並列プログラミングパターン

2011年5月27日

目的

このセッションを通じて可能になること:

- デザインパターンに隠されるコンセプトと並列デザインパターンについて説明できる
- 既存のシリアルコードやアルゴリズムをスレッド化する際に、より良いアルゴリズム構造のデザインパターン(タスク並列かジオメトリー分解か)を選択できるようになる
- 既存のシリアルコードやアルゴリズムをスレッド化する際に、より良い構造をサポートするデザインパターン(SPMD、ループ並列、ボス/ワーカーのいずれか)を選択できるようになる

内容

パターン言語構造

タスク並列パターン

ジオメトリー分解パターン

構造のサポート

まとめ

内容

パターン言語構造

タスク並列パターン

ジオメトリー分解パターン

構造のサポート

まとめ

デザインパターンとパターン言語

デザインパターンとは:

- コンテキストの問題の解決法
- 再発する問題を解決する高品質な構造化された記述
- すべての設計者が優れたデザインを区別する、「無名の質」を攻略することができるよう、専門技術をコード化することを探求すること

パターン言語とは:

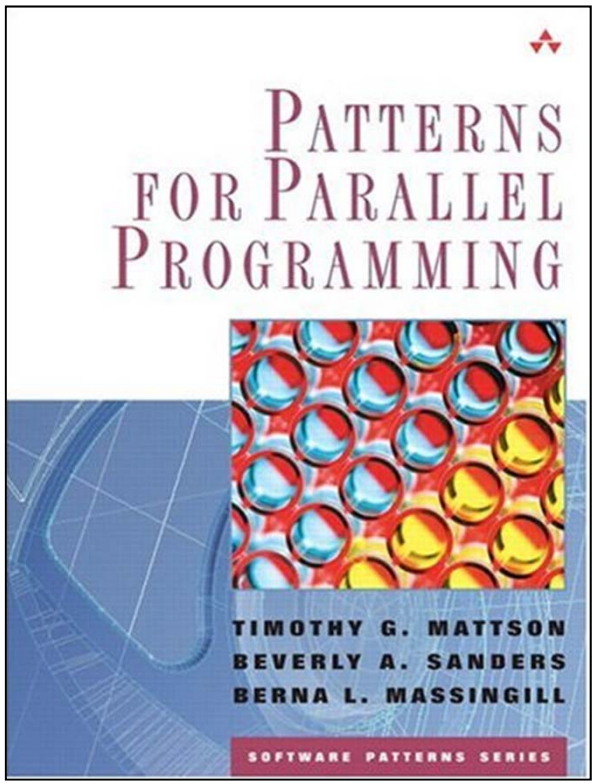
- 設計活動をサポートする設計パターンが構造化されたカタログ

参考文献

並列アルゴリズム設計におけるパターン言語について解説

MPI、OpenMP そして Java による例題を提示

プログラマーがどのように並列プログラミングについて考えるのか、著者の仮説を述べている

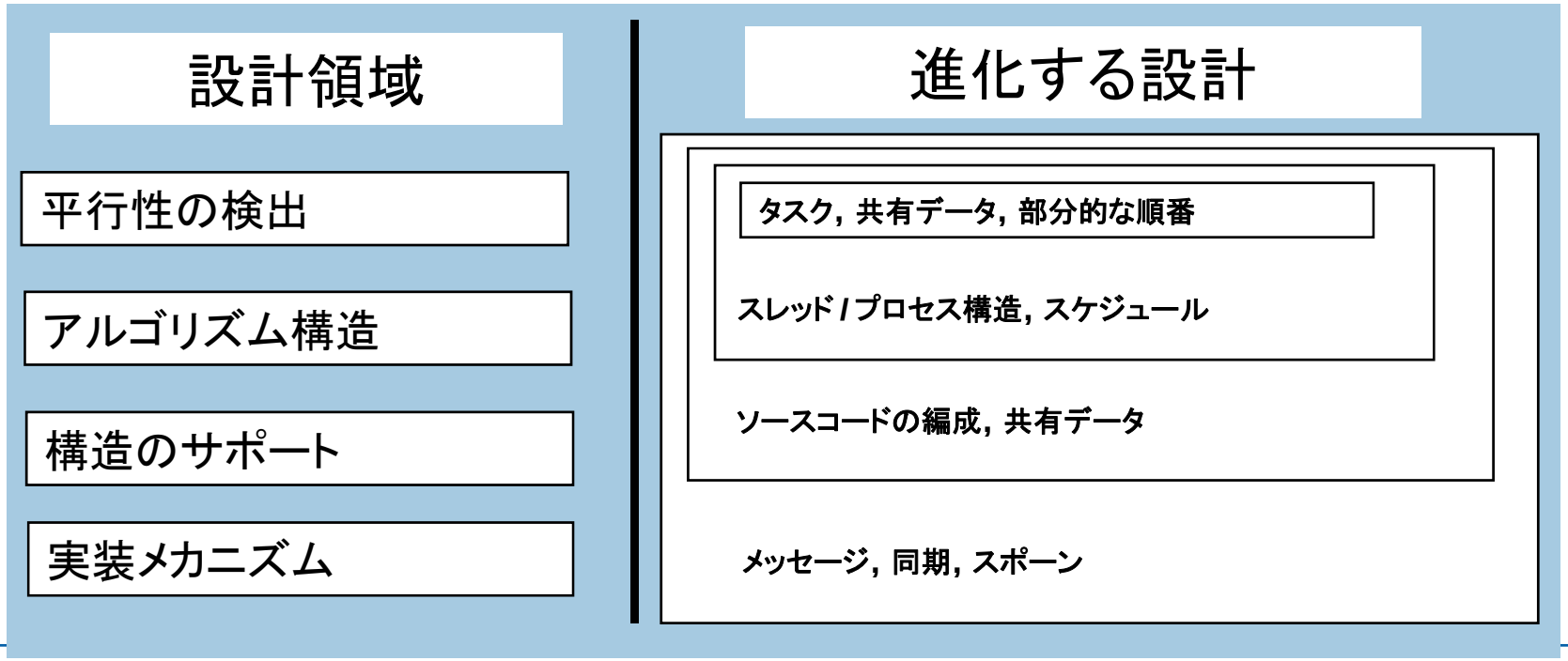


Patterns for Parallel Programming, Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Addison-Wesley, 2005, ISBN 0321228111

パターン言語構造

ソフトウェアの設計は、4 つの設計領域における改善考察の一連の過程であるとみなすことができる

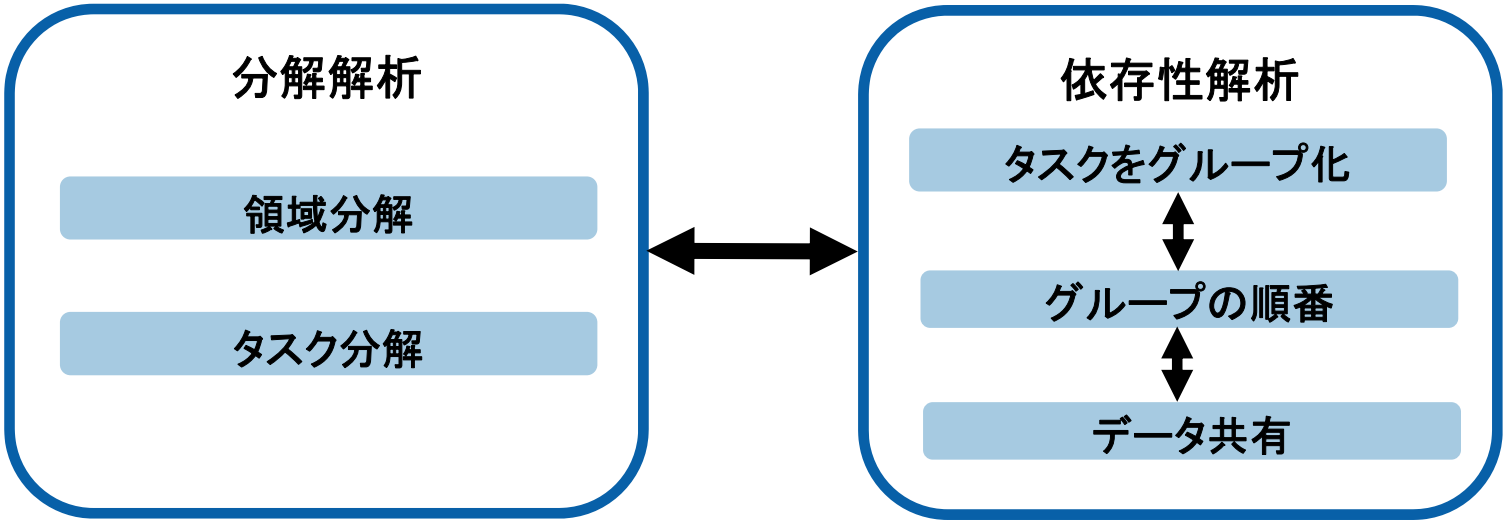
- 次第に低レベルの要素を設計に追加していく



平行化の設計領域を判別する

並列化する領域を見つける:

