Parallel Programming with Object Assemblies

Secure Compiler Seminar (2010/5/12)

秋山 茂樹

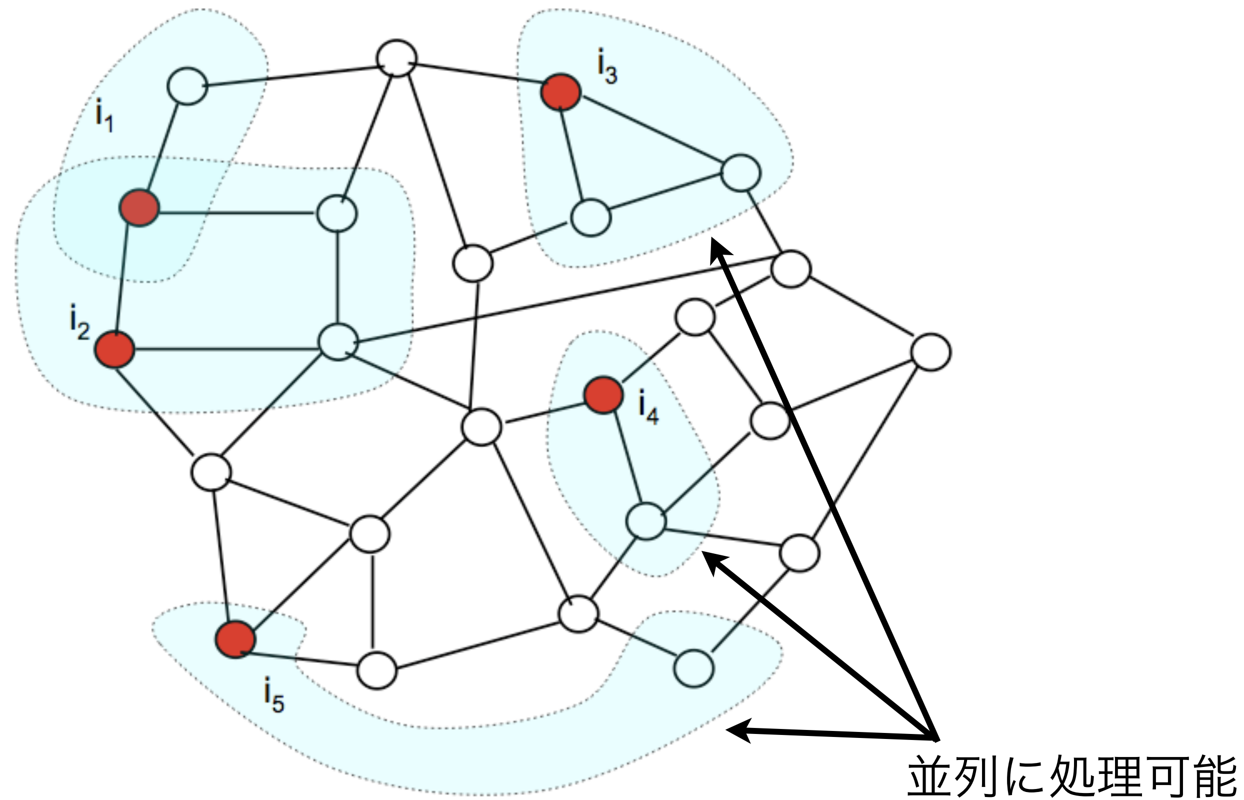
紹介する論文

- タイトル
  - “Parallel Programming with Object Assemblies”
- 著者
  - Roberto Lubllinerman, Swarat Chaudhuri, Pavol Cerny
- 発表場所
  - OOPSLA 2009
- 内容
  - irregular application のための並列プログラミングモデルの提案

irregular application

- 疎で変更可能なデータ構造の操作
  - → 木やグラフなど、ポインタベースのデータ構造
  - physical simulation, mesh refinement, spanning tree computations, n-body simulation, social network analysis, sparse matrix computation, etc...
- 特徴
  - データの更新が局所的
  - 入力依存、実行時に変更されやすい
    - 静的に並列化することが不可能

amorphous data-parallelism



参考：M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval. How much parallelism is there in irregular applications? In PPOPP, pages 3–14, 2009.

Contributions

- irregular application のための並列プログラミングモデル Chorus を提案
    - 操作的意味論の提示
    - race-freedom, deadlock-freedom の証明
- プログラミング言語 JChorus の設計と実装
- JChorus による real-life application の実装と評価

Introduction

irregular application の並列化

- irregular application が持つ次の性質を利用
  - 破壊的更新による影響がヒープの一部に限定される (locality)
- ただし
  - データ構造が動的に構成される (dynamism) ため、静的な並列化は困難

行いたいこと

- 次の処理を動的に行える必要がある

1. 処理する region を決定する
   - region: ヒープの一部分 (オブジェクトの集合)
2. region の所有権を獲得する
3. その region 上で処理を行う
4. region の所有権を放棄する

既存のモデル

- multi-thread, software transaction, PGAS, actor
  - dynamism を扱いつつ、locality をうまく活用して並列化できるプログラミングモデルは存在しない

既存のモデル: multi-thread

- ヒープはすべてのスレッドで共有
  - オブジェクトの所有権を扱うためにロックを使う
  - region レベルで所有権を扱うプリミティブを提供していない

既存のモデル: software transaction

- ヒープの read/write を管理し、衝突を検出したら処理を最初からやりなおす
  - 規模の大きな irreglar application では性能が出ない
    - 衝突を管理するオーバヘッドのため
  - 静的な解析が困難
    - ヒープの更新が大域的であるため

既存のモデル: PGAS

- Partitioned Global Address Space
  - 言語レベルでヒープを静的に分割
  - irregular application の動的な性質を扱うことができない

既存のモデル:
Actor, Active Object

- データはアクター内にカプセル化
  - メモリアクセスは message passing
- locality と dynamism を扱えるが、
  - コーディングが難しい
  - 性能が出ない

Solution: Chorus

- *object assembly* による locality と dynamism の表現
  - object assembly
    - 一つのスレッドを持つオブジェクトの集合
    - ある assembly は他の assembly 内のオブジェクトにアクセスすることができない

ヒープを object assembly に（動的に）分割し処理を並行に実行する

assembly に対する操作

- *update*
  - assembly に属するオブジェクトを更新する
- *merge*
  - 複数の assembly を一つにマージする
    - オブジェクトの所有権を獲得
- *split*
  - 一つの assembly を複数の assembly に分割する
    - オブジェクトの所有権を放棄

行いたいこと (Chorus)

- 次の処理を動的に行える必要がある
  1. 処理する region を決定する
     - region: ヒープの一部分 (object assembly)
  2. region の所有権を獲得する (merge)
  3. その region 上で処理を行う (update)
  4. region の所有権を放棄する (split)

