address Space (PGAS)
物理メモリ
ノード 1
ノード 2
ノード 3
ノード 4


処理系による
動的負荷分散
タスク
ノード 1
• 大域アドレスを指定して大域メモリアクセス
• 裏で通信が発行され、遠隔メモリのデータを取得
大域アドレス空間
Partitioned Global Address Space (PGAS)
物理メモリ
ノード 1
ノード 2
ノード 3
ノード 4

Itoyori のプログラム例 ▷ 並列マージソート

逐次プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {

    msort(a       , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Itoyori のプログラム例 ▷ 並列マージソート

逐次プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {

    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

入力配列を 2 分割してそれぞれ
再帰的にソートする**分割統治法**

Itoyori のプログラム例 ▷ 並列マージソート

逐次プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {

    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

入力配列を 2 分割してそれぞれ再帰的にソートする**分割統治法**

ソート済みの 2 つの配列をマージ

Itoyori のプログラム例 ▷ 並列マージソート

逐次プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {

    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

小さい配列向けのより高速なアルゴリズム（挿入ソートなど）に切り替え

入力配列を 2 分割してそれぞれ再帰的にソートする**分割統治法**

ソート済みの 2 つの配列をマージ

Itoyori のプログラム例 ▷ 並列マージソート

逐次プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {

    msort(a      , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Itoyori プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a      , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    merge(a, n, n/2);
  }
}
```

Itoyori のプログラム例 ▷ 並列マージソート

逐次プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {

    msort(a       , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

Itoyori プログラム:

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a       , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    merge(a, n, n/2);
  }
}
```

2 つの再帰呼出しを並列実行可能な**タスク**として生成(**Fork**)し、
それらの完了を待つ(**Join**) ⇨ **タスク並列**(Fork/Joinモデル)

Itoyori のプログラム例 ▷ 並列マージソート

大域アドレスにアクセスする前後で
checkout/checkin APIを呼ぶ

逐次プ

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {

    sort_small(a, n);

  } else {

    msort(a       , n/2);
    msort(a + n/2, n/2);

    merge(a, n, n/2);
  }
}
```

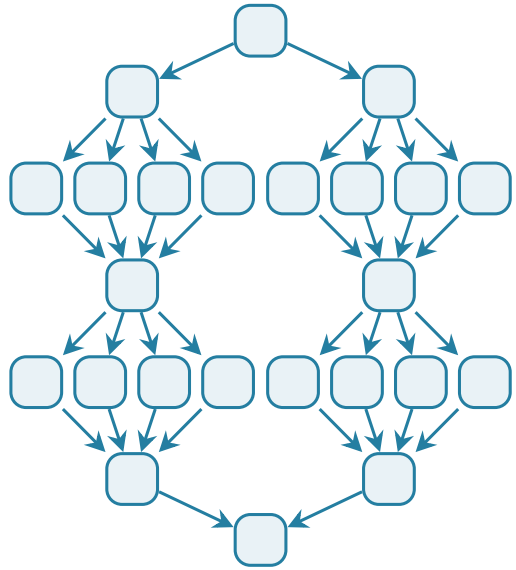
```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a       , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    merge(a, n, n/2);
  }
}
```

2 つの再帰呼出しを並列実行可能な**タスク**として生成(**Fork**)し、
それらの完了を待つ(**Join**) ⇨ **タスク並列**(Fork/Joinモデル)

タスク並列(Fork/Join)モデル

- 動的に生成(**Fork**)されるタスクを
  Joinの依存関係を満たしながら並列実行
- 任意のタイミングで、再帰的にタスク生成可能
  - **分割統治法**と相性が良い
  - 大量のタスク(≫ CPUコア数)を作っても良い

```
// Fork/Join を一度に記述できる
parallel_invoke(
  [=]{ msort(a      , n/2); },
  [=]{ msort(a + n/2, n/2); }
);
```



Fork/Join の依存関係グラフ

- **汎用的:** 多くの問題が簡潔に記述可能
  - 行列演算、FFT、ソート、N 体問題、動的計画法、木探索、...

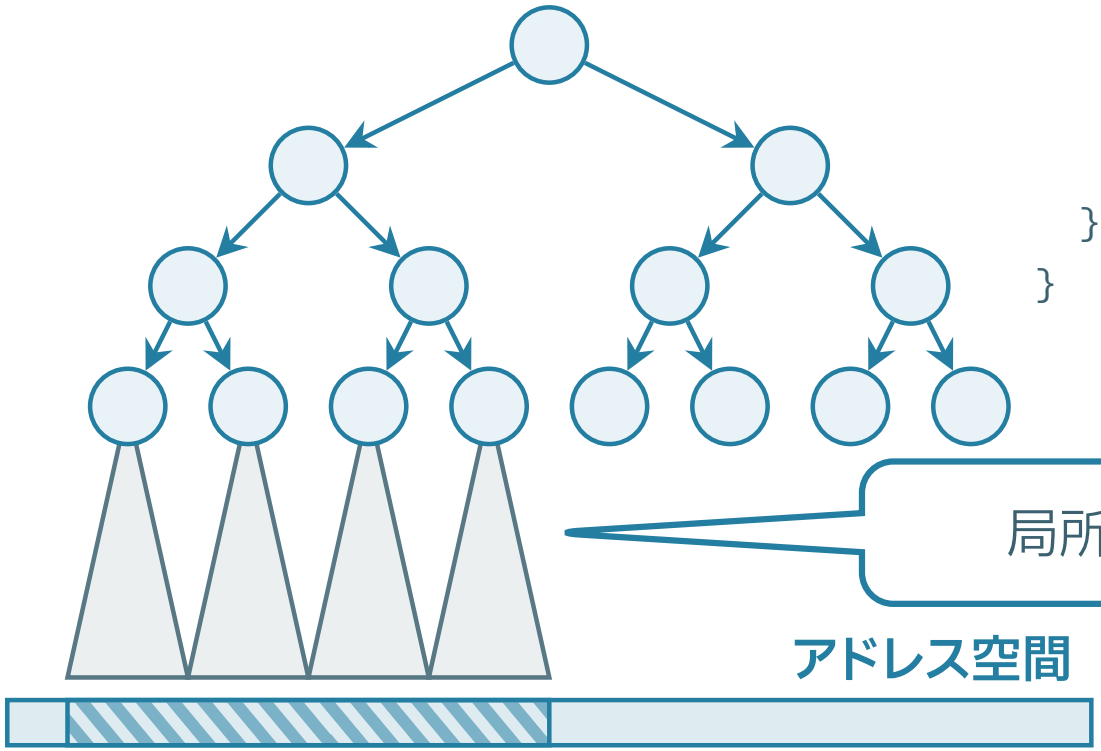
タスク並列と大域アドレス空間(PGAS)を単に組み合わせるだけでは性能が出ない

- 既存の処理系では**局所性**への配慮が足りていない
  - Scioto **[Dinan+, SC' 09]**、HotSLAW **[Min+, PGAS '11]**、Grappa **[Nelson+, USENIX ATC '15]**、など
- **問題 1:** 同じ大域メモリへの複数回アクセスで冗長な通信が発生
  - 各タスクで独立に細かい通信が冗長に発行されてしまう
- **問題 2:** タスクの実行位置が深いメモリ階層を考慮しない
  - タスクスケジューリング戦略の問題
- タスクが実行されるノードが事前に決定できないため、プログラマ側での最適化も困難

前提 ▷ タスク並列における局所性

近い依存関係を持つ**近隣タスクは**
近いメモリにアクセスする傾向

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a      , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    merge(a, n, n/2);
  }
}
```

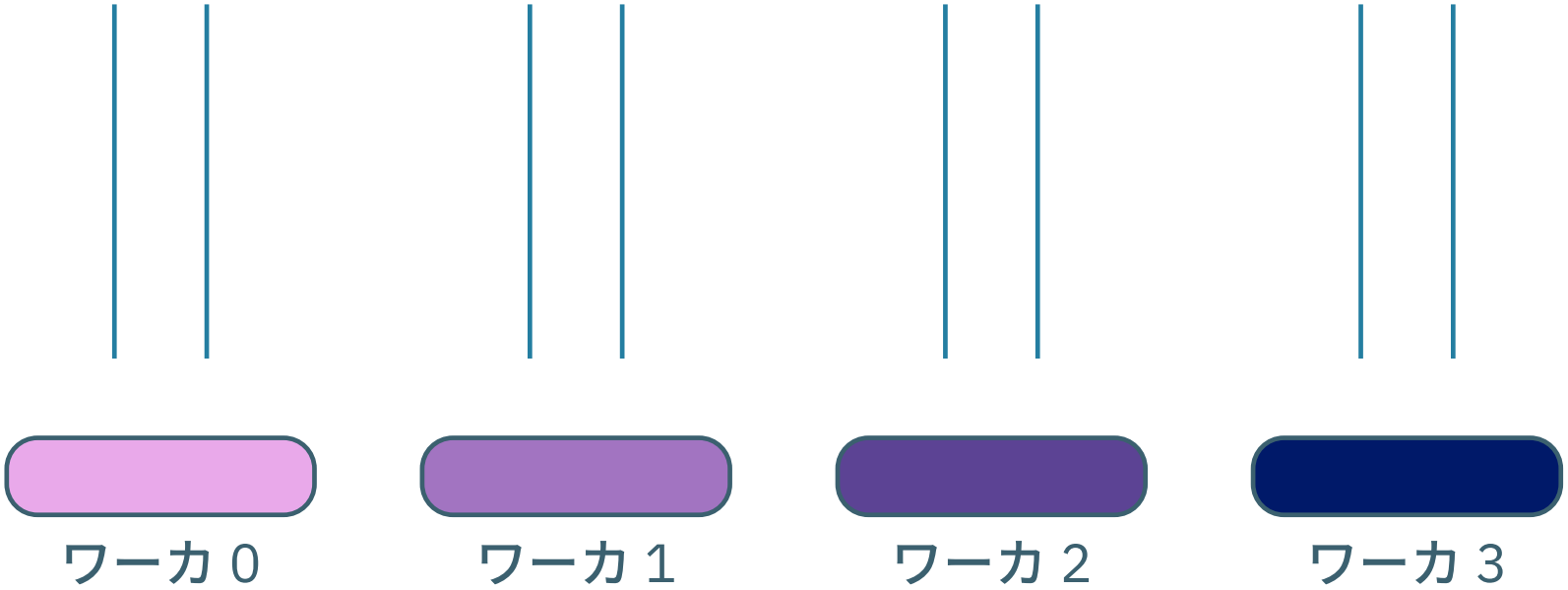


局所性のあるメモリアクセス

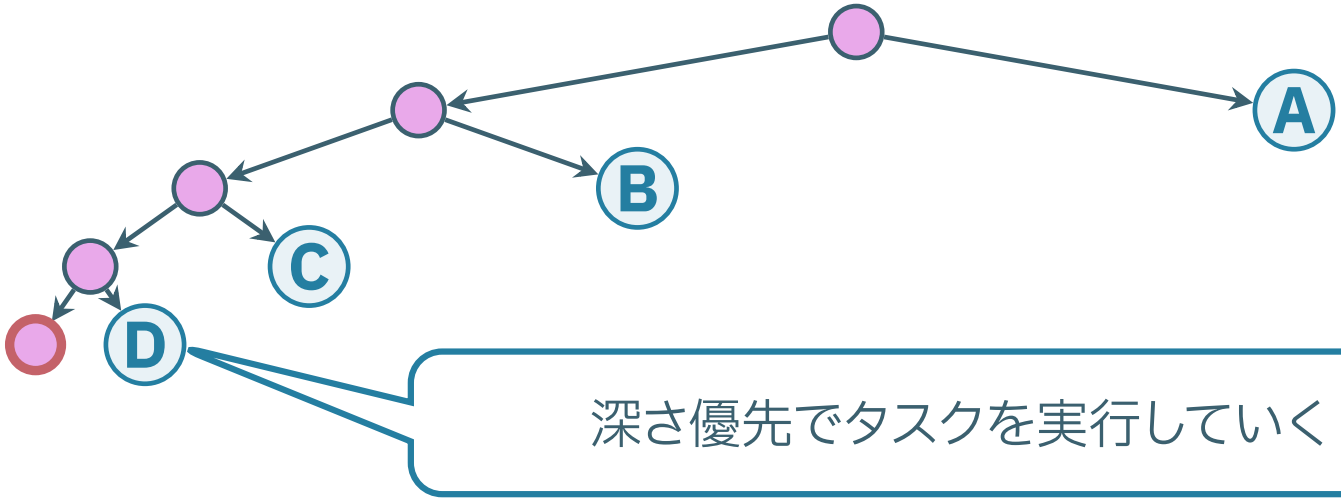
ワークスティーリングによるスケジューリング例

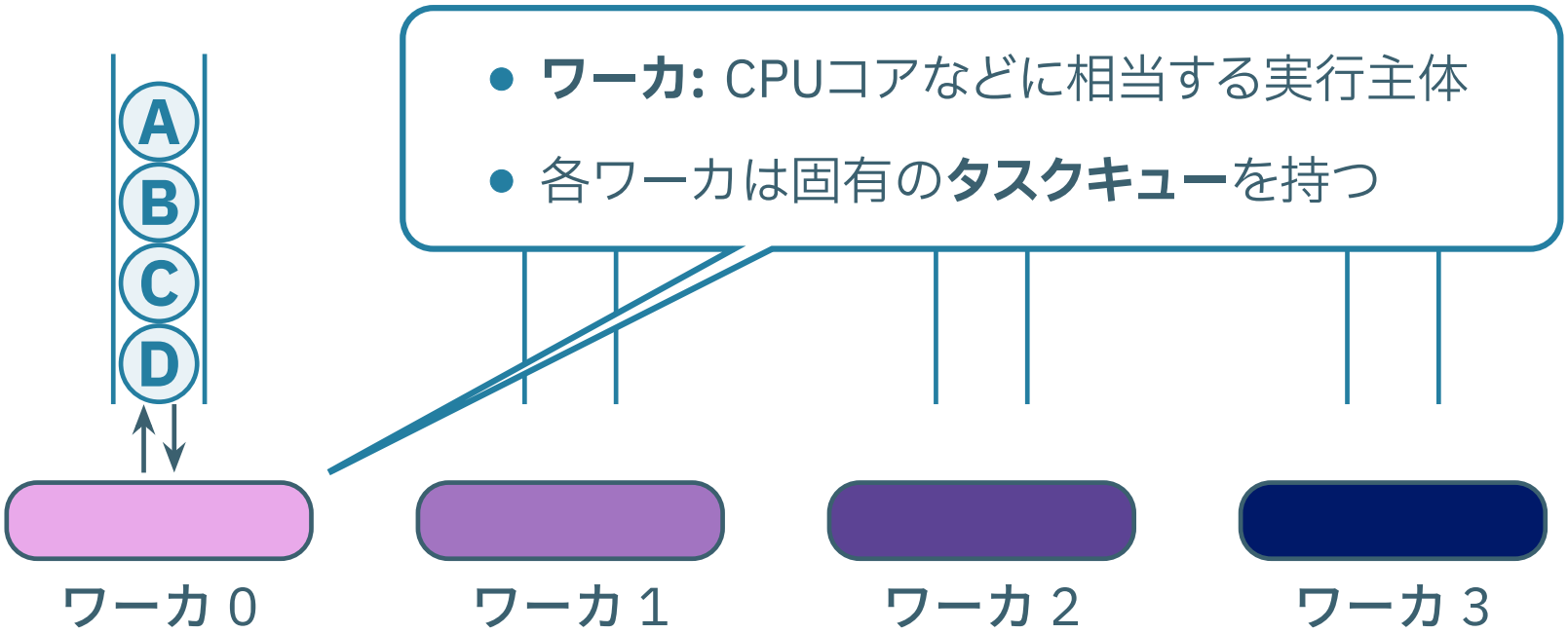
- **ワークスティーリング**[a]はタスク並列処理のスケジューラとして広く用いられる
- メモリ階層は無視したランダムなスケジューリング

[a] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *Journal of the ACM* 46.5 (1999).