- 既存の問題を解決するためシリアル・プログラムから始める
- アプリケーションをタスクやデータの集合に分解する
- 分解する前にタスク間の依存性を解析する



平行性の検出

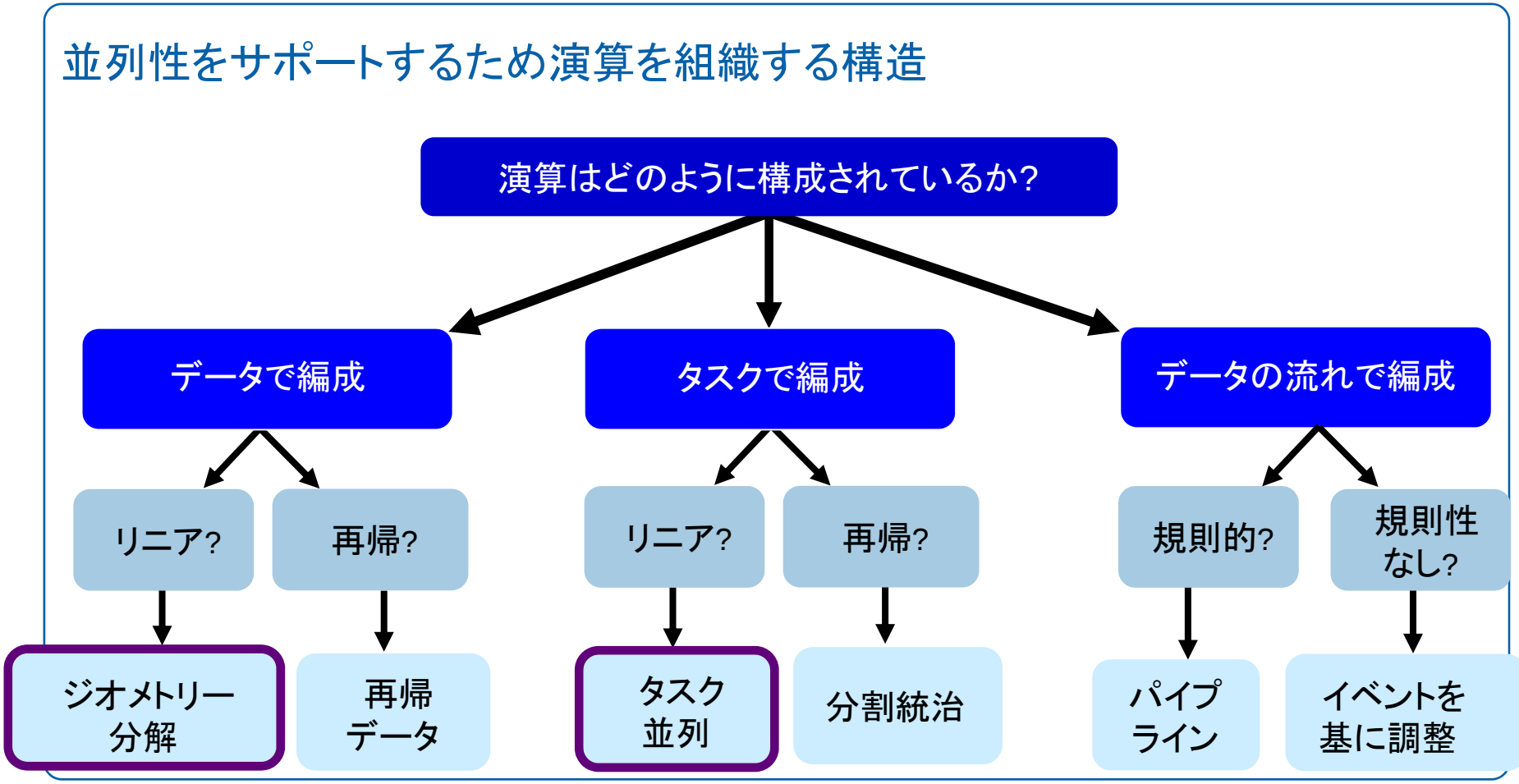
シリアルコードから:

1. インテル® VTune™ Amplifier XE などの解析ツールを利用する
2. アプリケーションのホットスポットを特定する
3. ホットスポットが見つかったコードを調査する
4. ホットスポット内のタスクは独立して実行可能かどうか判断する

設計文書から:

- コンポーネントの設計を調査する
- コンポーネントの独立した操作を見つける

アルゴリズム構造の設計領域



アルゴリズム構造への影響

アルゴリズムの設計目標に影響する次の項目を考慮する:

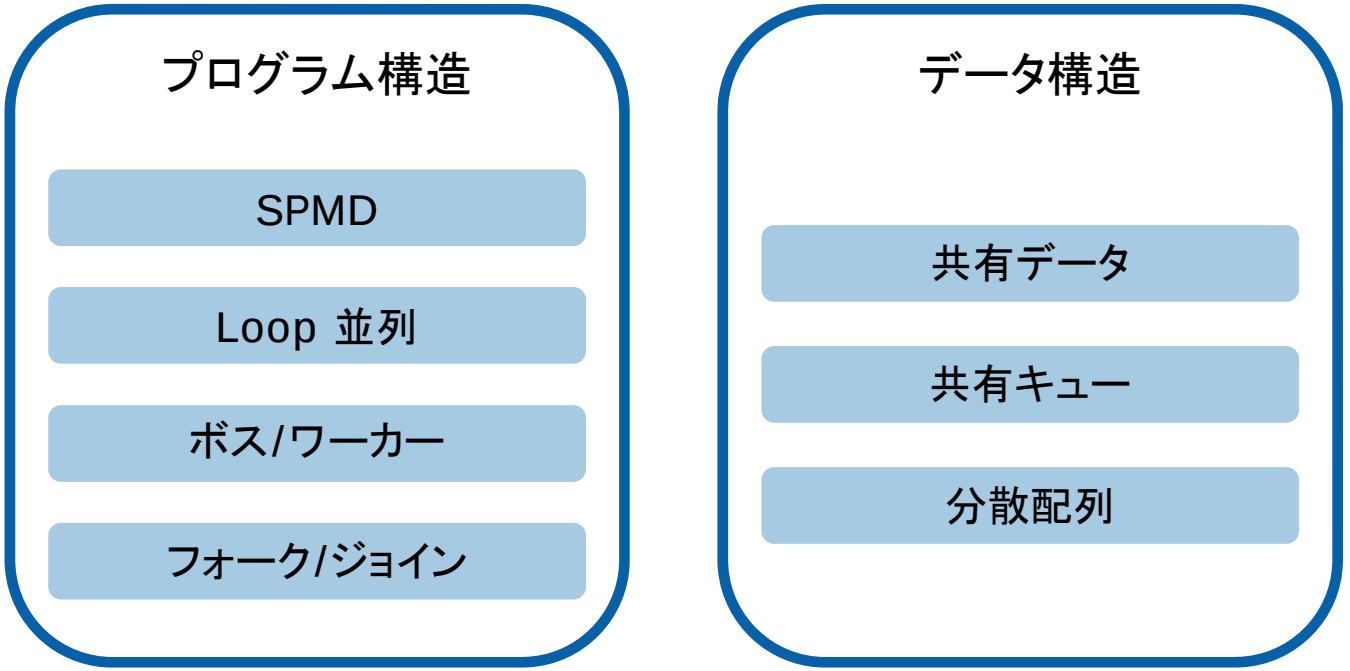
- **効率性**
  - 並列プログラムは演算リソースを効率よく利用して性能を高めなければならない
- **簡易性**
  - 簡単なアルゴリズムは、開発、デバッグ、検証、そして保守を容易にする
- **可搬性**
  - プログラムは多様な並列コンピューターで実行できなければならない
- **拡張性**
  - 並列アルゴリズムは、スレッドやコアの数、そしてデータセットの大きさに関わらず有効でなければならない