プログラミングモデル

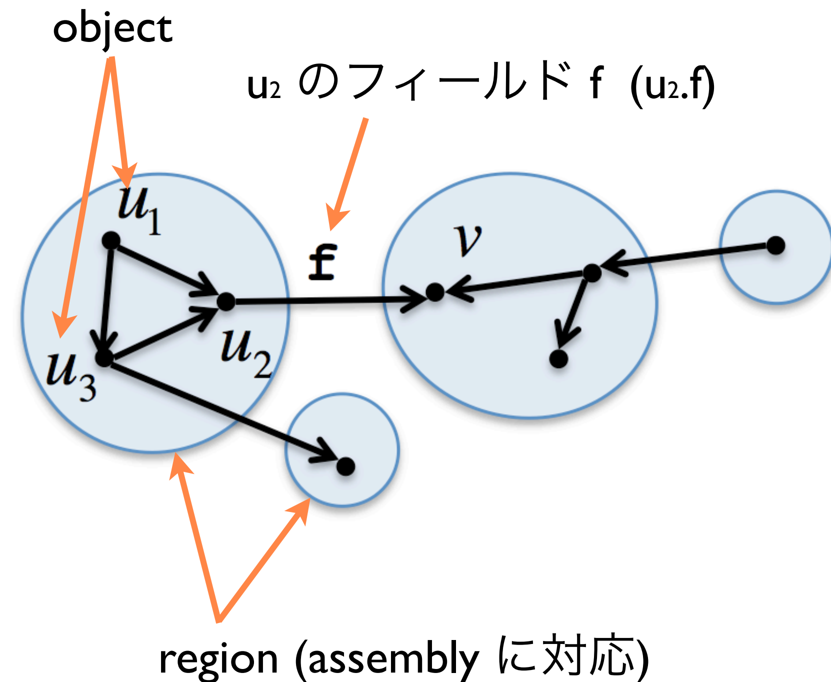
Chorus

Chorus

- Essential Chorus
  - Heaps
  - Object assemblies
  - Merges, splits, and modifications

ヒープ

- グラフとして表現
  - オブジェクトは頂点
  - ポインタは有向辺
  - ヒープの分割を region と呼ぶ



object assembly

- ローカル変数とスレッドを備えた region
  - スレッドは region 外のオブジェクトにアクセス不能
    - 変数の破壊的更新が他の assembly に影響を与えない
- assembly に対する操作
  - merge, split, update

assembly の動作

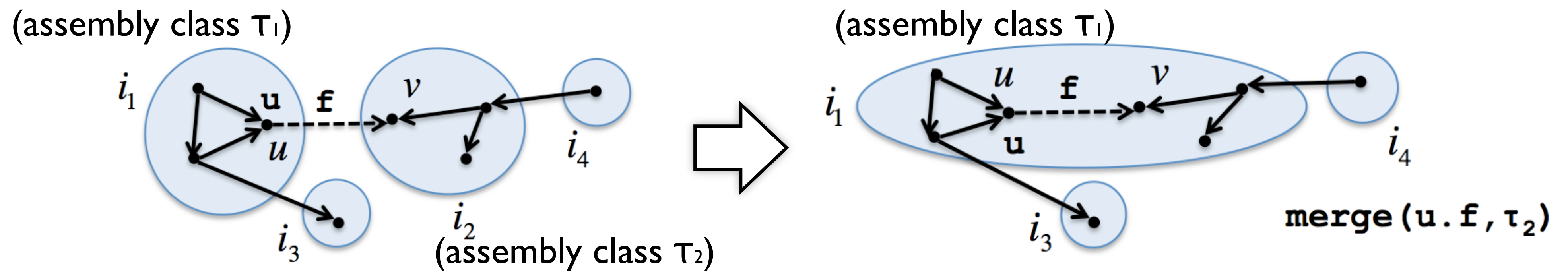
- guarded update の集合として記述
  - *Guard* : { *Update* }
    - *Guard*:
      - *Update* を実行するための条件
      - 条件式 or merge
    - *Update*:
      - オブジェクト・ローカル変数の更新処理
      - split

merge

- merge(u.f, τ)
  - u.f が指すオブジェクトの属する assembly をマージ
  - Guard の一種
    - 指定の assembly をマージできるなら真
    - そうでなければ偽
  - Chorus における唯一の同期操作

merge

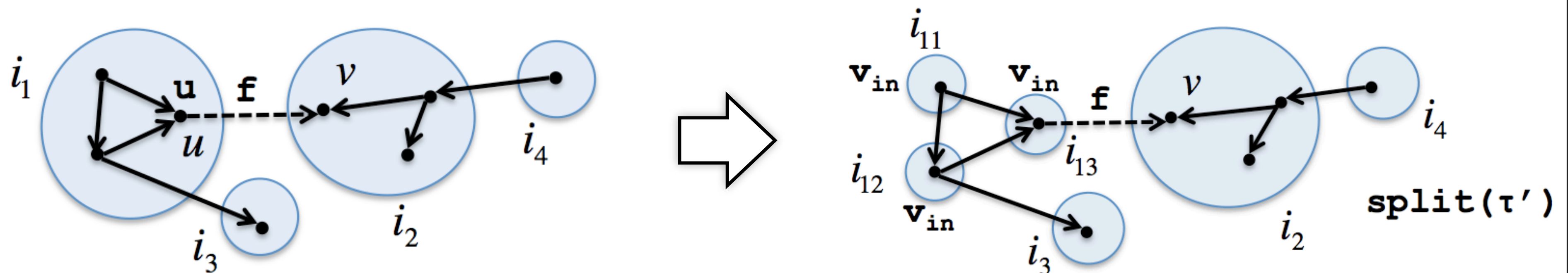
- merge(u.f, τ2)