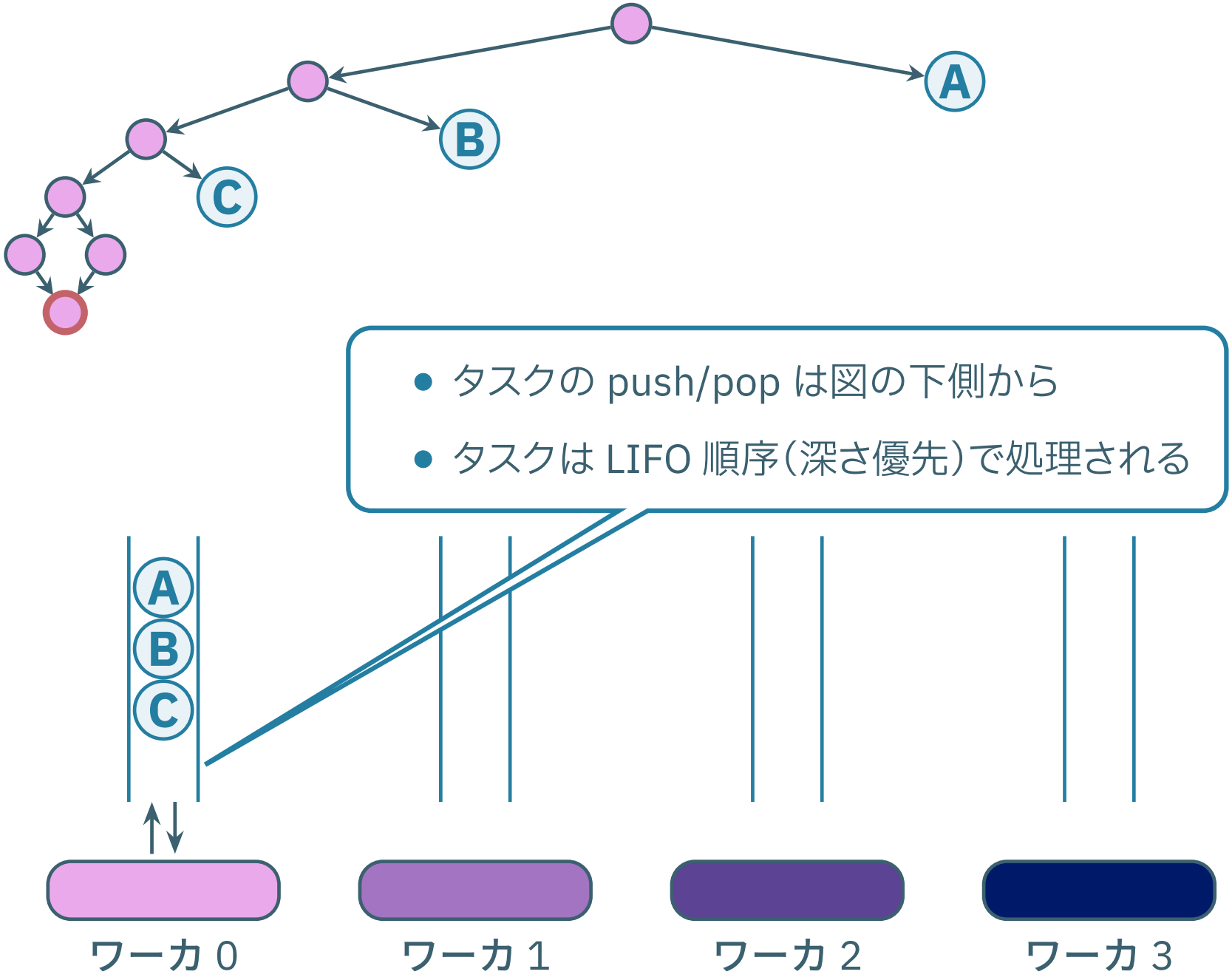


ワークスティーリングによるスケジューリング例

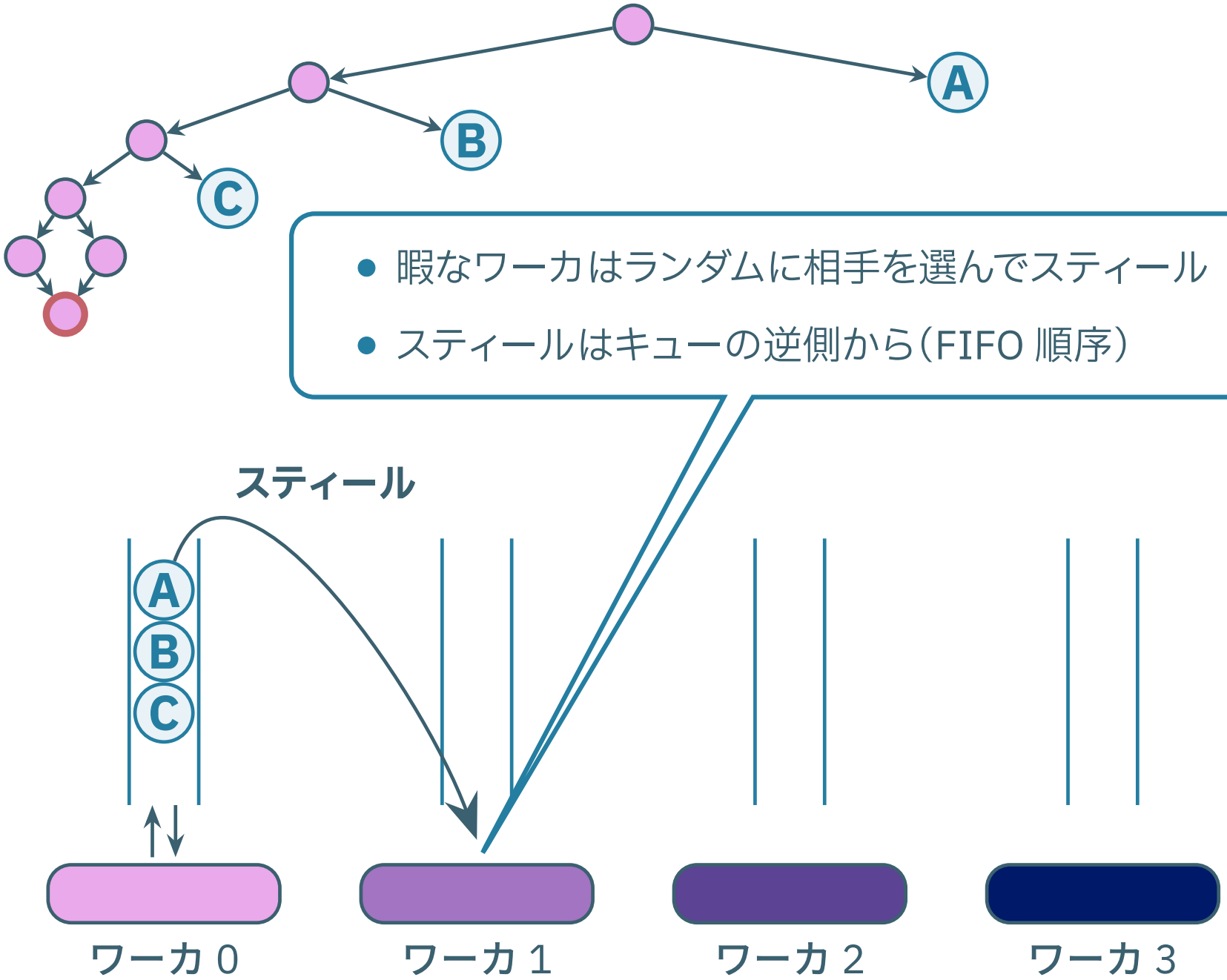




ワークスティーリングによるスケジューリング例

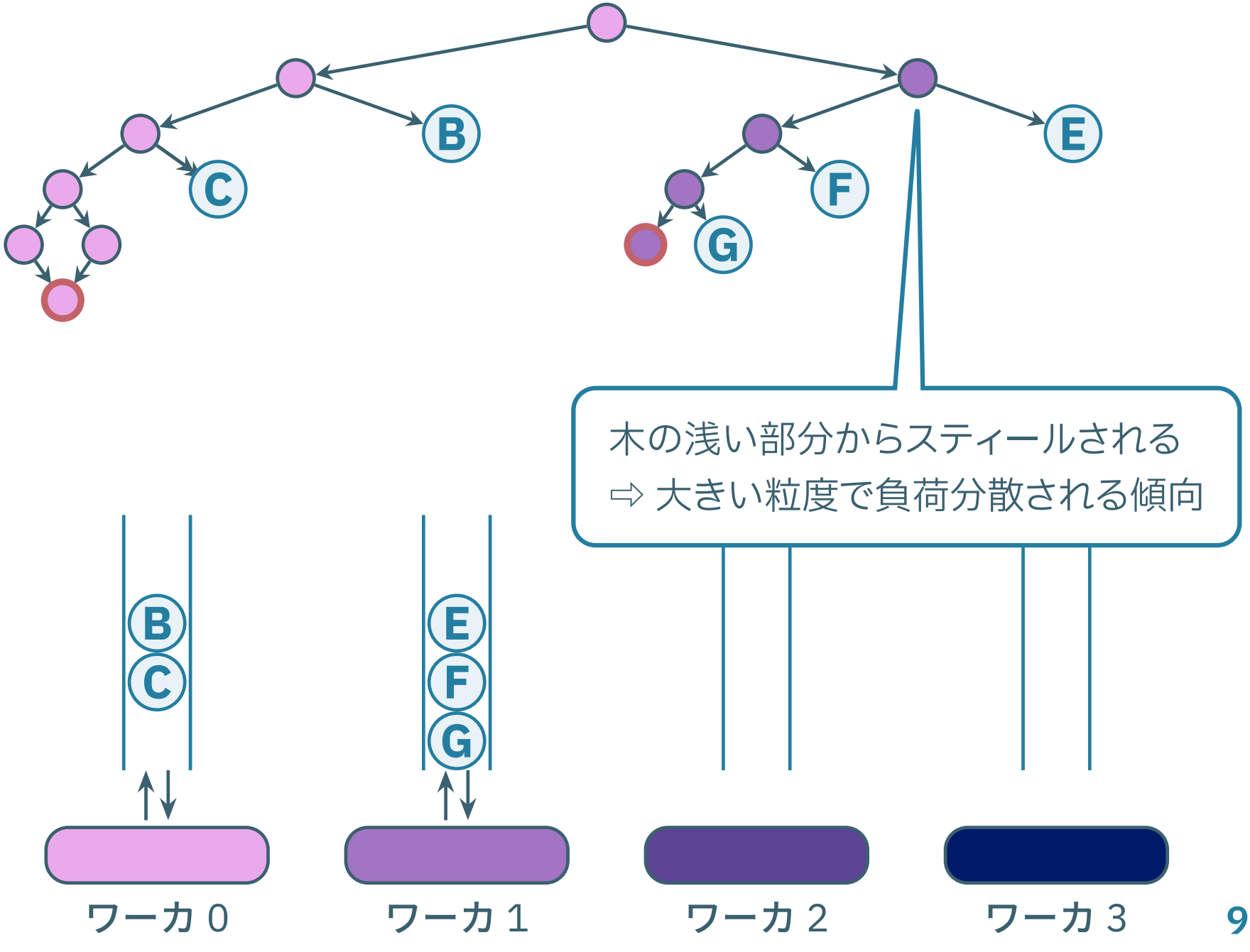


ワークスティーリングによるスケジューリング例

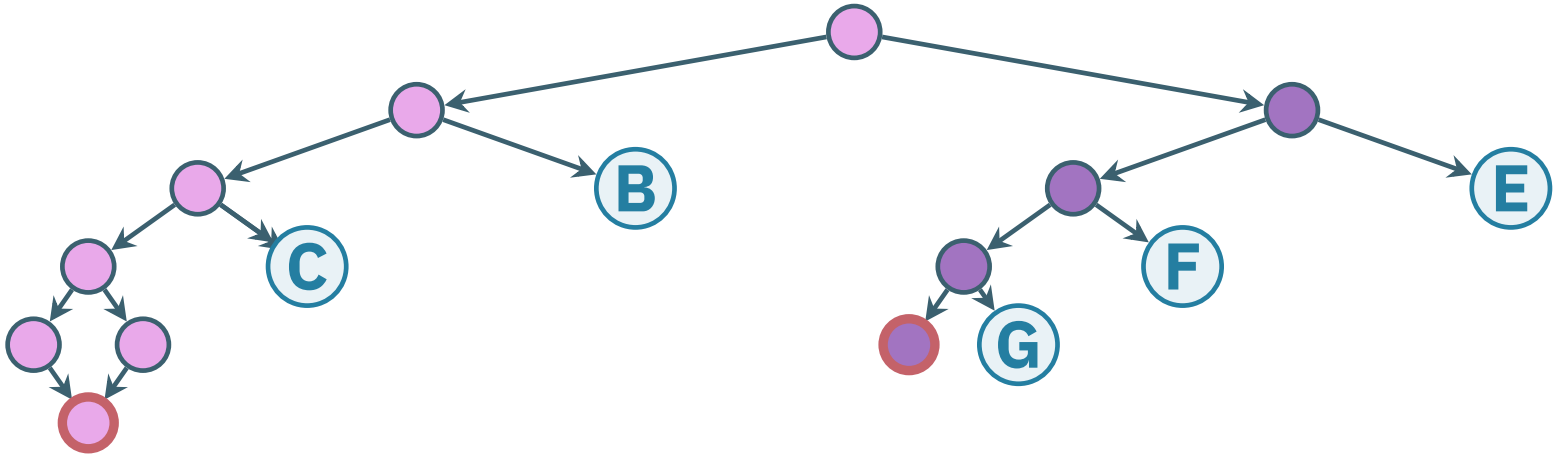


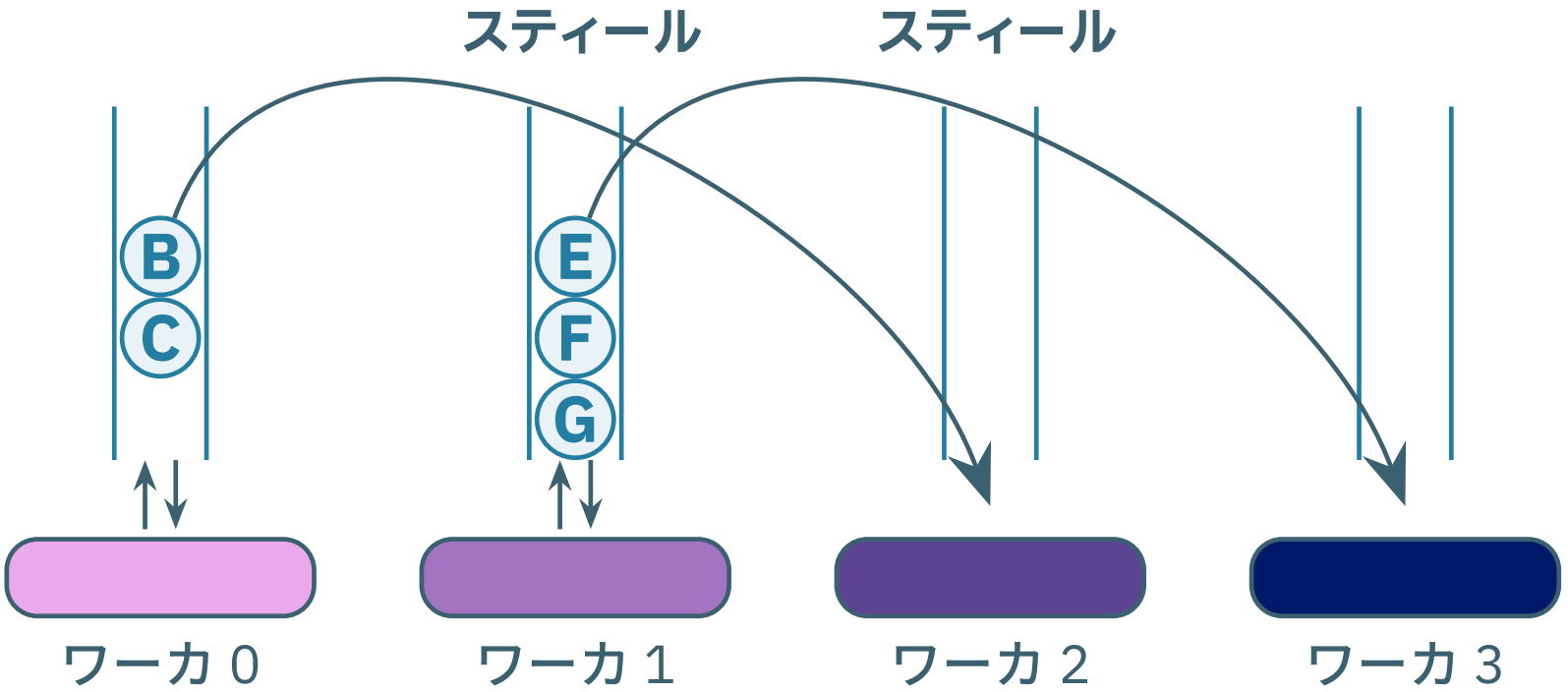
- 暇なワーカはランダムに相手を選んでスティール
- スティールはキューの逆側から（FIFO 順序）