これらは相互に影響を及ぼす

- 最適な並列コードを開発するには、これらのバランスを取ることが重要

設計領域でサポートする構造化

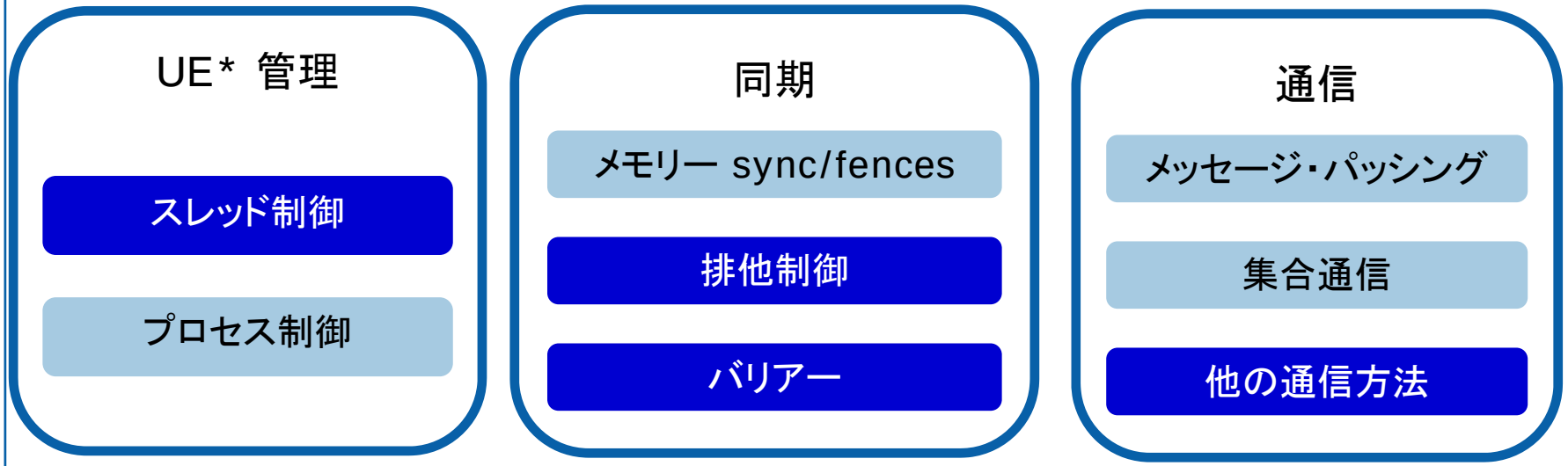
ソースコードを組織化し、プログラム構造とデータ構造を分類するため、ハイレベルの実装要素が利用されてきた