- merge(u.f, τ1', τ2)
  - 現在の assembly と τ2 のインスタンスをマージし、τ1' のインスタンスとなる

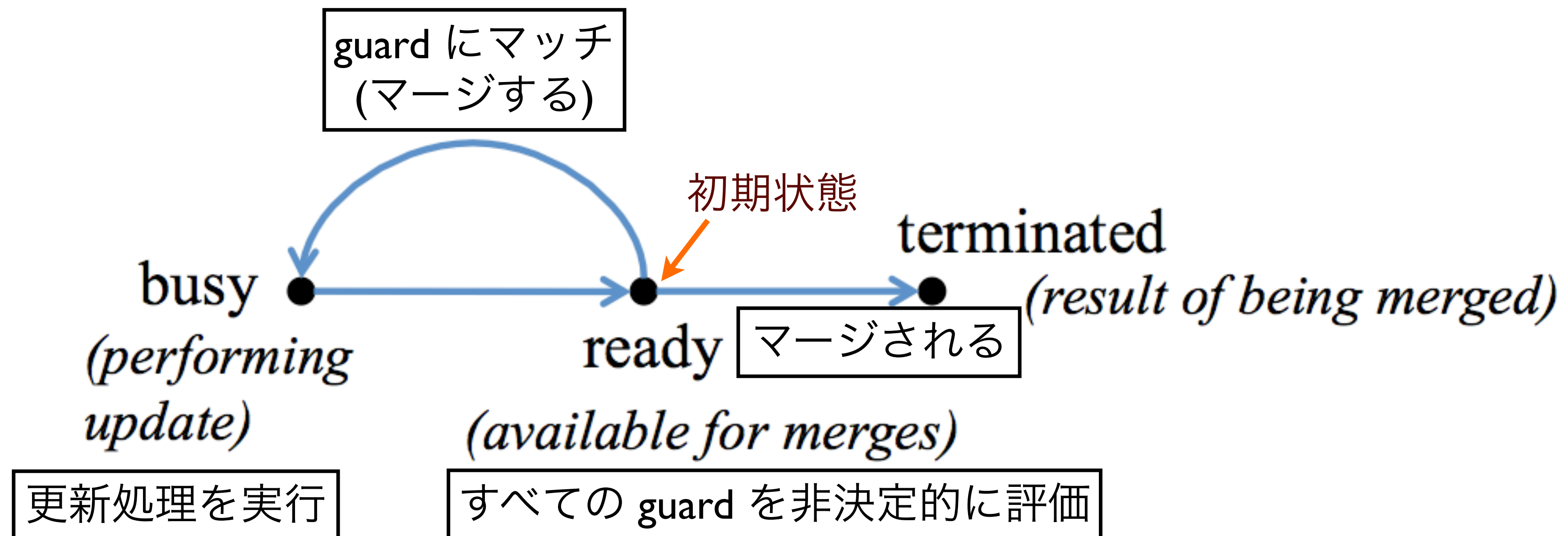
split

- split(τ’)
  - assembly を、各オブジェクト一つのみ持つ assembly の集合に分割する
  - 元の assembly は終了



control state

- assembly の control state



merge 時の動作

- assembly i1 が assembly i2 を merge
  1. i2 が ready であるか確認
  2. i2 のスレッドを終了させる
  3. i1 の状態を busy に変更
  4. i2 に属するオブジェクトの所有権を i1 に受け渡す

注意点

- 対象の assembly が ready 状態であれば、明示的な同意なしにマージすることが可能
  - assembly で行う処理が完了しないこともある

プログラムの停止

すべての assembly が

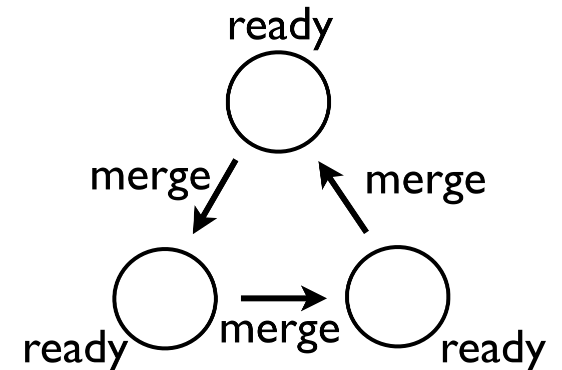
- ready 状態で、かつ
- merge ができない場合
  - すなわち、別の assembly へのポインタを持たない場合

にプログラムが停止

race- and lock-freedom

- race-freedom
  - assembly 間でオブジェクトを共有することがないため
- lock-freedom
  - ready 状態のときにはいつでも merge できるため

対象が update 中 (busy状態) であればそのオブジェクトを merge できないが、update 中に merge されることはない



JChorus

```
assembly A1 {
    Obj a, b, c, ...;
    action {
        merge(minOutEdge, A2) : {
            ... マージした assembly のオブジェクトを使って処理 ...
            ... オブジェクトの更新、split ...
        }
    }
    void compute() { ... }
}
```

assembly ローカル変数

guarded updates

メソッド

その他いろいろな機能

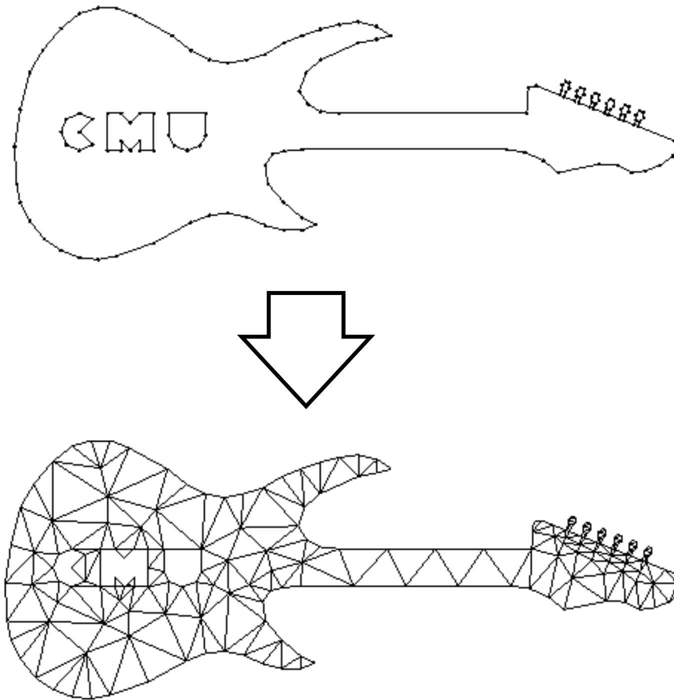
Examples

Delaunay Mesh Refinement (1)

- Delauney triangulation
  - 頂点の集合を次を満たす三角形の集合へ変換
    1. 三角形の外接円の内部に頂点がない
    2. 何らかの制約（アプリケーション依存）
       - これらを満たさない三角形を “bad” と呼ぶ
- Mesh Refinement
  - mesh から bad な三角形をなくすこと

Delauney Mesh Refinement (2)

- Delauney triangulation



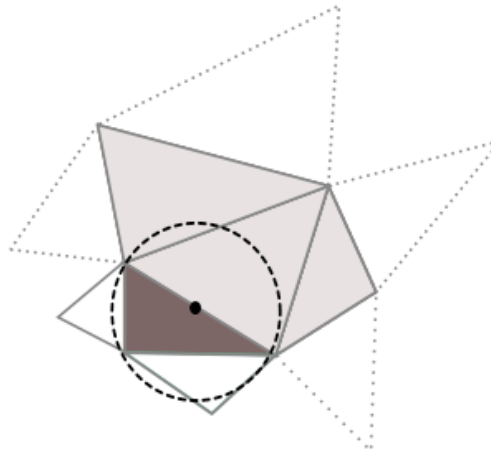
参考: http://www.cs.cmu.edu/~quake/tripaper/triangle3.html (via 池尻くん)

Delauney Mesh Refinement (3)