ワークスティーリングによるスケジューリング例

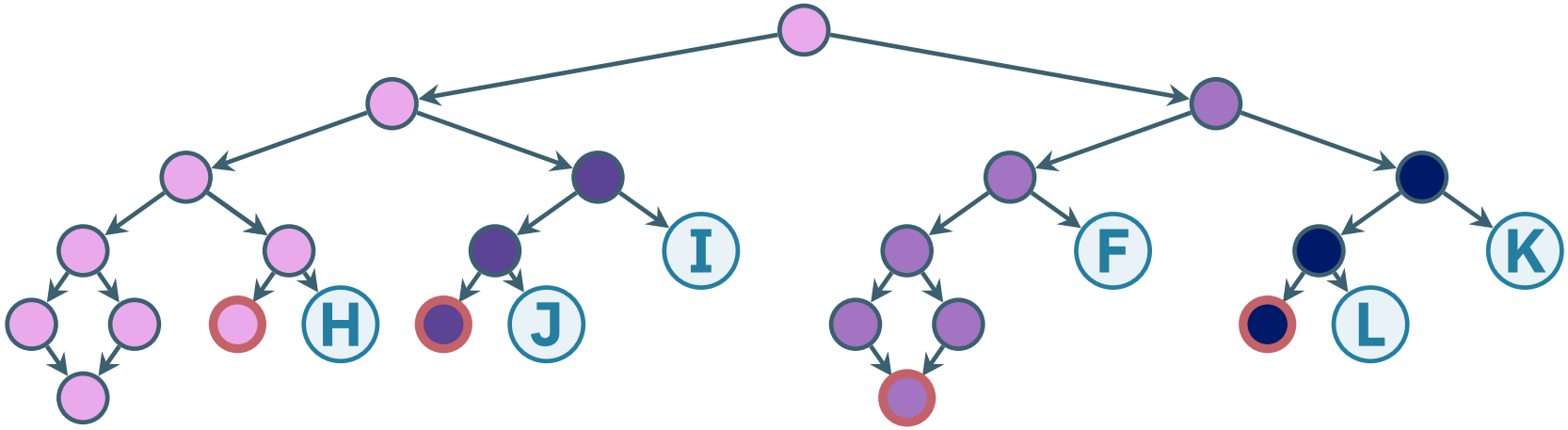


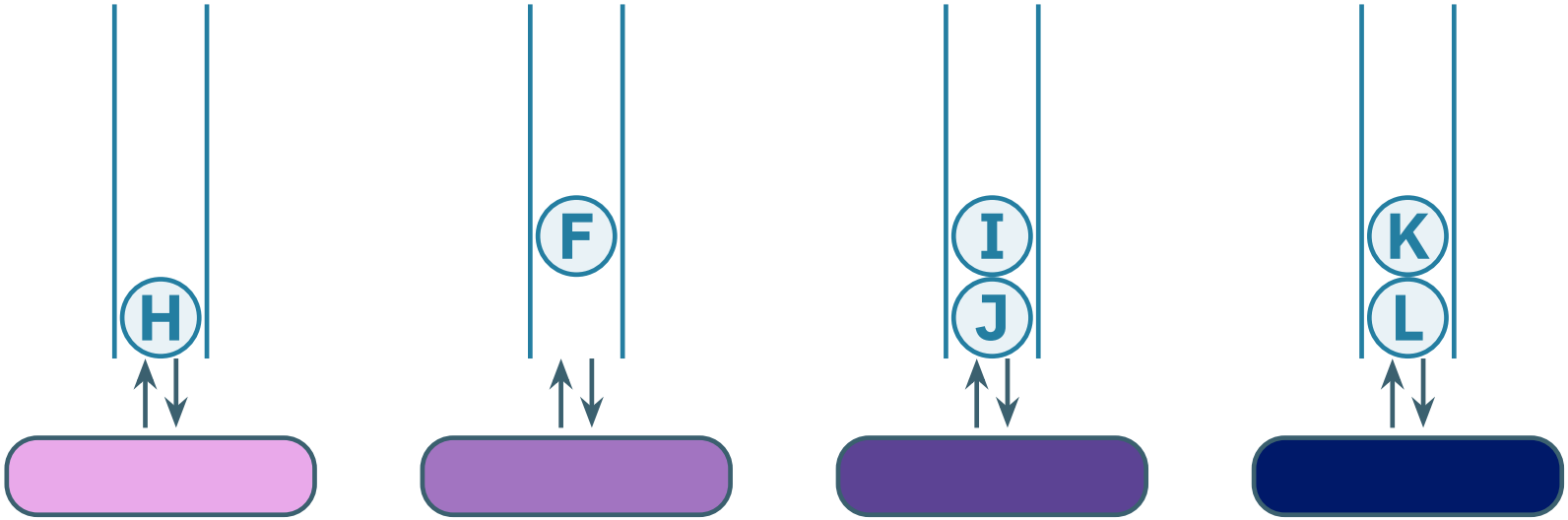
ワークスティーリングによるスケジューリング例



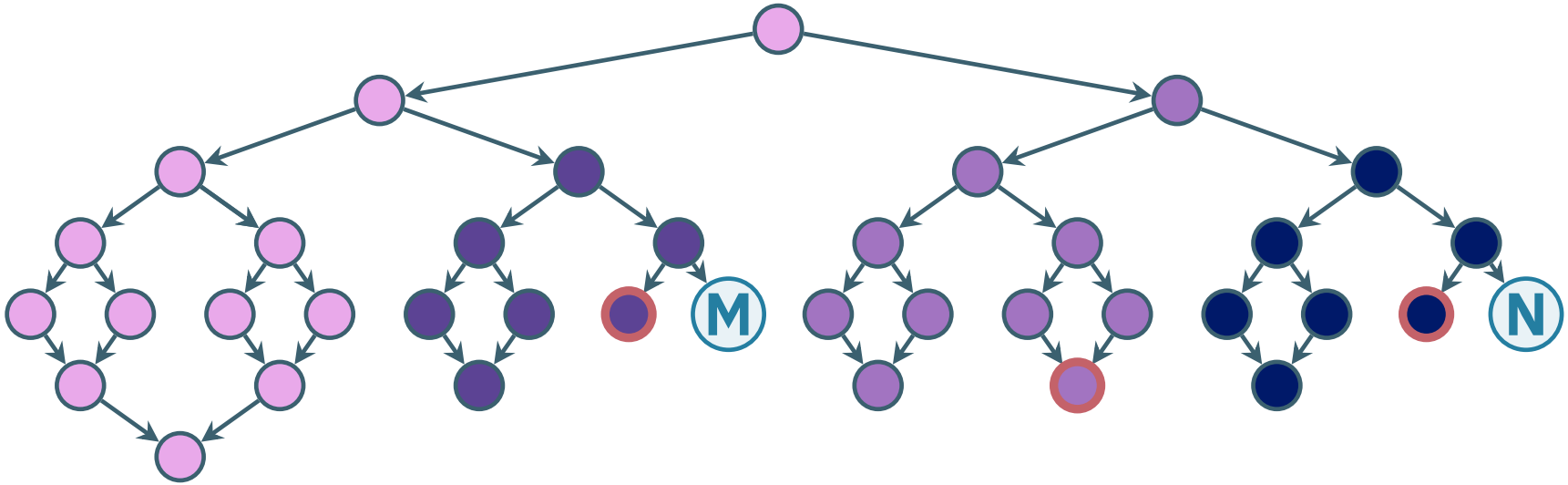


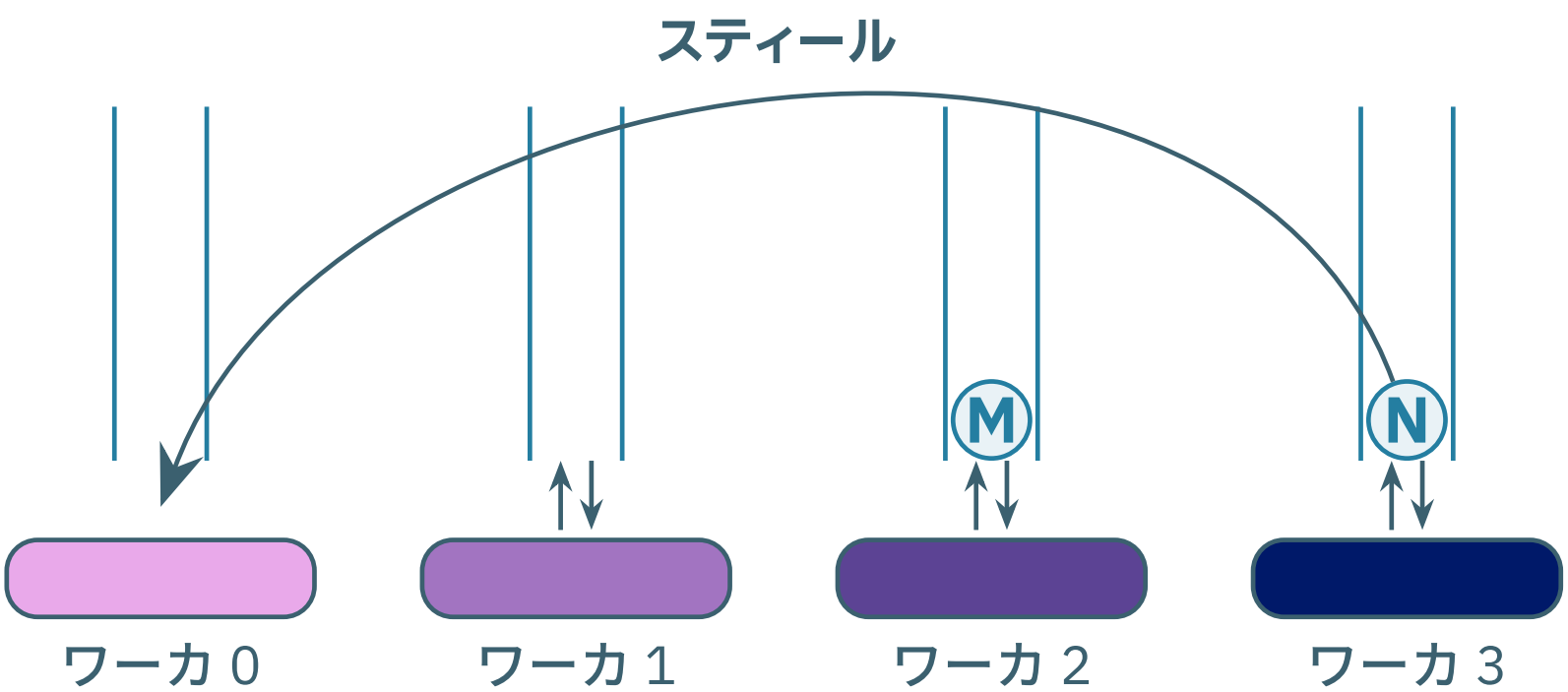
ワークスティーリングによるスケジューリング例



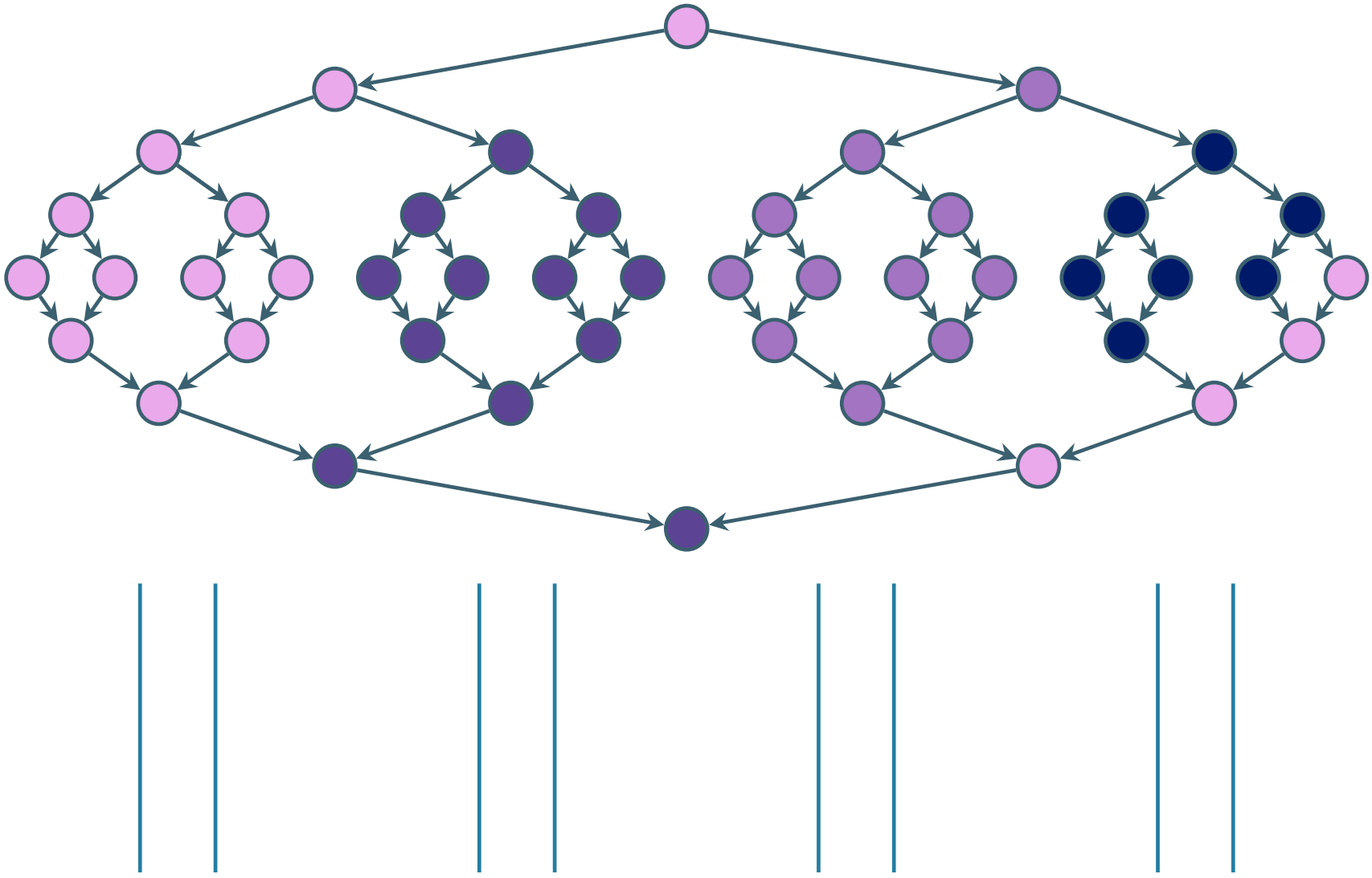


ワークスティーリングによるスケジューリング例

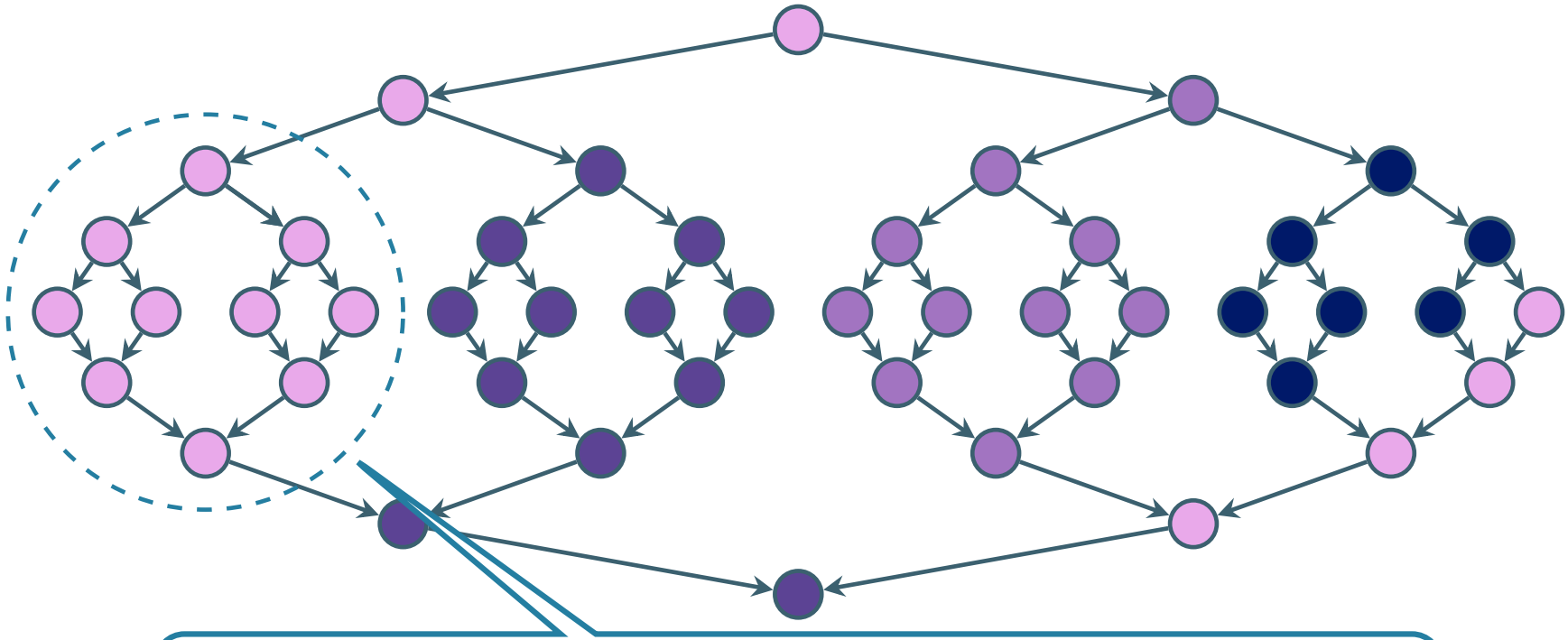




ワークスティーリングによるスケジューリング例



ワークスティーリングによるスケジューリング例



近隣のタスクは同じワーカが実行する傾向

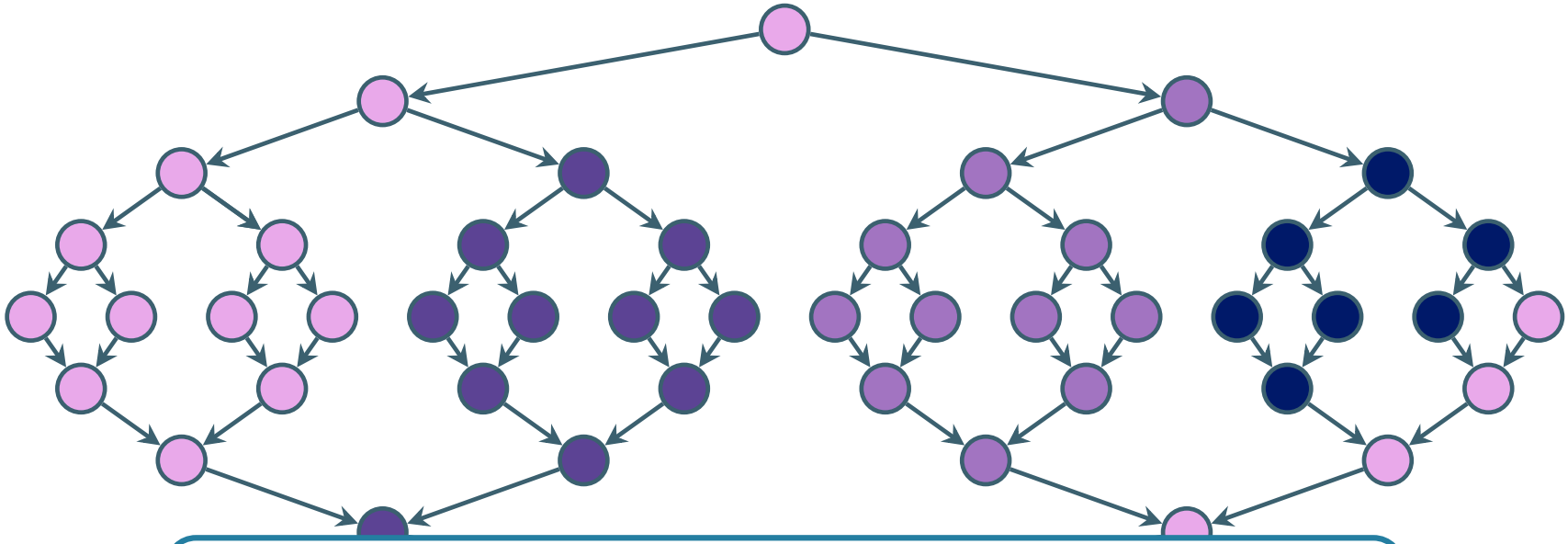
⇨ しかし各タスクは独立に大域メモリアクセスする必要

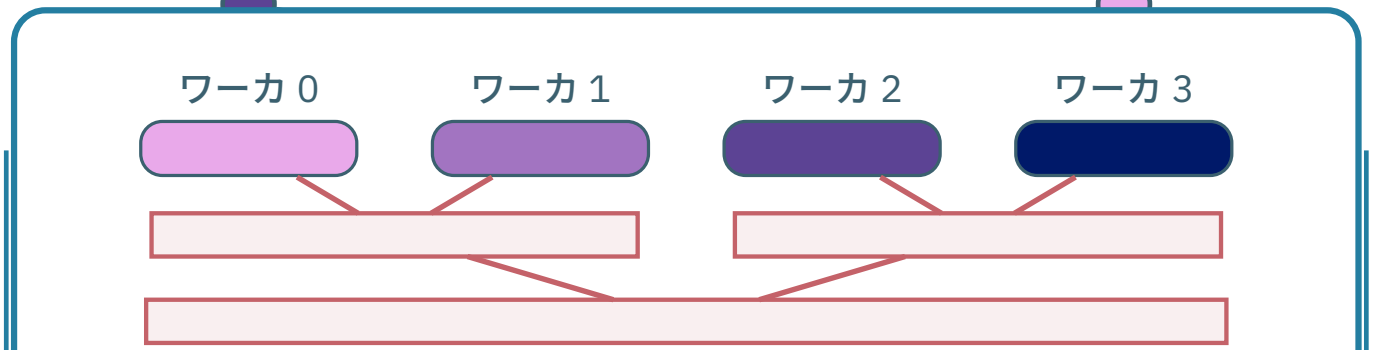
⇨ これらの大域メモリアクセスの局所性を活かしたい

（**局所性の問題 1**）

ワーカ 0　ワーカ 1　ワーカ 2　ワーカ 3

ワークスティーリングによるスケジューリング例





メモリ階層を意識したタスク配置にはならない

（**局所性の問題 2**）

本研究の貢献の概要

1. **PGAS にキャッシュを導入**することで冗長な通信を削減
   - 複数のタスクをまたいだキャッシュの共有
   - 昨年の発表時より効率的なキャッシュ実装で、API も少し異なる
2. 深いメモリ階層向けのタスクスケジューラ **Almost Deterministic Work Stealing (ADWS)** を Itoyori に実装（共有メモリ → 分散メモリへの移植）
   - 深いメモリ階層（階層キャッシュ、NUMA、分散メモリ）における局所性の向上
   - タスクの配置をデータの位置に対応付ける

大域メモリアクセスを多用する現実的な
アプリケーションで高いスケーラビリティ、効率を達成

Outline

1. ソフトウェアキャッシュを備える PGAS

2. 局所性に配慮したタスクスケジューラ ADWS

性能評価

まとめ

Outline

1. ソフトウェアキャッシュを備える PGAS

2. 局所性に配慮したタスクスケジューラ ADWS

性能評価

まとめ

Itoyori 向けに提案した Checkout/Checkin API

通常の仮想アドレスを
大域アドレスとして使用

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a      , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    merge(a, n, n/2);
  }
}
```

Itoyori 向けに提案した Checkout/Checkin API

通常の仮想アドレスを
大域アドレスとして使用

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a      , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    merge(a, n, n/2);