設計領域の実装メカニズム

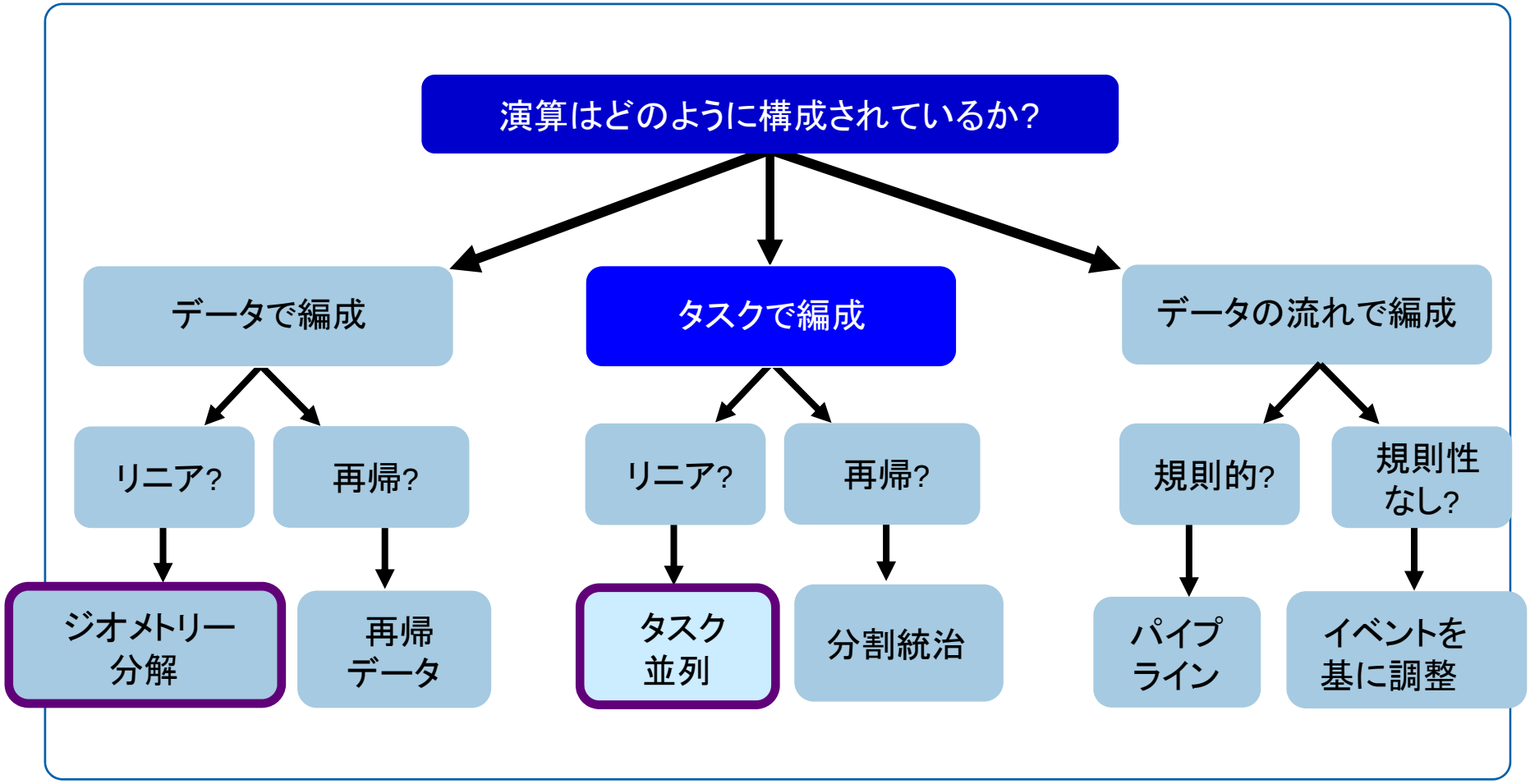
並列計算で利用される特定の構造を実装するローレベルのコンストラクト

- 適切なデザインパターンではない；パターン言語を自己充足するように含まれる



内容

アルゴリズム構造の設計領域



タスク並列パターン

問題:

- 一連の異なるタスクに関してどのように並列性を利用するか？

例:

- **レイ・トレーシング** – 各光線の計算は他の光線の計算から独立している

- **分子動力学** – 原子における力の作用は独立している; すべての作用が完了した後、各スレッドによる部分的な作用の計算が結合される

タスク並列パターンの関係

あらゆる並列アルゴリズムは同時発生するタスクの集合である

- コードを調査することで見つけることができる
- 共通の要因は、計算を同時発生するタスクの集合に分解することである
  - タスクは完全に独立しているかもしれない
  - タスクは満たさなければならない依存性があるかもしれない

通常タスクは計算の始まりとして認識される

タスクは動的に発生するかもしれない（例えば、分岐限定法による木探索）

一般的に処理を終える前に、すべてのタスクの完了を待機しなければならない

- すべてのタスクが完了する前に答えが見つかった場合、停止できるかもしれない
  - 例：検索処理の場合、検索中にアイテムが見つかったら、それ以上検索する必要はない。アイテムが見つからなければ、無いことを知るため検索を終了する

タスク並列パターンの作用

タスクの大きさ

- 小さなタスクは負荷を均等にする、そして、大きなタスクはスケジュールのオーバーヘッドを軽減する

依存性の管理

- 効率、簡易性、拡張性を損なうことなく行う

タスク並列パターンのソリューション

タスク並列のパターンデザインには 3 つの重要な要素がある

- タスクはどのようにそれを定義されるか
- タスク間の依存性
- タスクをスレッドへ割り当てる

タスクの定義

シリアルアルゴリズムをタスクに分解するには、2 つの基準がある:

1. 少なくともスレッド数(もしくはコア数)と同じ程度のタスクがあるべきである
   - スレッドよりもかなり多くのタスクを用意することが望ましい
     - スケジューリングにおいて大きな柔軟性を与える
     - スレッド間で負荷バランスを保つことを支援する