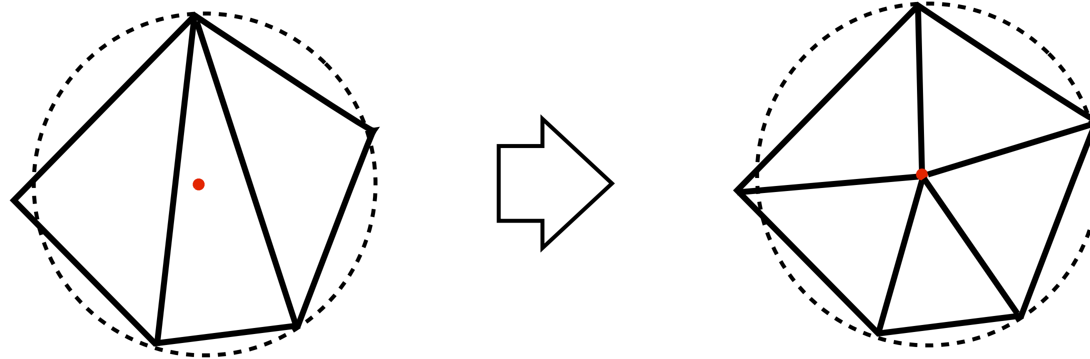
- Mesh Refinement のアルゴリズム
  - bad な三角形それぞれについて
    1. 外心 p を挿入
    2. 外接円がpを含むすべての三角形を集める (この領域を cavity と呼ぶ)
    3. cavity の境界の頂点とpとを結んで、三角形を再構成する
- <u>bad な三角形ごとに並列に実行可能</u>

cavity と retriangulation

- cavity



- retriangulation



Chorus による実装 (1)

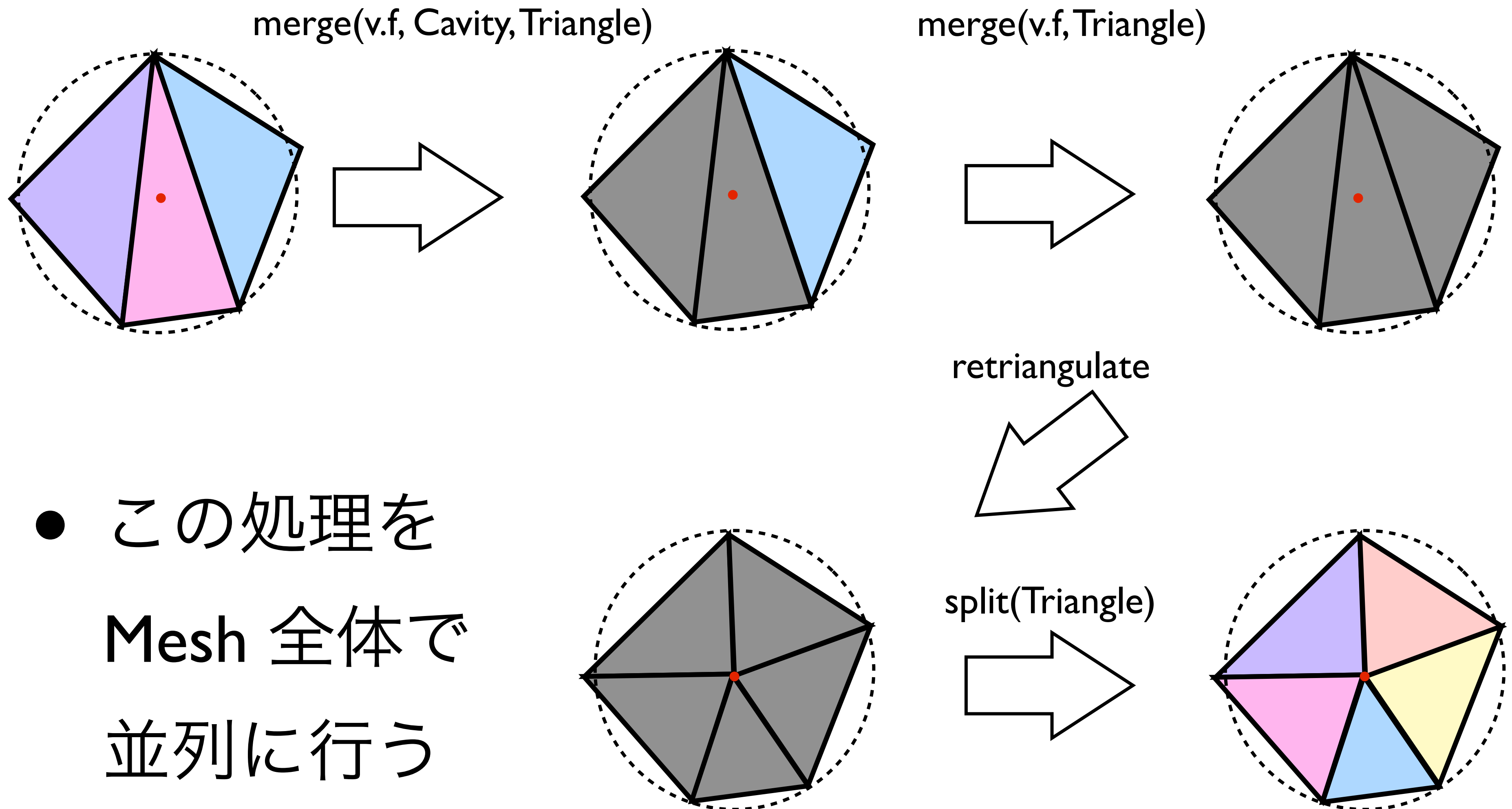
- アイデア
  - 三角形を一つ所有する assembly : Triangle
  - 三角形を複数所有する assembly : Cavity
  - 初期状態では Triangle のみ存在

Chorus による実装 (2)

- アイデア
  - Triangle が bad なら
    - Cavity になる
  - Cavity が完全でないなら
    - 隣接しており、条件に合致する三角形を merge
  - Cavity が完全なら
    - 三角形を再構成し、split(Triangle)

Cavity が完全: 内部にpを含む外接円を持つ三角形がすべて集まった状態

Chorus による実装 (3)



- この処理を Mesh 全体で並列に行う

JChorus のコード

```
assembly Triangle {
    Triangle(TriangleObj t) {
        if (isBad) become(Cavity, t);
    } }
assembly Cavity {
    action { merge(neighbors, Cavity, Triangle) : { build(); } }
    void build() {
        if (isComplete()) {
            retriangulate(); split(Triangle);
        } } }
```