  }
}
```

- 以降、大域メモリ領域 $[a, a+n)$ へのローカルアクセスが可能
- read、read_write、write のいずれかのモードを指定
  - read、read_writeならば最新のデータが読み込まれる

Itoyori 向けに提案した Checkout/Checkin API

通常の仮想アドレスを
大域アドレスとして使用

```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a      , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    merge(a, n, n/2);
  }
}
```

- 以降、大域メモリ領域 $[a, a+n)$ へのローカルアクセスが可能
- read、read_write、write のいずれかのモードを指定
  - read、read_writeならば最新のデータが読み込まれる

- 大域メモリアクセスの終了
- 指定する引数は対応するチェックアウト呼び出しと同じ
  - read_write、writeならば書き込みをグローバルに反映

PGAS へのキャッシュ実装の概要

- リモートメモリのデータをキャッシュすることで冗長な通信を削減
  - 大域メモリアクセスの時間的・空間的局所性を活用
- 物理メモリは仮想（大域）アドレス空間に動的にマッピング
  - キャッシュサイズは一定、LRUで追い出し
- キャッシュコヒーレンスはスケジューラと協調して効率的に管理
  - 同一プロセスで実行される複数のタスクでキャッシュを共有可能
  - データ競合のない（data-race-free）プログラムを仮定
- キャッシュ実装の詳細は SC23 で発表予定

Outline

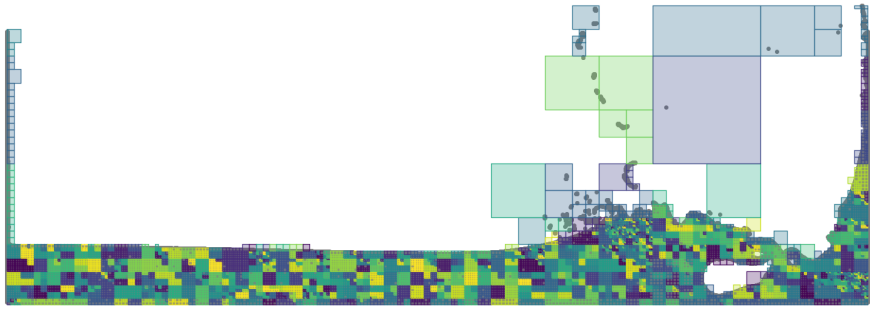
1. ソフトウェアキャッシュを備える PGAS

2. 局所性に配慮したタスクスケジューラ ADWS

性能評価

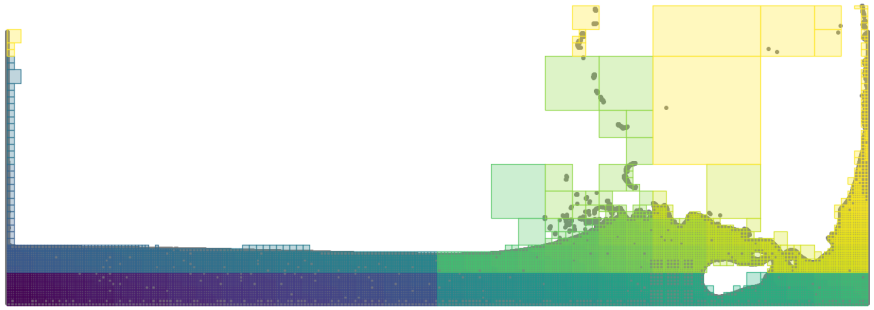
まとめ

均等な負荷分散と優れた局所性の両立



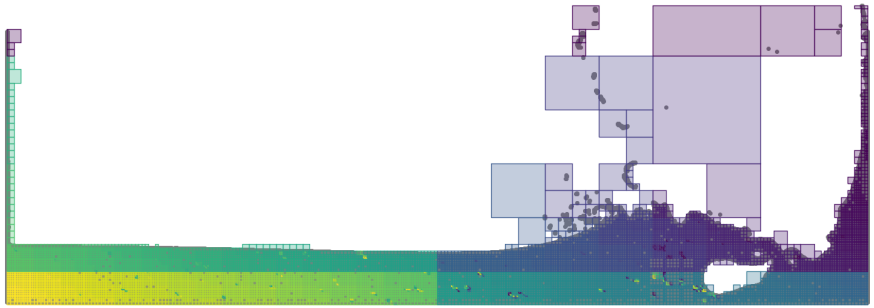
ランダムワークスティーリング

- 負荷分散◎
- 局所性 ×



決定的な負荷分散

- 局所性◎
- 粒子数が均等でも
  負荷が均等とは限らない



「ほぼ」決定的な負荷分散

- 負荷分散と局所性の両立

Almost Deterministic Work Stealing (ADWS) ▷ 概要

（ループ並列に限らない）タスク並列処理に対し
静的・動的負荷分散のハイブリッド

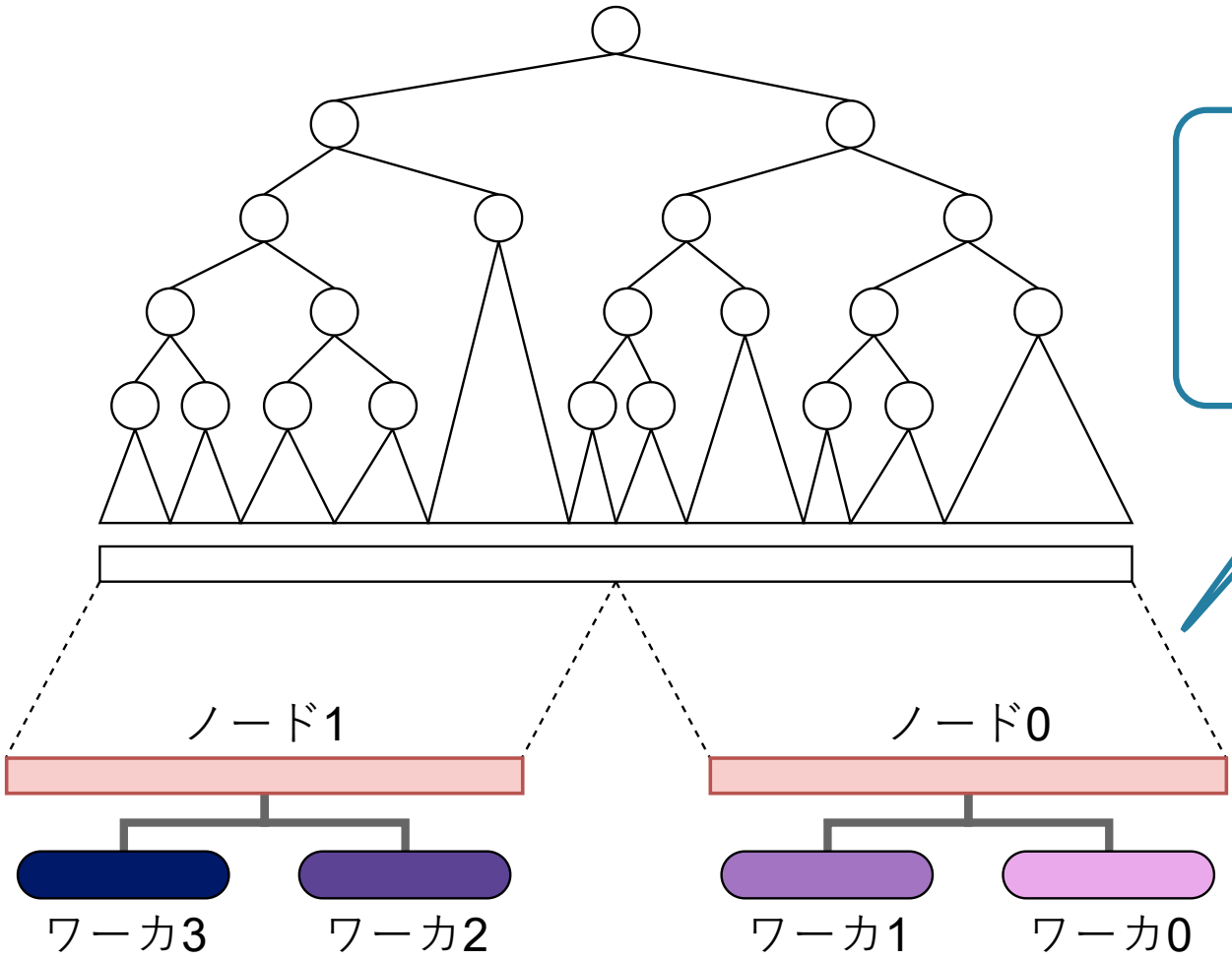
- **決定的なタスク配置**により任意のメモリ階層で局所性の向上
  - タスクの階層性をメモリの階層性に対応付ける
- **局所的な動的負荷分散**で負荷の不均衡を動的に調整
  - 可能な限り局所性を損ねないように
- 子タスクの仕事量の比のヒントが必要
  - バランスがあまり悪くない場合は勝手に比が等しいと推測してもうまく行く
- 共有メモリ ADWSは SC19[1] で発表、+α の内容を TPDS[2] で出版済