2. タスク中の計算量は、タスクとスレッドを管理するオーバーヘッドを相殺するくらいの大きさでなければならない

初期段階の分解がこの評価基準を満たさない場合、別の分解方法を考える

例: タスクを定義する

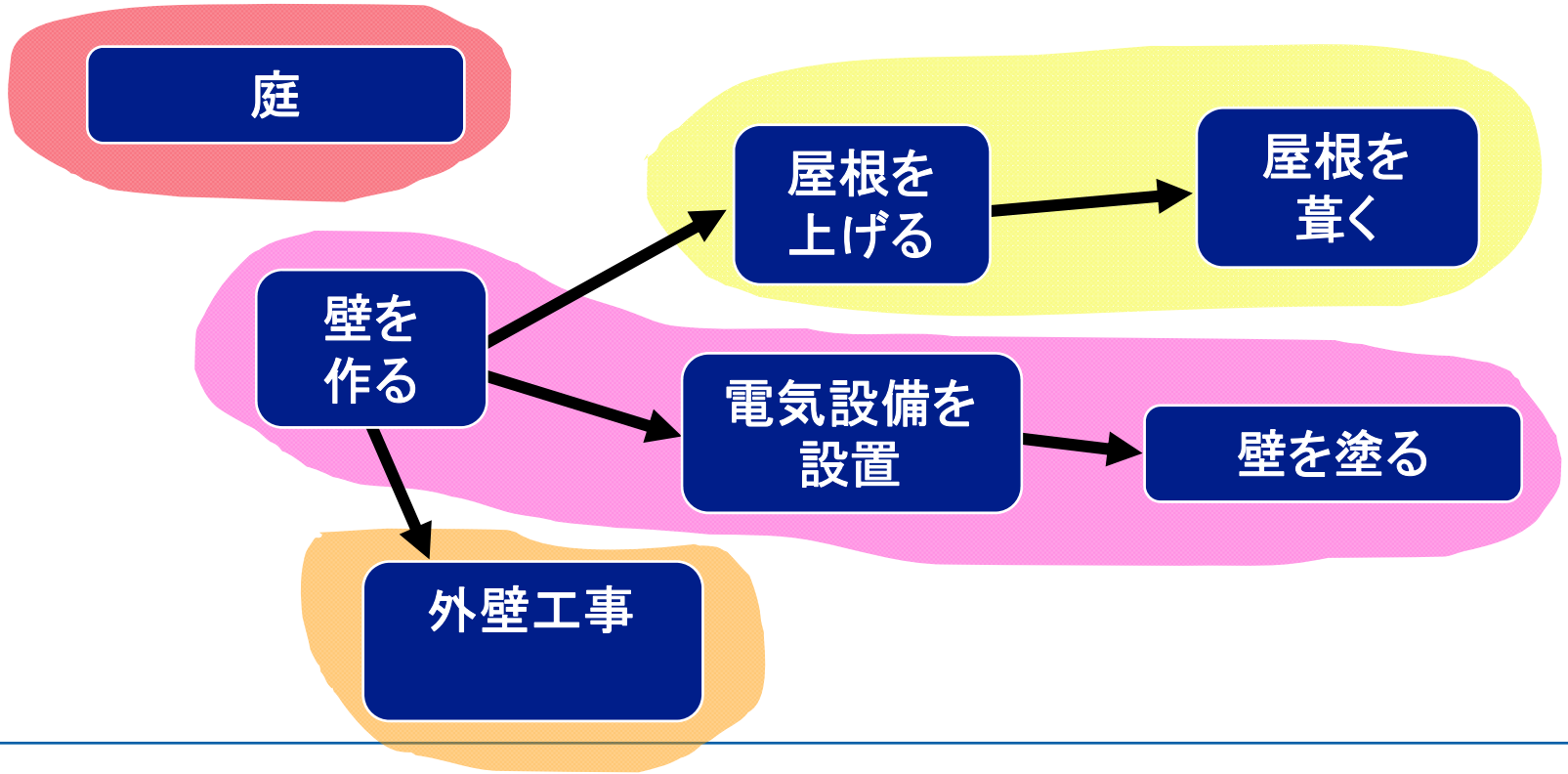
各ピクセルの処理が独立しているイメージ処理を想定

ここでのタスク分解の評価基準は?