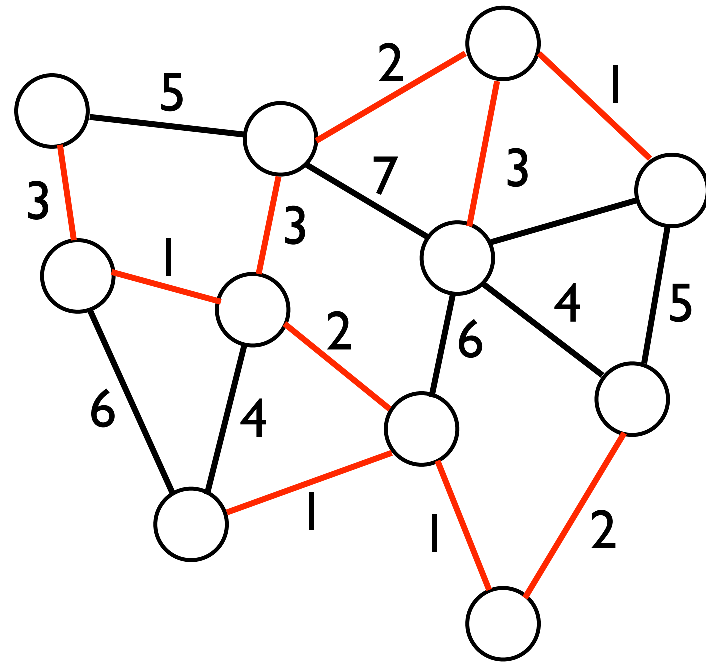
三角形が bad なら Cavity になる

隣接する三角形をマージ

Cavity が完全になれば、
三角形を再構成し Triangle に分割

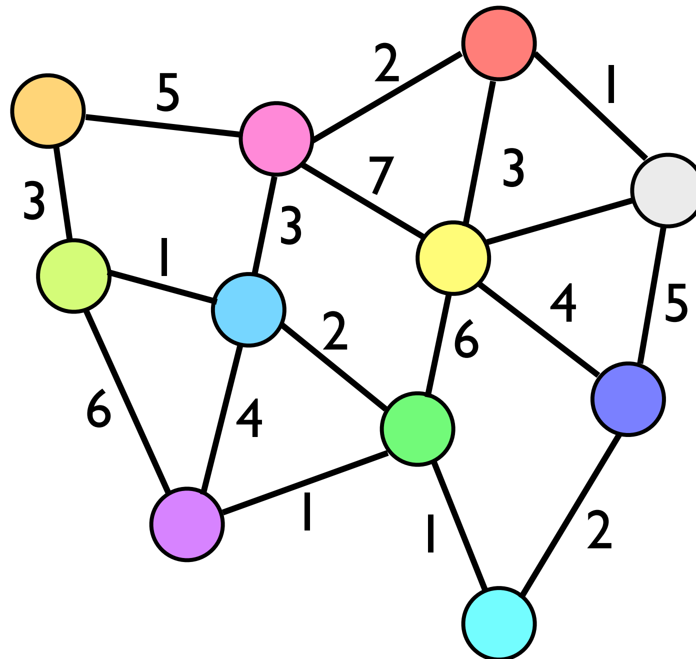
最小全域木(MST)の計算

- 逐次アルゴリズム
  - 訪問済みの頂点から未訪問の頂点への辺の中で、最小のものを選んでいく



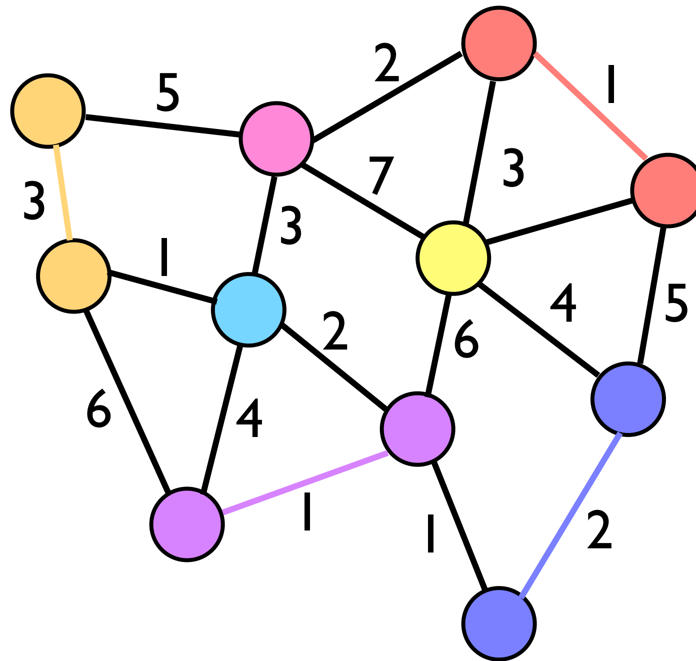
ChorusでMST計算 (1)

- 各ノードを assembly とする
  - assembly をノードが一つのMSTと考える



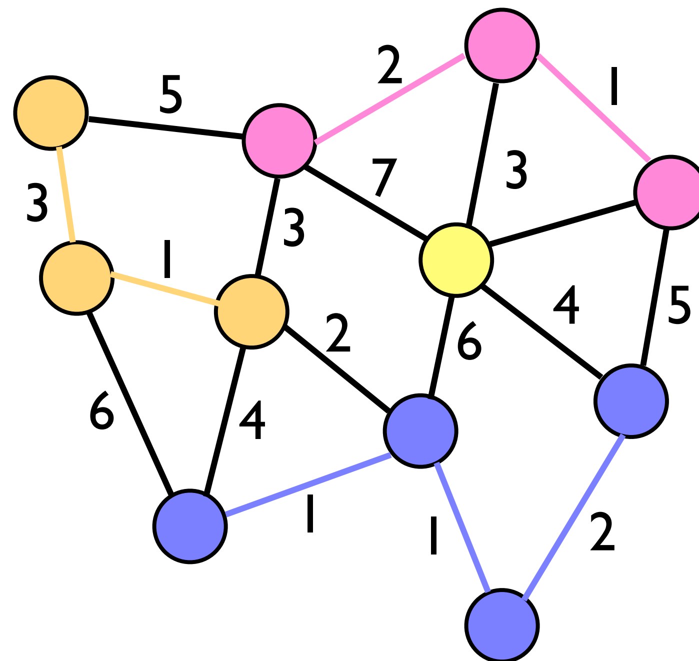
ChorusでMST計算 (2)

- 隣接する assembly のうち、辺が最小のものをマージ



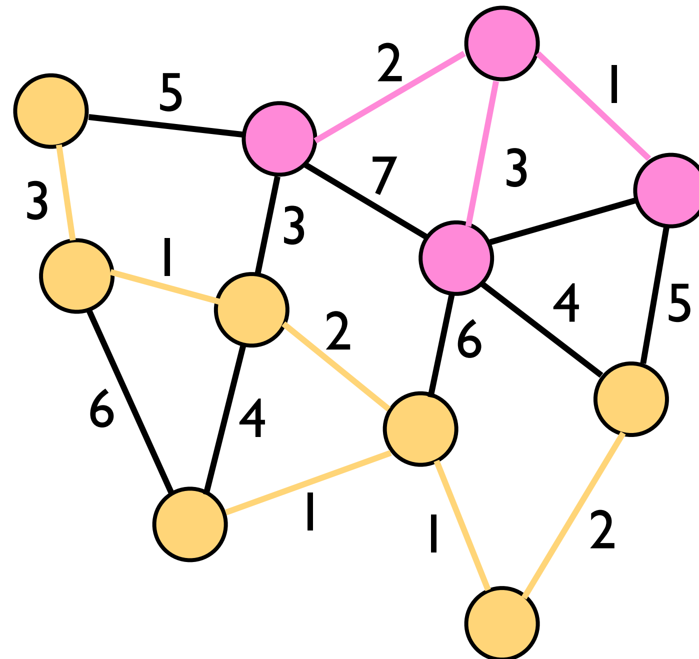
ChorusでMST計算 (3)