---

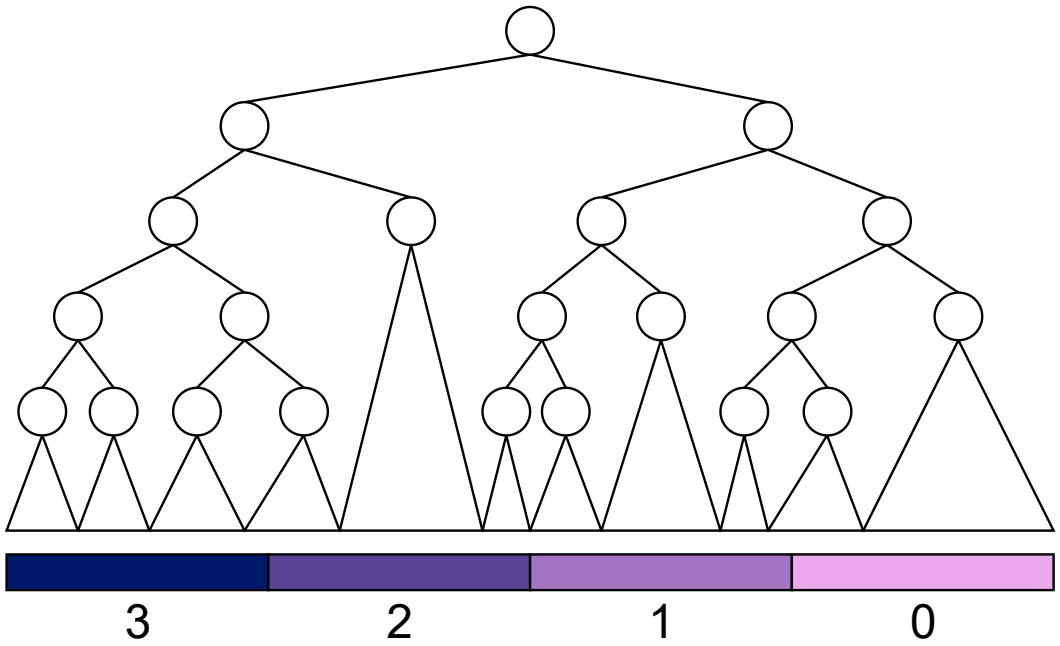
1 Shumpei Shiina and Kenjiro Taura. “Almost Deterministic Work Stealing”. In: SC ’19. 2019.

2 Shumpei Shiina and Kenjiro Taura. “Improving Cache Utilization of Nested Parallel Programs by Almost Deterministic Work Stealing”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022).

ADWS の基本的なアイデア

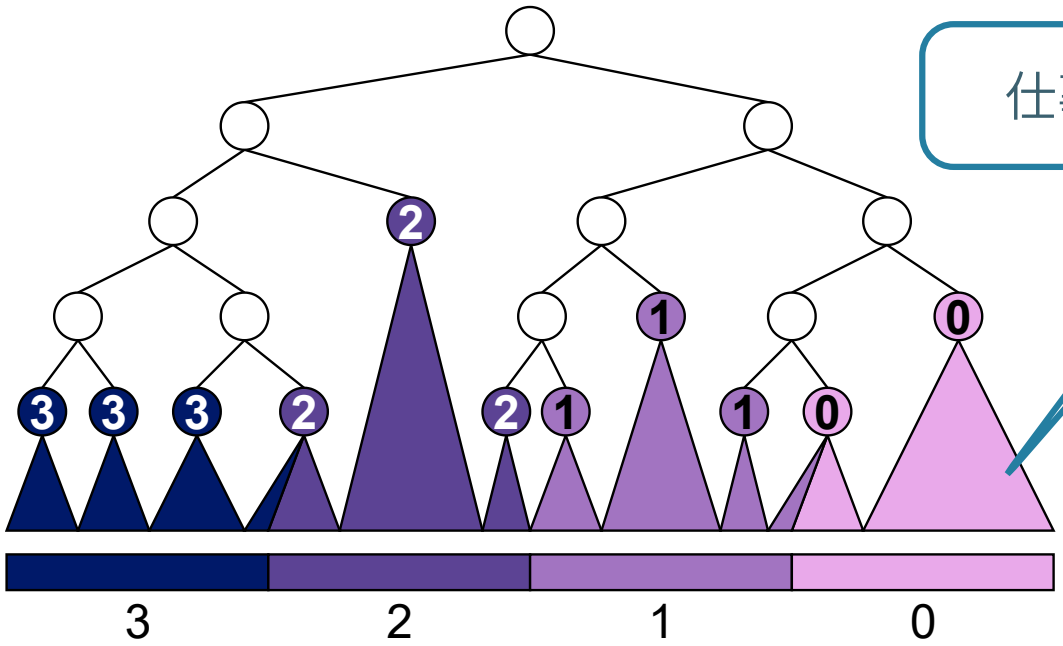


ADWS の基本的なアイデア



メモリの階層性を平坦化し
ワーカの数直線として近似的に表現

ADWS の基本的なアイデア

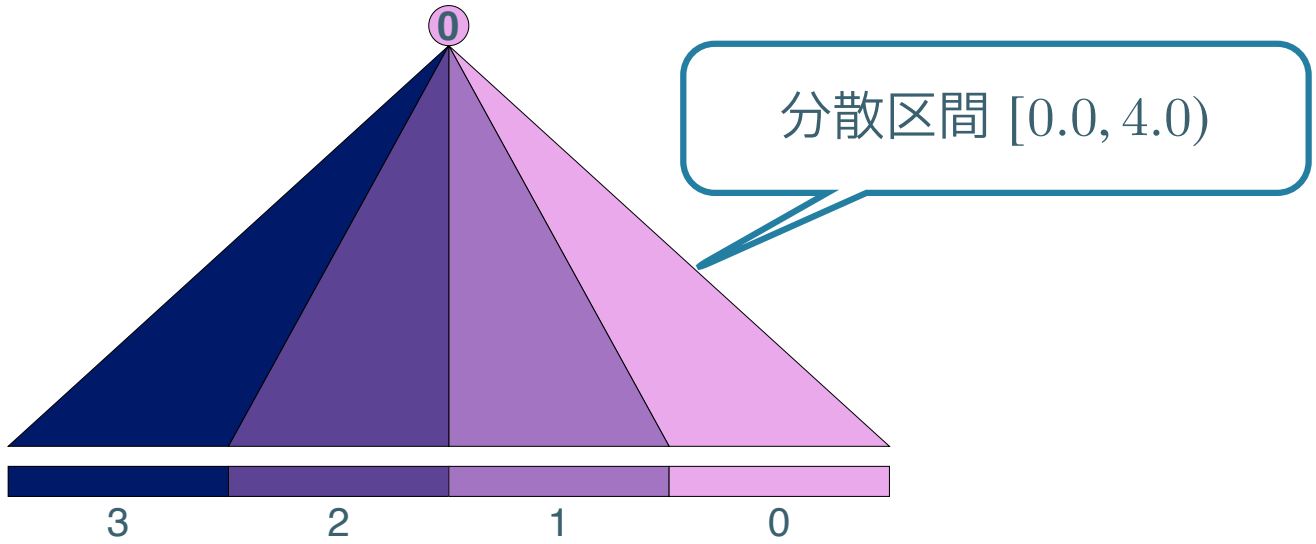


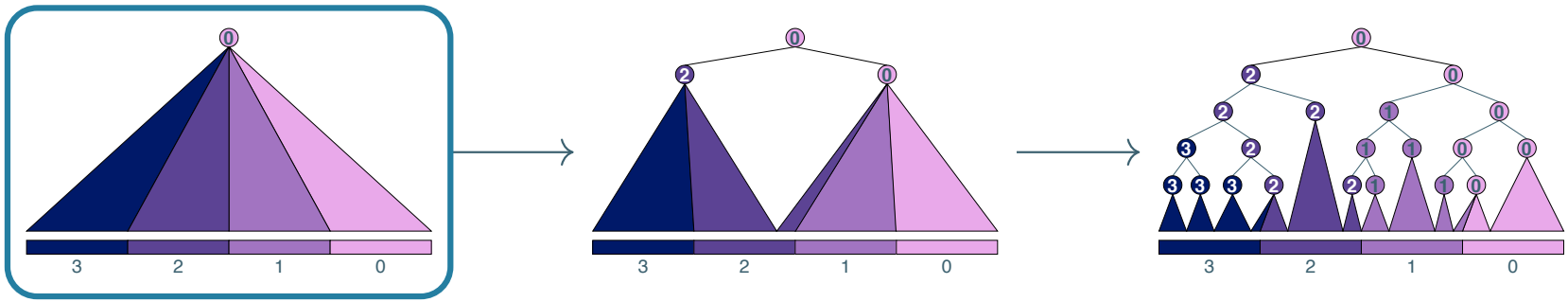
仕事量に比例した幅を持つ

一直線上に並んだタスクを
仕事量が均等になるようワーカに割り振る

ADWS の決定的なタスク配置

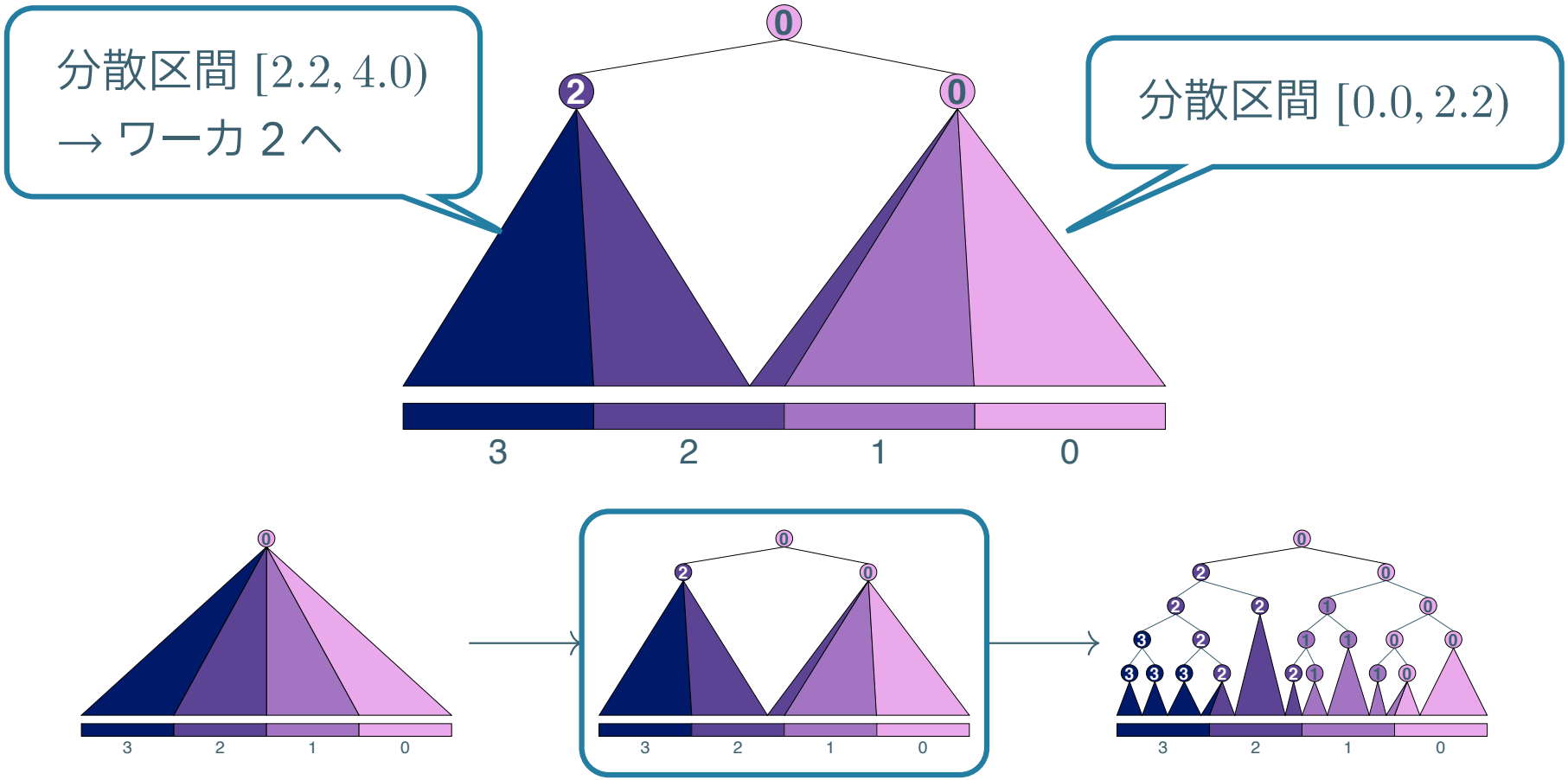
- 4ワーカのうちワーカ 0が根タスクの実行を開始
- **分散区間**は子孫タスクを配置すべきワーカの範囲





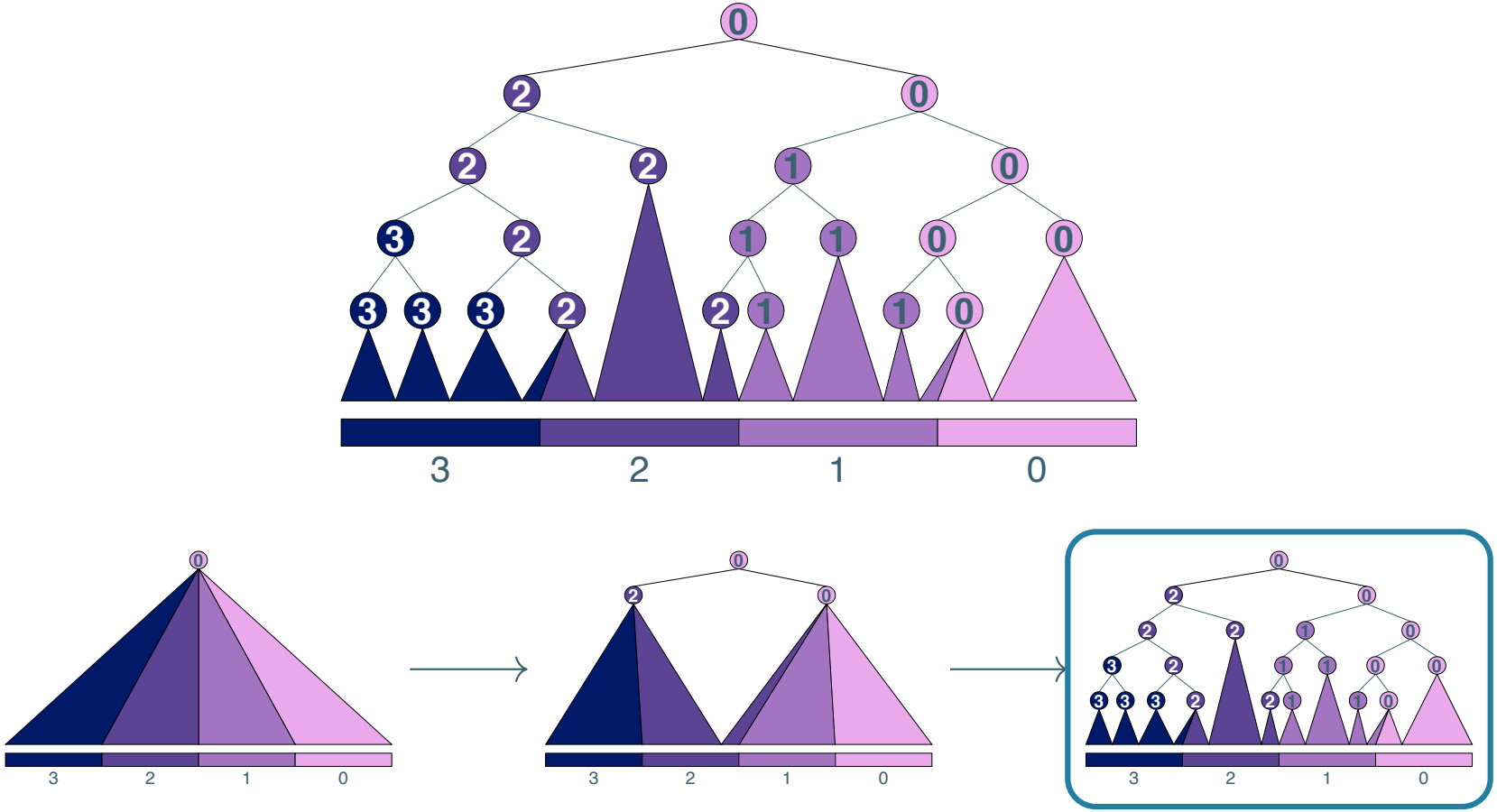
ADWS の決定的なタスク配置

- ヒントとして与えられた仕事量の比で分散区間を分割する
- 分散区間内の最も若い番号のワーカにタスクを割り当てる



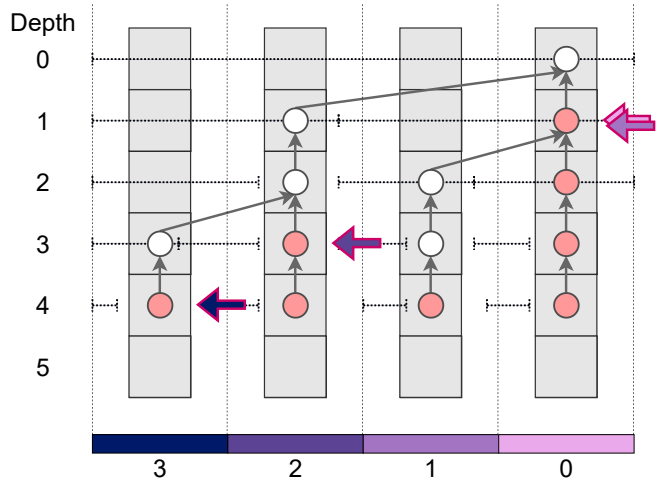
ADWS の決定的なタスク配置

- このようにタスク配置が非同期に決定されていく
- タスク階層を活用して可能な限り近いワーカ同士でタスクをスティールし合う（局所的な動的負荷分散）

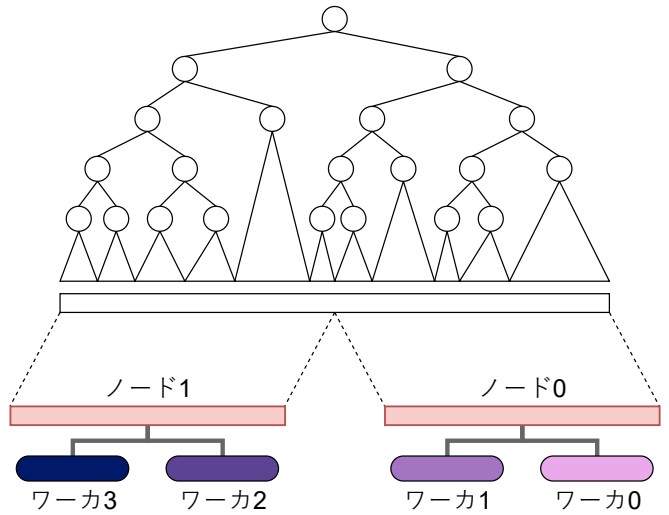


ADWS の Itoyori への実装

- ADWS は当初、共有メモリ向けに提案、実装、評価を行っていた
- 通信の輻輳を避けるため分散メモリ特有の最適化を行った
  - 主に局所的なワークスティーリングのためのデータ構造（詳細は予稿に）
- 大域メモリの分散配置ポリシーを ADWS のスケジューリングに合わせる



スティール範囲の木構造の管理



ADWS に合わせた大域メモリの配置

Outline

1. ソフトウェアキャッシュを備える PGAS

2. 局所性に配慮したタスクスケジューラ ADWS

性能評価

まとめ

Itoyori の性能評価

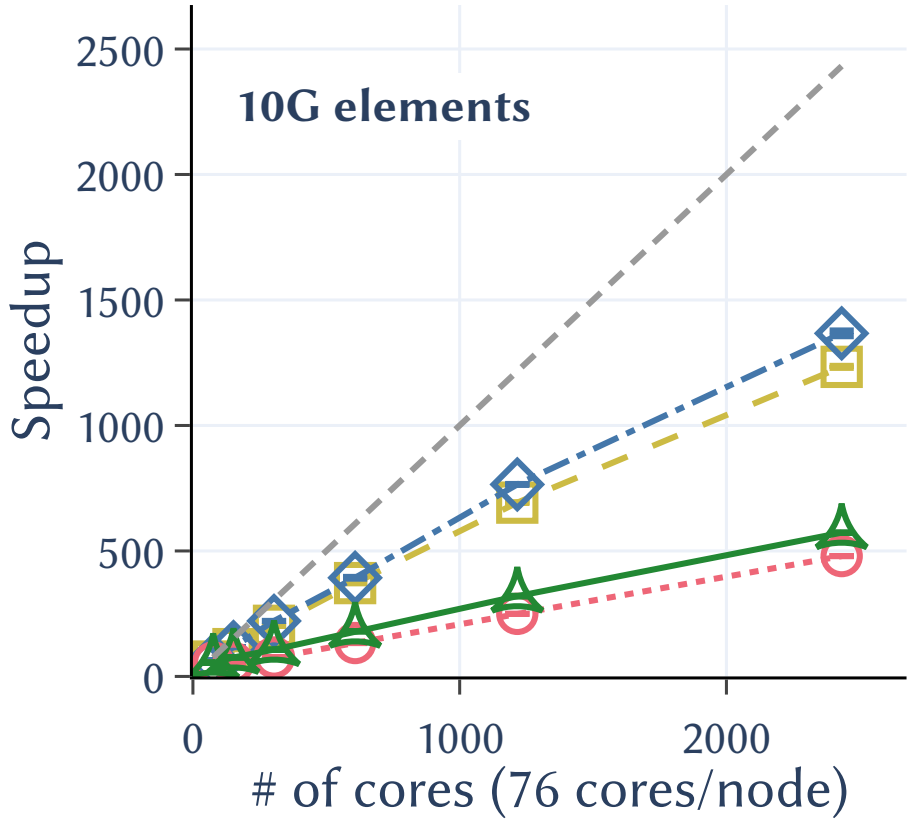
- 2 つの Fork-Join プログラムを共有メモリから分散メモリに移植:
  - **Cilksort:** 並列マージソート
  - **ExaFMM:** 高性能な N 体問題ソルバ
- 局所性の活用による性能向上を重点的に評価:
  - 大域メモリアクセスへのキャッシュの導入
  - ADWS によるスケジューリング
- 実験環境:
  - 大阪大学 **SQUID** スーパコンピュータ（汎用 CPUノード群）
    - Intel Xeon CPU と InfiniBand の標準的な構成
  - 東京大学 **Wisteria/BDEC-01 Odyssey** スーパコンピュータ
    - Fujitsu A64FX CPU と Tofu インターコネクト D の「富岳」と似た構成

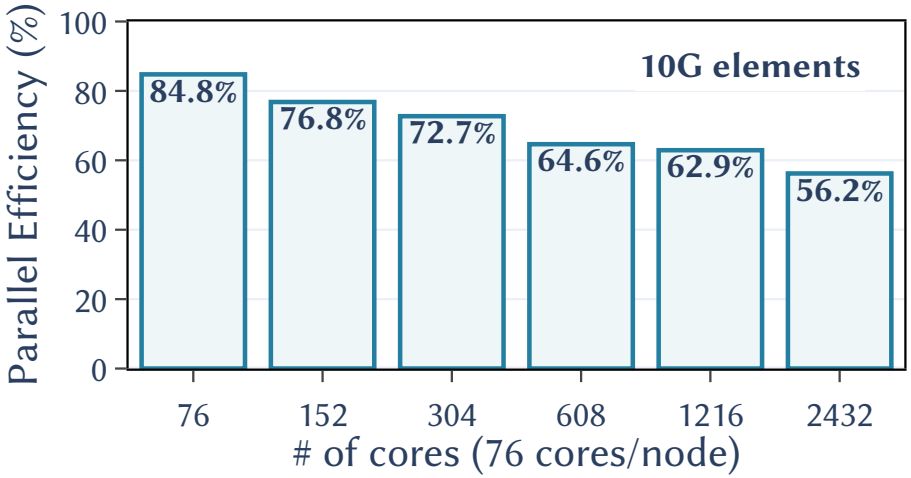
Cilksort（並列マージソート）

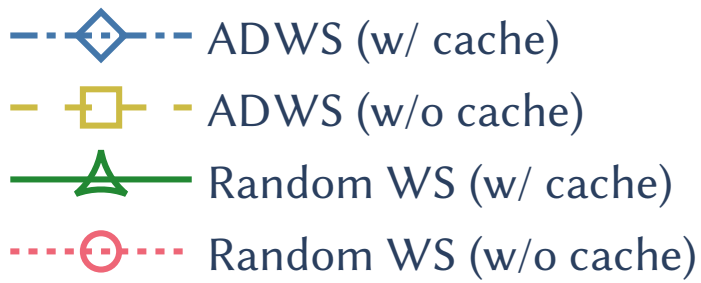
```
void msort(int* a, size_t n) {
  if (n < CUTOFF) {
    checkout(a, n, mode::read_write);
    sort_small(a, n);
    checkin(a, n, mode::read_write);
  } else {
    parallel_invoke(
      [=]{ msort(a      , n/2); },
      [=]{ msort(a + n/2, n/2); }
    );
    // マージ処理も内部で再帰的に並列化
    merge(a, n, n/2);
  }
}
```

- タスク並列処理系 Cilk の評価に用いられたベンチマーク
- 並列マージソートのマージ処理の内部も並列化したもの
- Itoyori に移植して評価を行った

Cilksort の台数効果 (Strong Scaling)



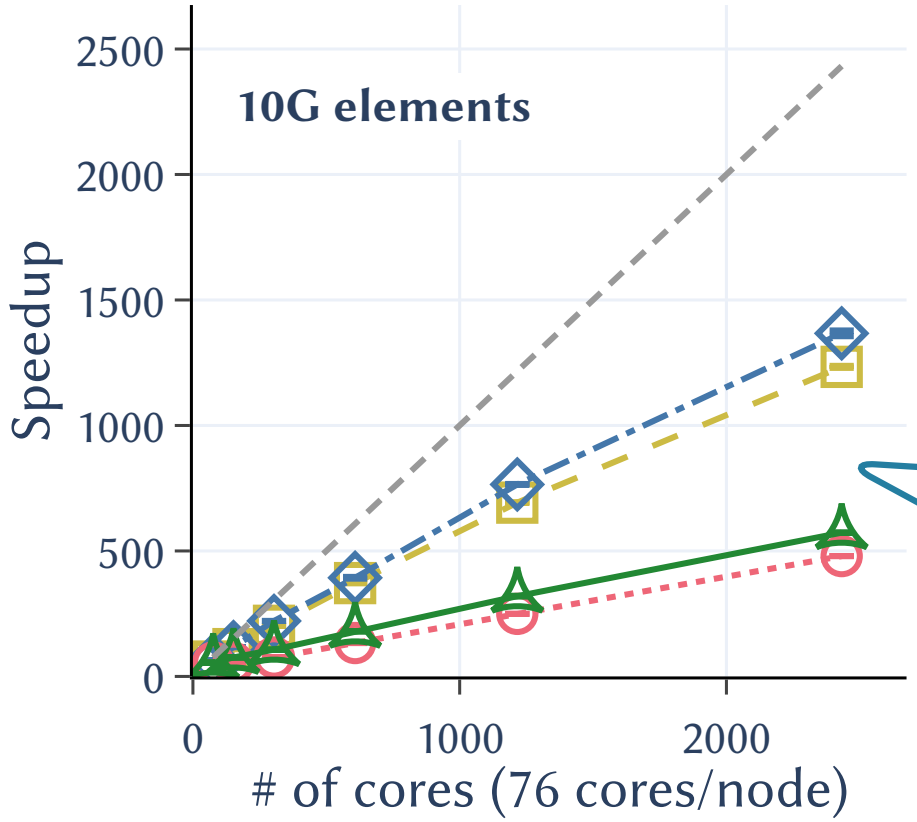




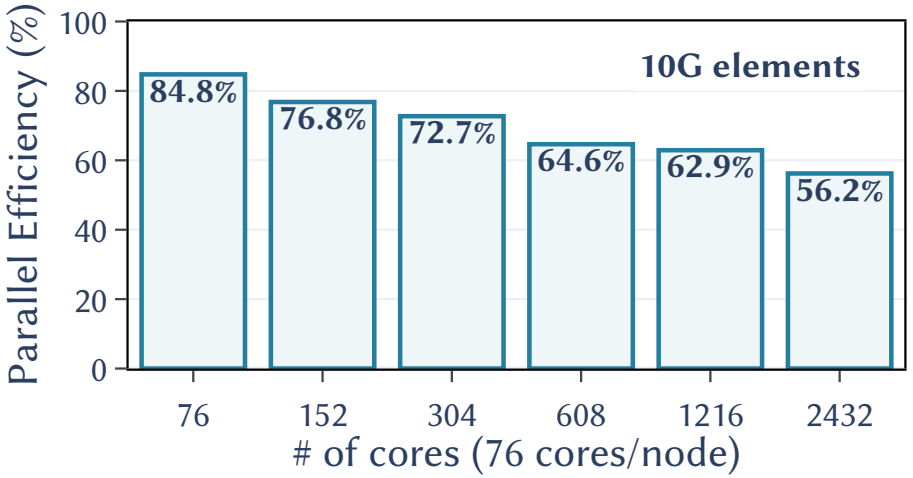
実験設定:

- **SQUID** 上で実験
- 破線の直線: 逐次実行時間をもとにした理想的な台数効果
- 最適な性能となるよう充分大きいカットオフ (16384 要素) を用いた
- カットオフ: 逐次実行に切り替える閾値 (配列の要素数)

Cilksort の台数効果 (Strong Scaling)



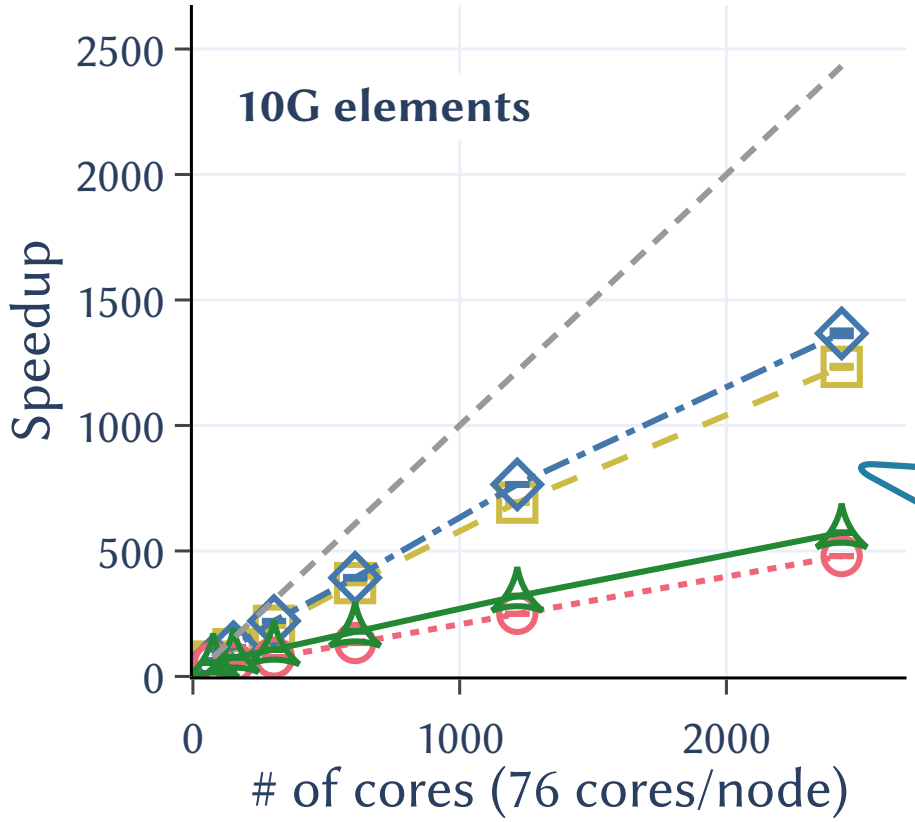
- ADWS の導入で **×2.4 倍**
- キャッシュの導入で ×1.2 倍