- マージ



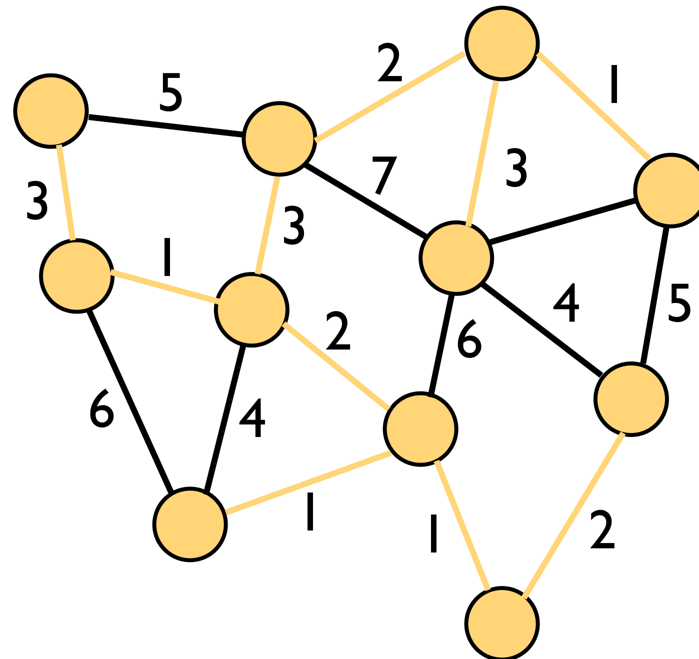
ChorusでMST計算 (4)

- さらにマージ



ChorusでMST計算 (5)

- 完了



MST計算のコード

```
assembly ComputeMST {
    Tree currentTree;
    SortedList outgoingEdges;
    Edge minOutEdge;
    action {
        merge(minOutEdge, ComputeMST) : {
            computeNewMinEdge();
        }
    void computeNewMinEdge() { ... }
}
```

```
object Edge {
    Node from, to;
    int weight;
}
```