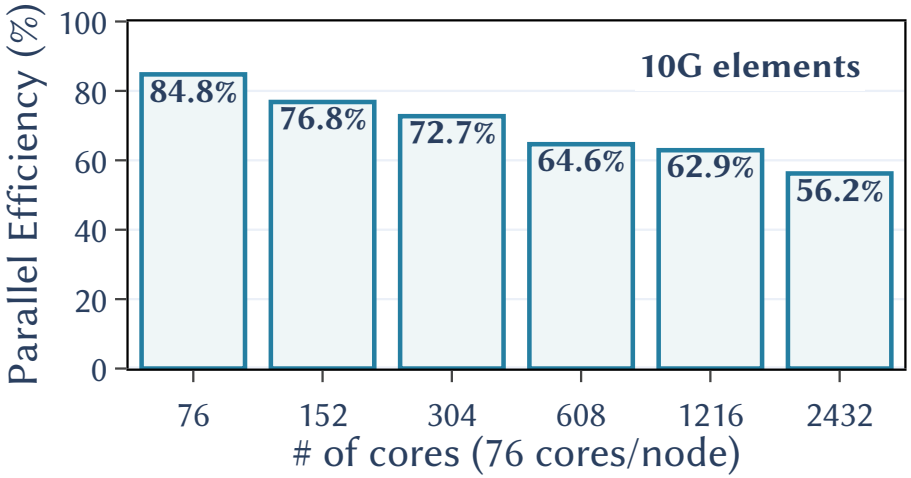
実験設定:

- **SQUID** 上で実験
- 破線の直線: 逐次実行時間をもとにした理想的な台数効果
- 最適な性能となるよう充分大きいカットオフ (16384 要素) を用いた
- カットオフ: 逐次実行に切り替える閾値 (配列の要素数)

Cilksort の台数効果 (Strong Scaling)



- ADWS の導入で **×2.4 倍**
- キャッシュの導入で ×1.2 倍

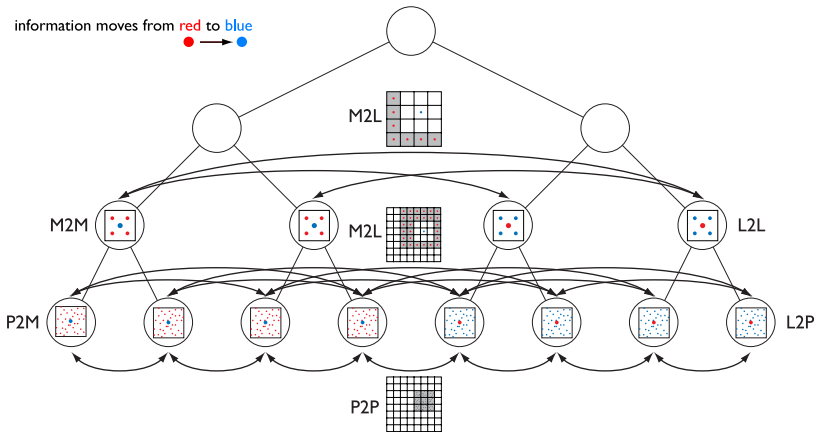


実験設定:

- **SQUID** 上で実験
- 逐次実行と比べた並列化効率
- 2432 コアで 56.2%
- カットオフ: 逐次実行に切り替える閾値（配列の要素数）

ExaFMM（高性能な FMM ライブラリ実装）

- **ExaFMM**の Fork-Join 実装[1]を Itoyori を用いて分散メモリに移植
  - 木構造を用いて遠い粒子間の計算を近似 → **不規則な計算**
- 移植は細かい変更のみ、**大枠としての並列アルゴリズムは変更なし**
  - MPI で書き直すなら並列アルゴリズム自体を再考する必要
- **高い移植性**、では性能は?

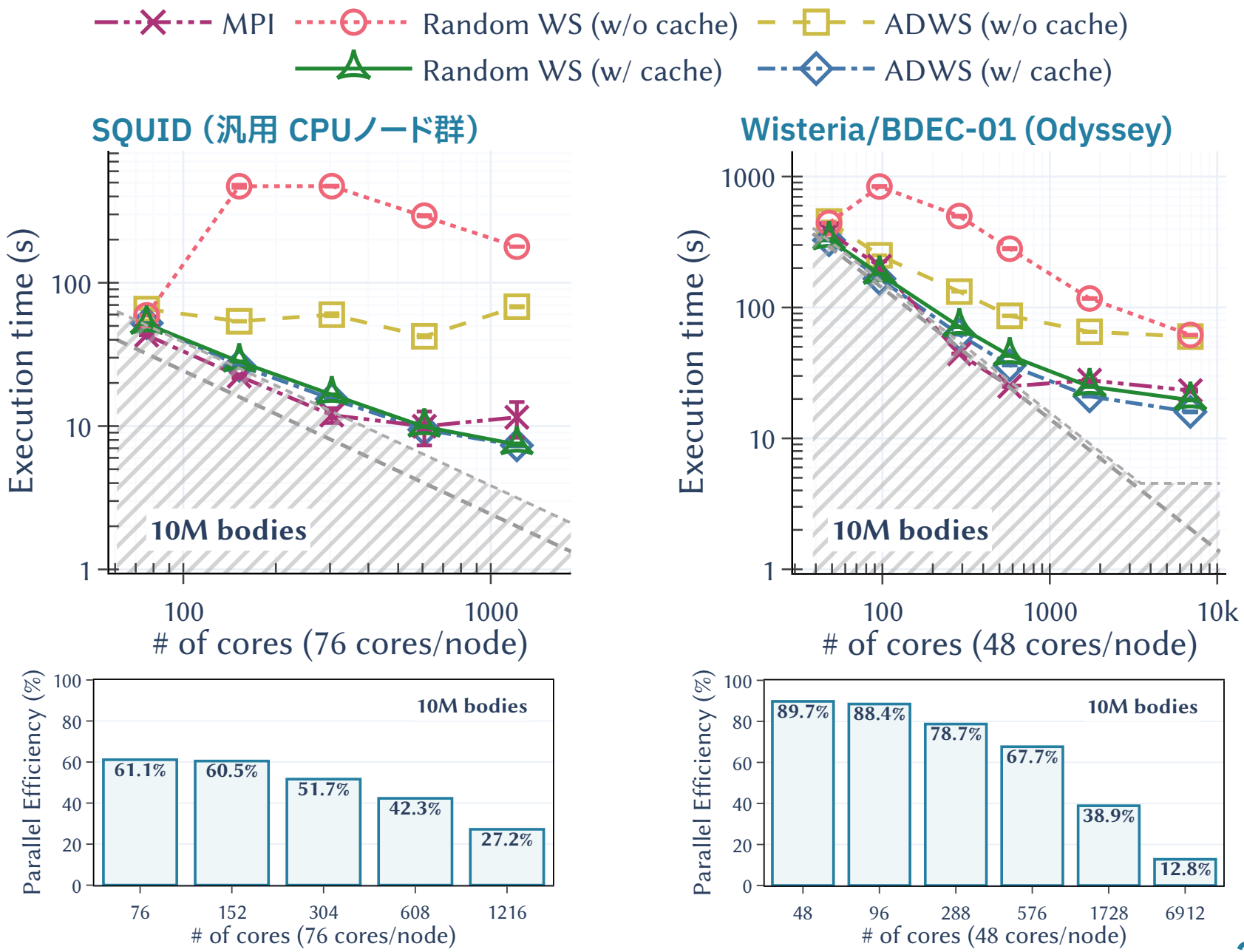


ExaFMM の木構造を用いた計算[2]

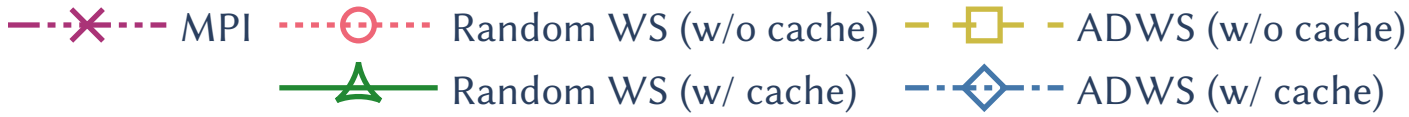
[1] Kenjiro Taura, Jun Nakashima, Rio Yokota, and Naoya Maruyama. “A Task Parallel Implementation of Fast Multipole Methods”. In: ScalA’ 12. 2012

[2] Rio Yokota, Lorena A. Barba, Tetsu Narumi, and Kenji Yasuoka. “Petascale Turbulence Simulation Using a Highly Parallel Fast Multipole Method on GPUs”. In: *Computer Physics Communications* 184.3 (2013)

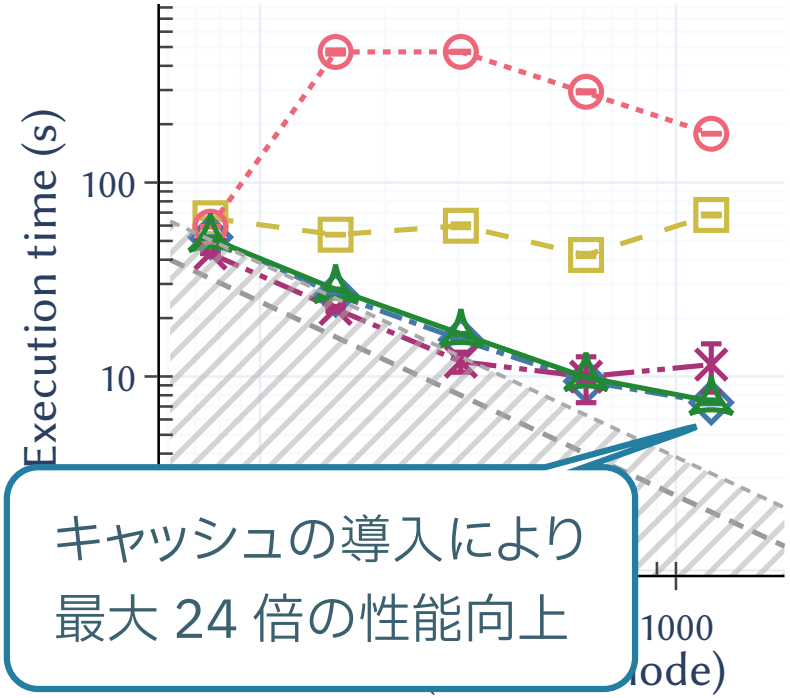
ExaFMM の性能評価 (y=実行時間、Strong Scaling)



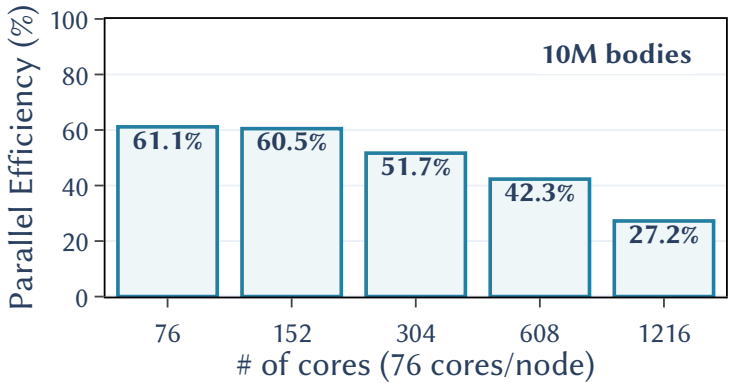
ExaFMM の性能評価 (y=実行時間、Strong Scaling)



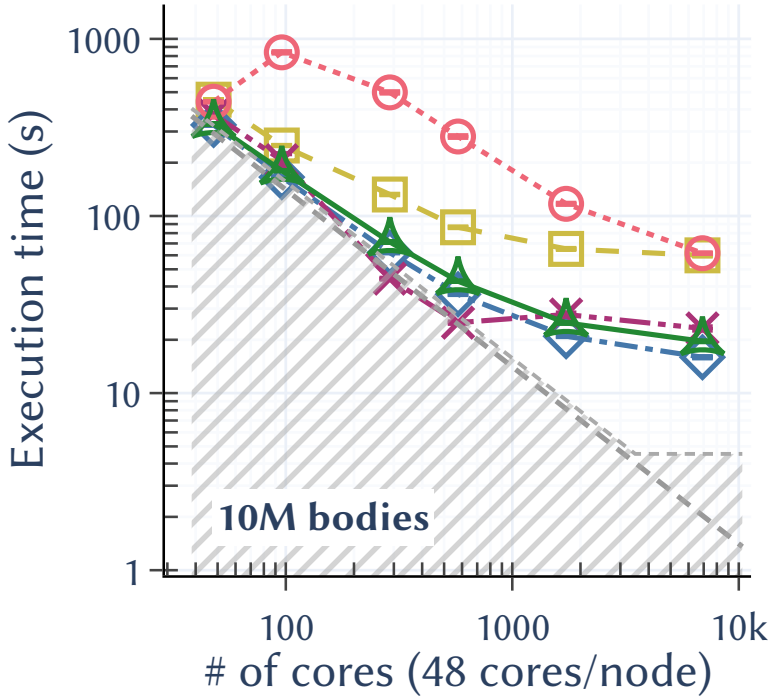
SQUID (汎用 CPUノード群)

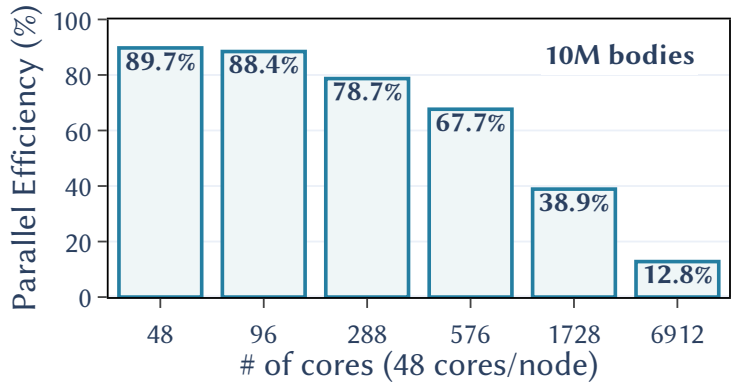


キャッシュの導入により
最大 24 倍の性能向上

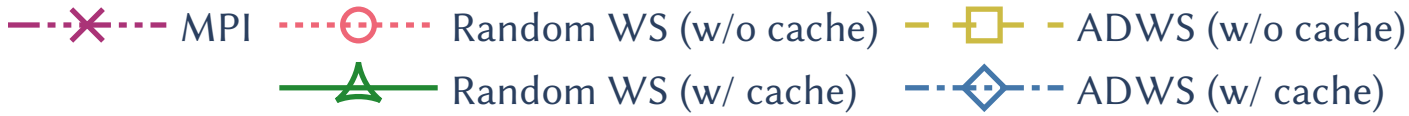


Wisteria/BDEC-01 (Odyssey)

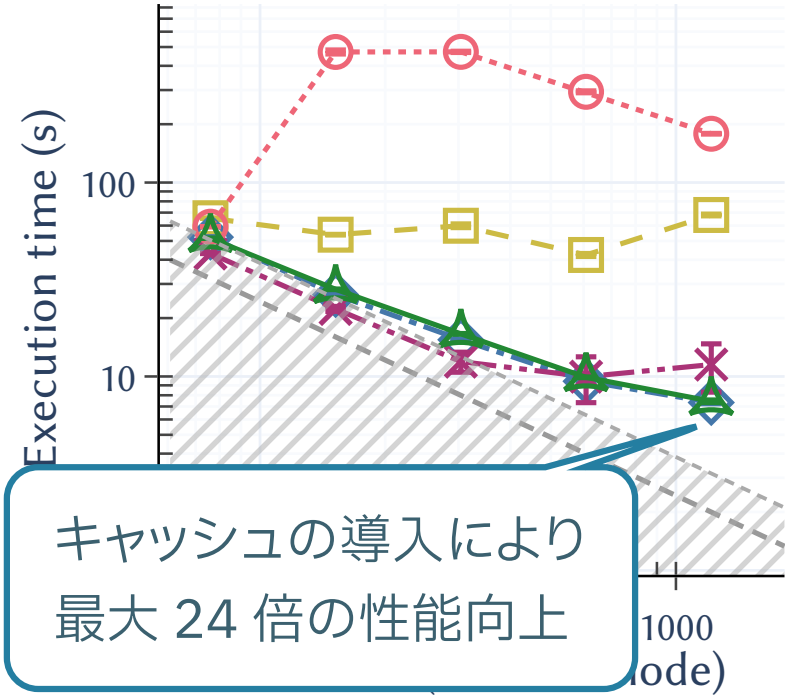




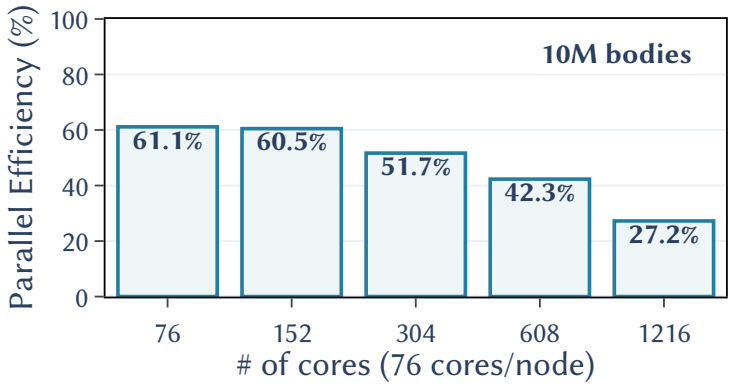
ExaFMM の性能評価 (y=実行時間、Strong Scaling)



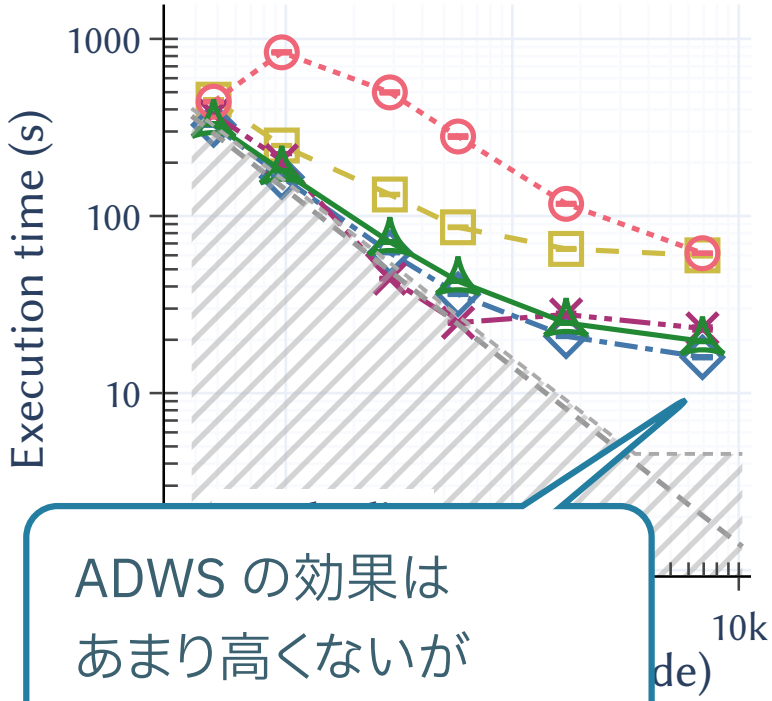
SQUID (汎用 CPUノード群)



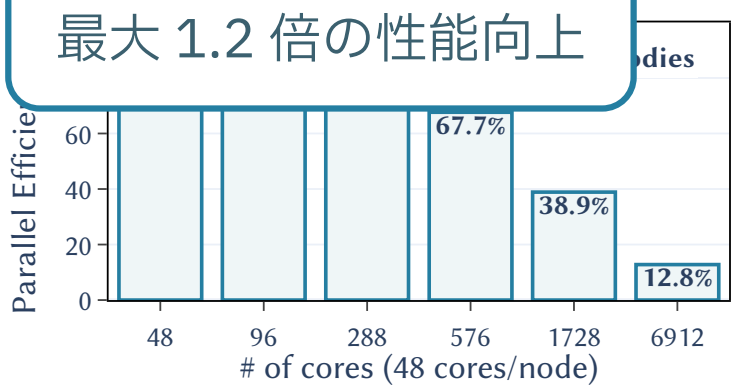
キャッシュの導入により
最大 24 倍の性能向上



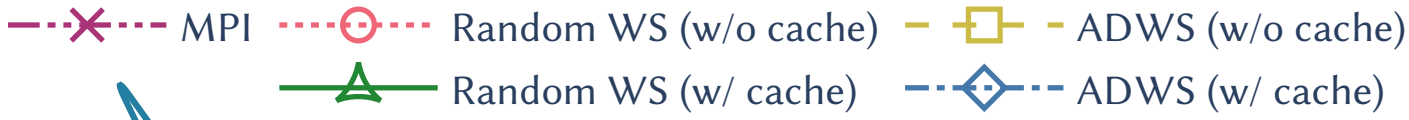
Wisteria/BDEC-01 (Odyssey)



ADWS の効果は
あまり高くないが
最大 1.2 倍の性能向上

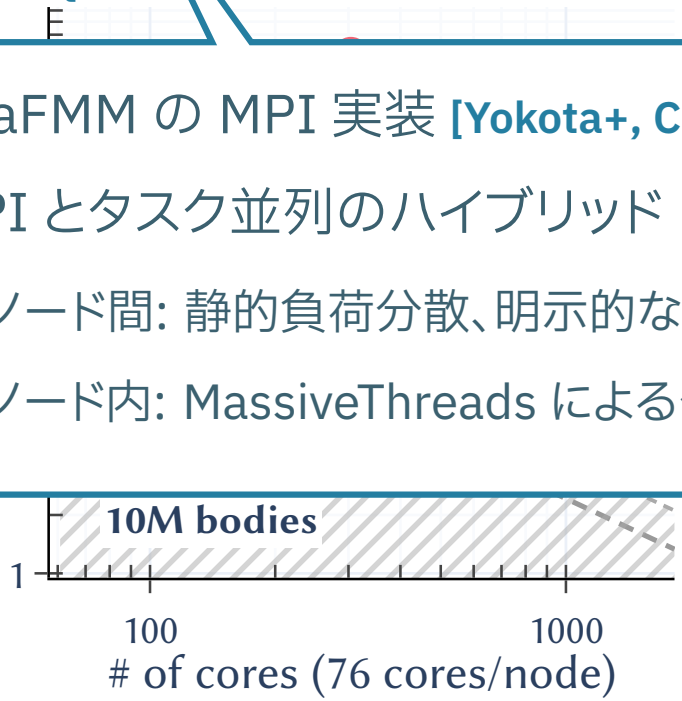


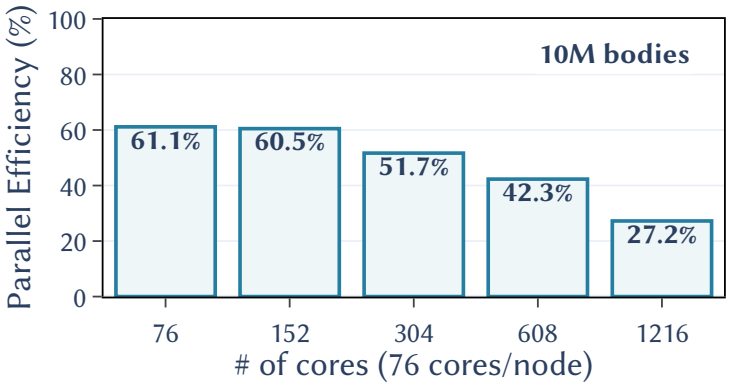
ExaFMM の性能評価 (y=実行時間、Strong Scaling)



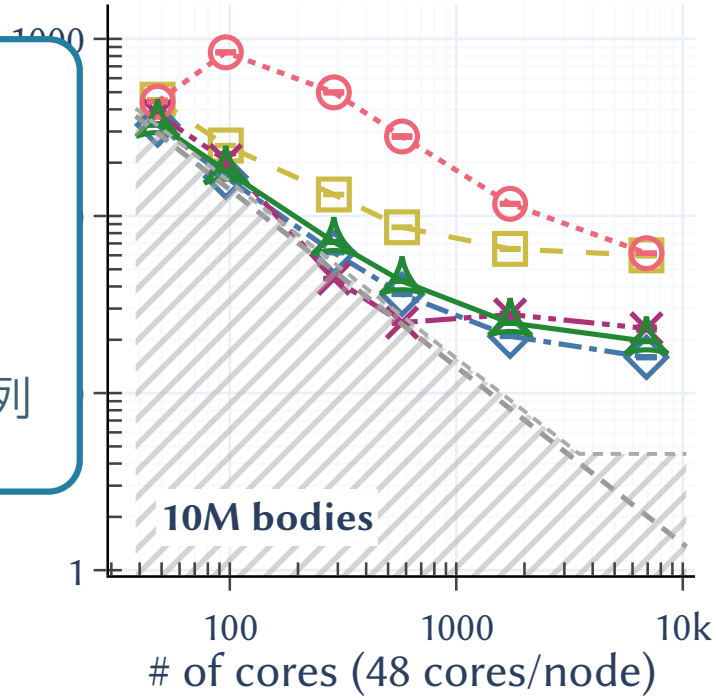
- ExaFMM の MPI 実装 **[Yokota+, CPC '13]**
- MPI とタスク並列のハイブリッド
  - ノード間: 静的負荷分散、明示的な通信
  - ノード内: MassiveThreads によるタスク並列

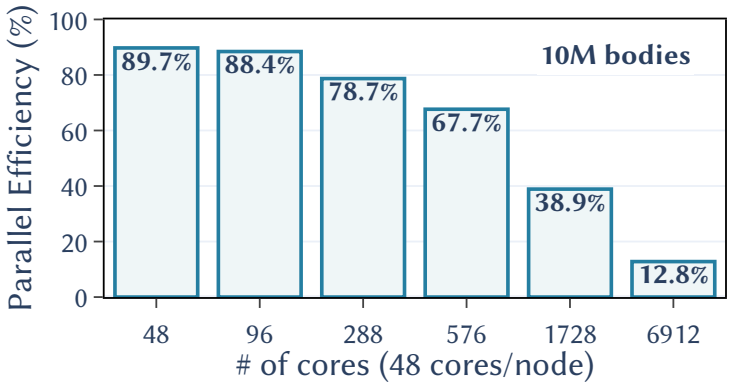
SQUID (汎用 CPUノード群)



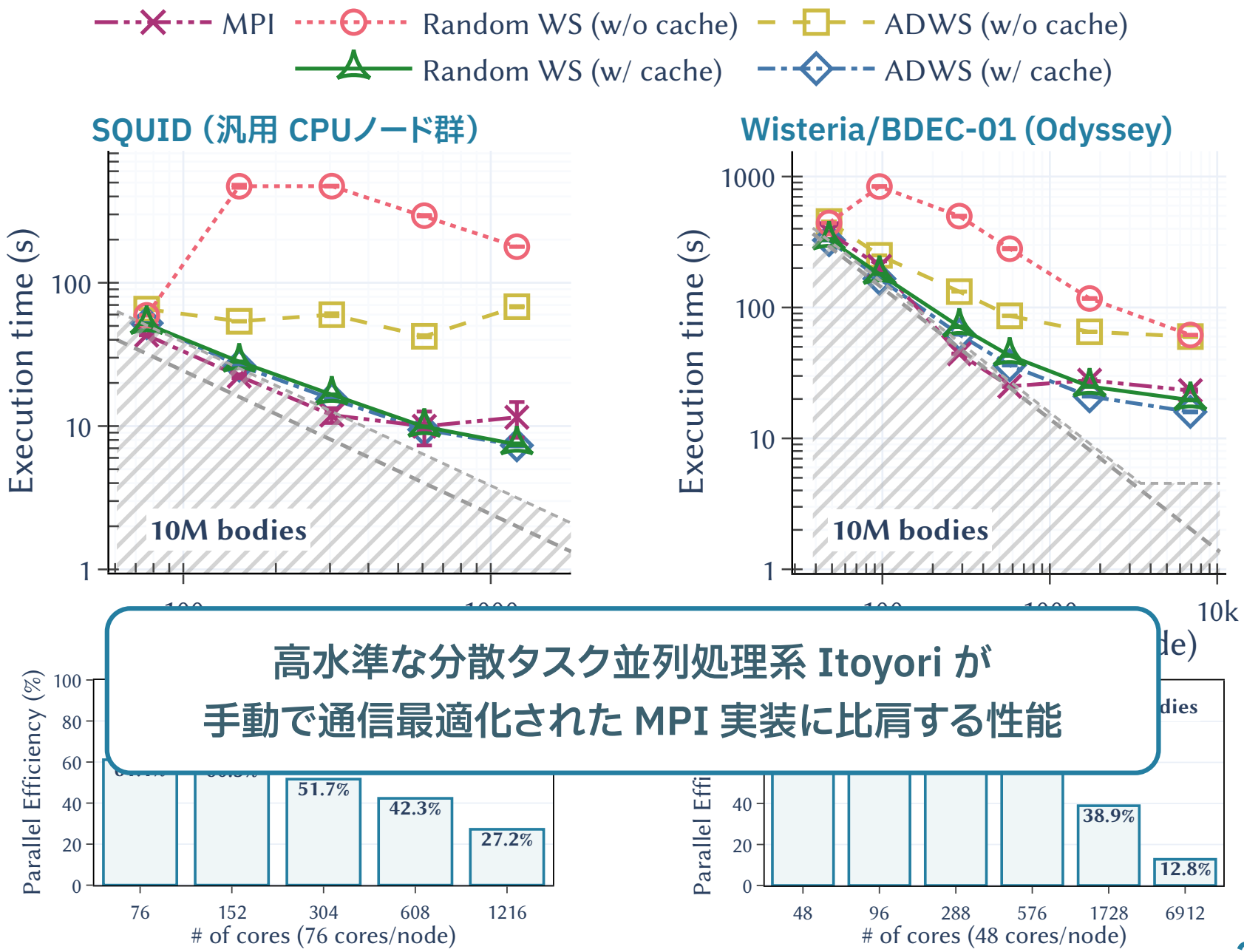


Wisteria/BDEC-01 (Odyssey)





ExaFMM の性能評価 (y=実行時間、Strong Scaling)



高水準な分散タスク並列処理系 Itoyori が
手動で通信最適化された MPI 実装に比肩する性能

Outline

1. ソフトウェアキャッシュを備える PGAS

2. 局所性に配慮したタスクスケジューラ ADWS

性能評価

まとめ

まとめ

- グローバルビューに基づくタスク並列モデルは高い生産性が期待できるが、性能面が課題
- 本研究では、局所性を最適化することで高い性能を達成
- 大域メモリアクセスのキャッシュ、ADWS のいずれも性能を大きく改善
  - それぞれ最大 24 倍（ExaFMM）、2.4 倍（Cilksort）の性能向上

分散タスク並列処理系 Itoyori では
MPI より簡単で生産性の高いプログラミングで
高いスケーラビリティ、高効率を実現

GitHub:

https://github.com/itoyori/itoyori

▷ Itoyori ユーザになってくれる方、アプリも募集中です