最小の辺を持つ assembly と merge

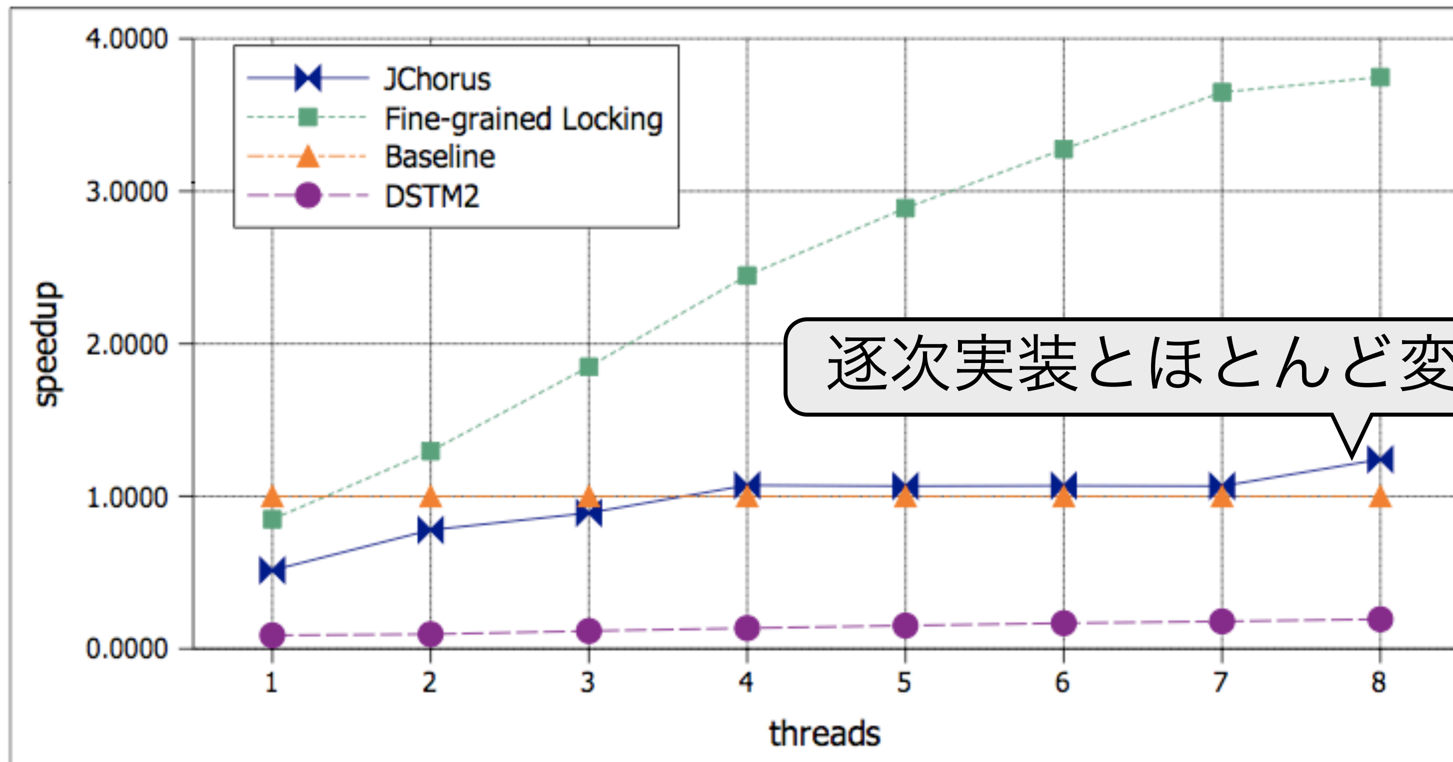
MST をマージし、最小の辺(minOutEdge)を計算

Experiments

評価実験

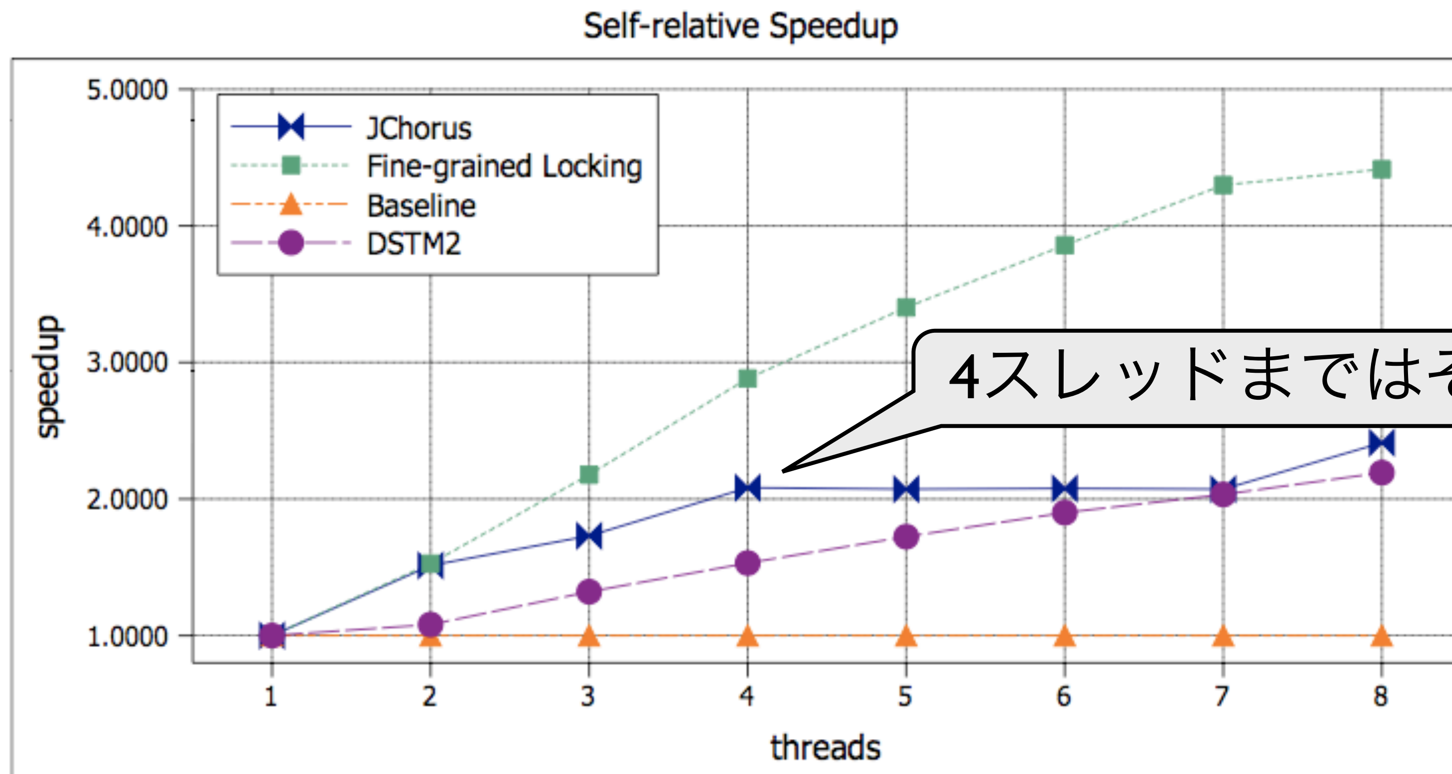
- アプリケーション
  - Delauney Mesh Refinement
  - Minimum Spanning Tree computation
- 比較対象
  - Sequential Java
  - Fine-grained Locking
  - DSTM2
- 環境
  - Xeon X5550 (2.66GHz, 8-core),
  - 24GB Mem, 64-bit Linux, Java 1.6

Delauney
Mesh Refinement (1)



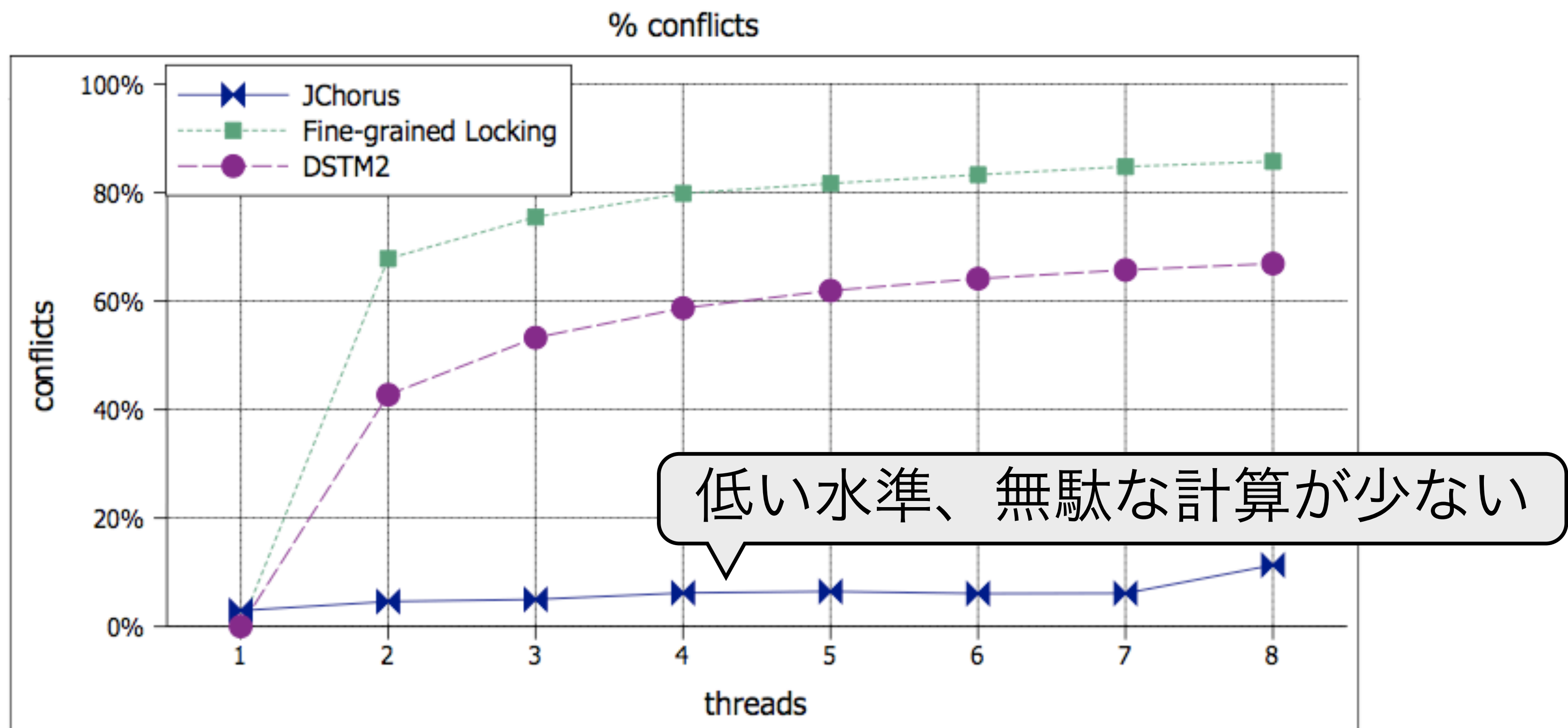
Javaコードを介することによる
不要なランタイムチェックが原因ではないか

Delauney Mesh Refinement (2)

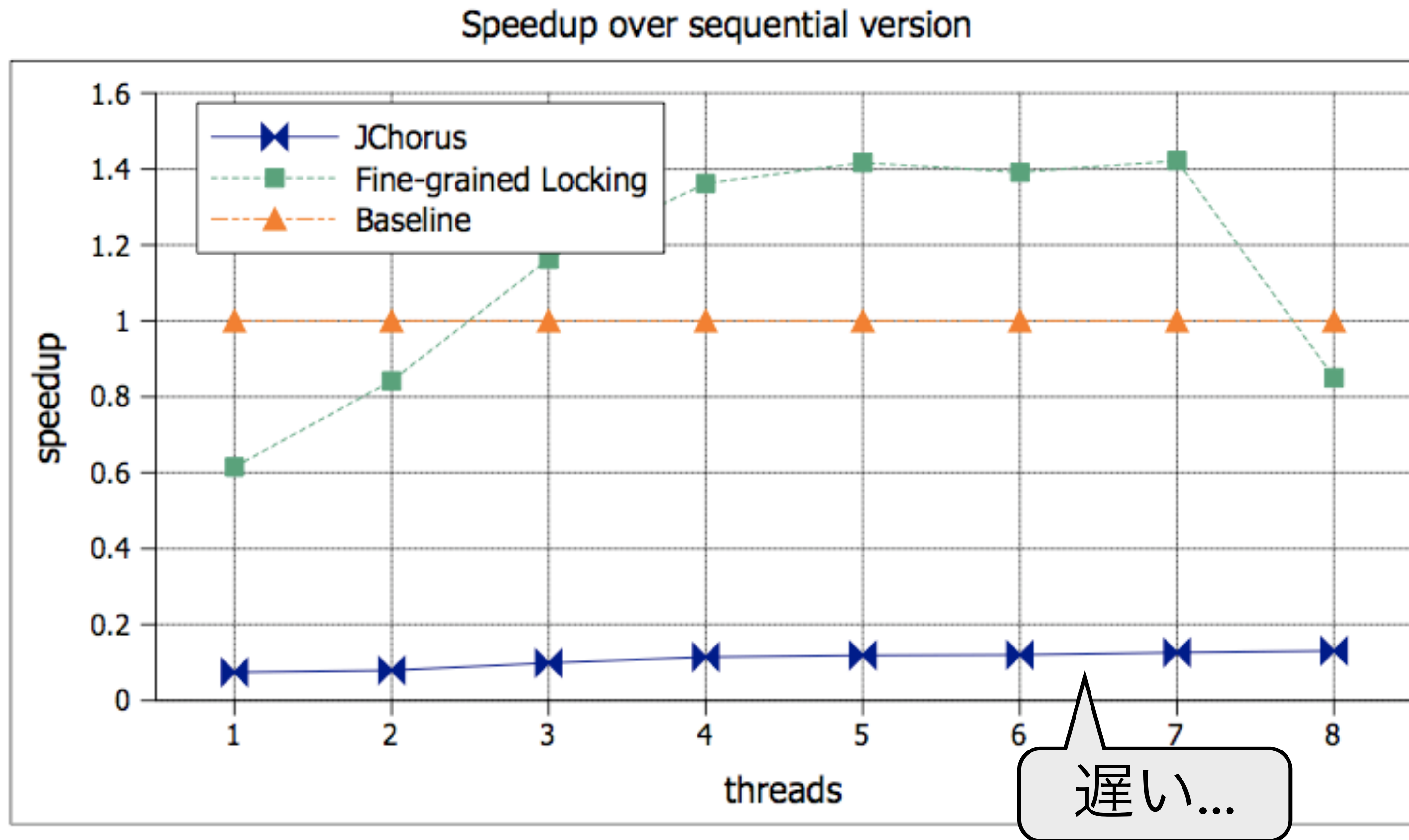


Delauney Mesh Refinement (3)

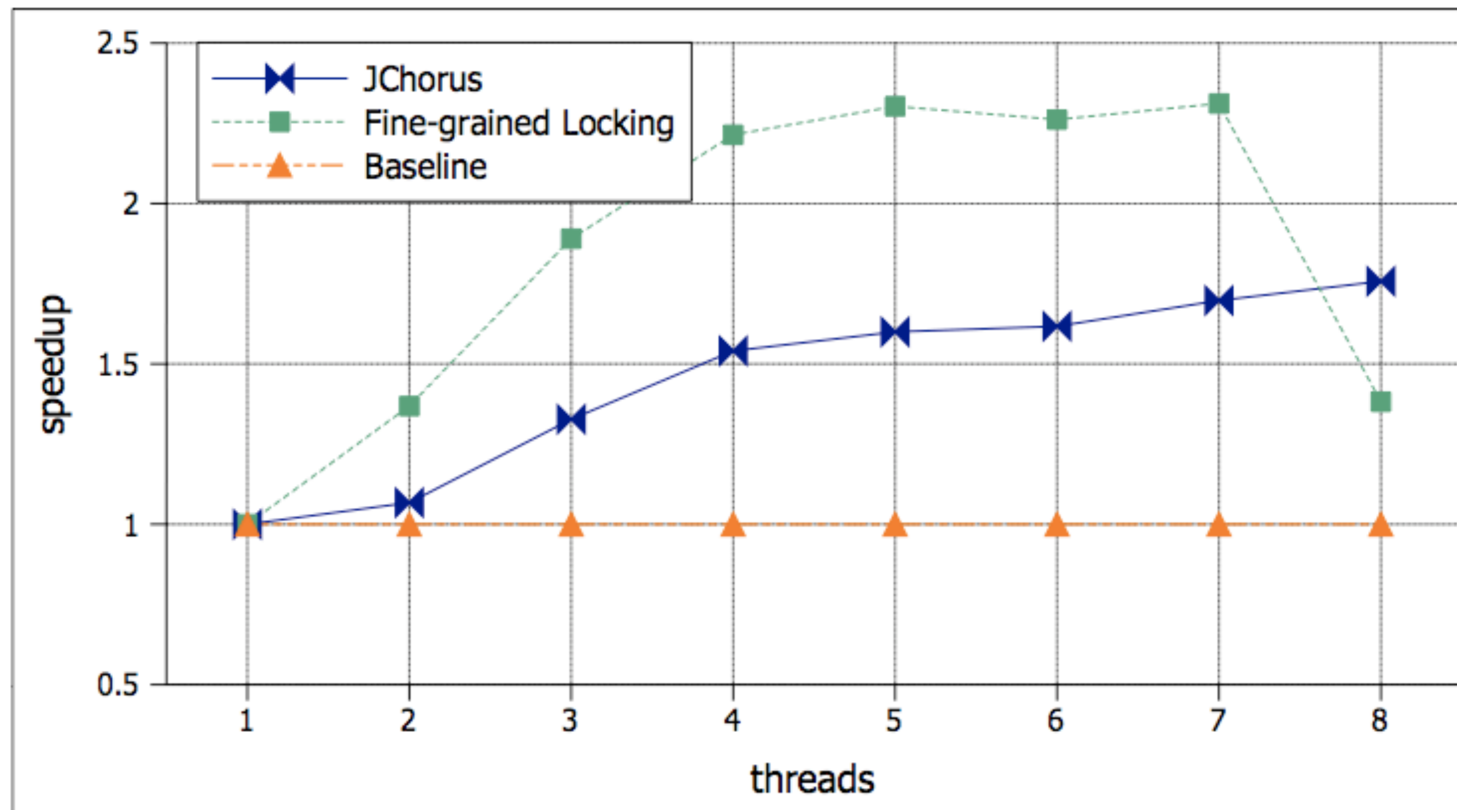
- % conflicts
  - cavity のうち、マージされて処理が行われなかったものの割合



MST computation (1)



MST computation (2)



Conclusion

まとめ

- irregular application のための並列プログラミングモデルを提案
  - locality と dynamism をうまく利用し、並列性を引き出すことができた
- 今後の課題
  - JChorus の効率の良い実装
  - assembly に関する不変条件の推論