- タスク == 1 ピクセル
- タスク == 1 行
- タスク == イメージ内のブロック

タスク並列パターンにおける依存性の管理

実行するタスクグループの順番を強制することで、タスク間の順番を制限する



タスク並列パターンにおける依存性の管理

データ依存性の解決はより複雑

- 単純なケースは依存性がない場合（驚異的並列）

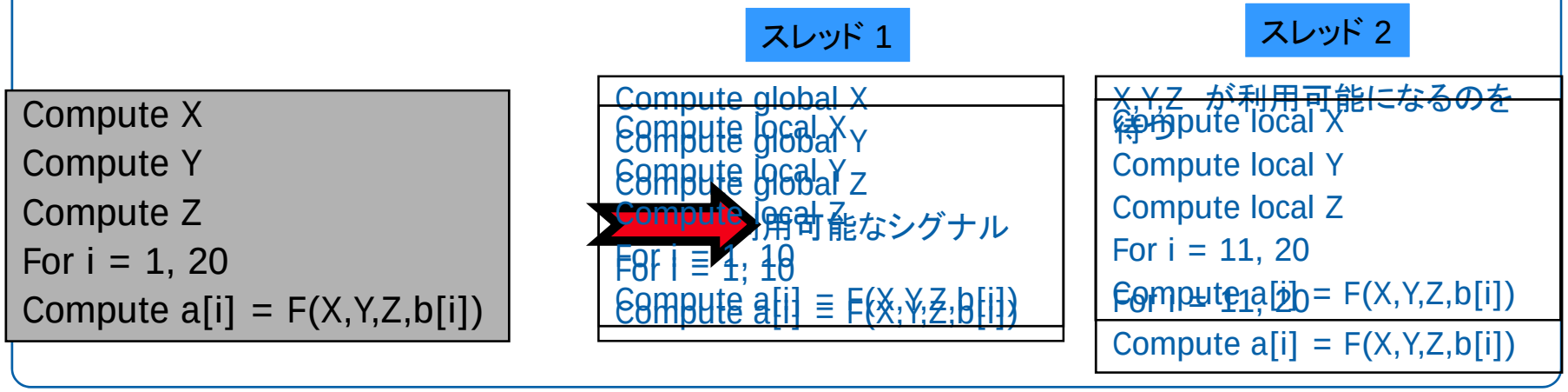
- タスク間のデータ依存を扱う戦略:
  - 依存性の排除（データ複製）
  - インダクション変数を変換
  - リダクションの実装
  - 明示的な保護（共有データパターン）

排除可能な依存性

コードを変更することで排除可能な見せかけの依存性

最も簡単な解決策は、ローカル変数を作成し、初期化して利用すること

- スレッド内で複製を行わなければいけないかもしれない



排除可能な依存性

インダクション変数はループの各反復でインクリメントされる

```
i1 = 4;
i2 = 0;
for (i = 0; i < N; i++)
{
   B[i1++] = ... ;
   i2 = i2 + i;
   A[i2] = ... ;
}
```

インクリメント式をループインデックスを利用した計算式に置き換える

```
for (i = 0; i < N; i++)
{
  B[i+4] = ...;
  A[((i+1)*i)/2)] = ...;
}
```

“分離可能な” 依存性

各スレッドで計算されたデータの累積に利用される共有データ構造への依存性

- 累積を並列領域外で行う:
  - 各スレッドへデータ構造を複製する(必要であれば初期化する)
  - タスクは複製されたローカルデータを利用して実行
  - 並列実行が完了したら、すべてのローカルデータを共有データへ結合

もっとも一般的な例: リダクション

- データの収集は、すべての要素に演算を適用し 1 つのデータに結合する
  - 演算は交換かつ結合可能でなければならない

```
for (int i = 0; i < N; i++)
   sum += a[i] * b[i];
```

明示的な共有データの保護

タスク間の依存性を排除もしく分離できなければ、共有データの更新は保護されなければならない

共有データにアクセスするコード領域にクリティカル領域を実装

- データが更新されるならば、**すべてのアクセス**（読み書き両方）は、クリティカル領域でなければいけない
- クリティカル領域を排他制御するプログラムロジックを追加する
- 適切な同期オブジェクトを利用して相互排他を作成する

タスクのスケジュール

タスクを実行するためスレッドに割り当てる

負荷バランス良くタスクをスケジュールする

2 つのスケジュール方式が利用される

- 静的スケジュール
  - タスクは演算開始時にスレッドへ割り当てられ、変更されない
  - プログラムは簡単でオーバーヘッドは最小
  - 可能であれば静的スケジュールを利用する
- 動的スケジュール
  - タスクは演算の過程で割り当てられる

静的スケジュールにおける考察

仕事は各スレッド向けにユニークなスレッド ID で識別されるかもしれない

タスクが別々の独立している関数呼び出しの収集である場合

- スレッドに割り当てられるタスクのグループ呼び出し

タスクがループ反復の場合、2 つのスケジュール方法がある:

1. ループ反復の総数をスレッド数で割り、反復のブロックをスレッドに割り当てる
2. 各スレッドはスレッド ID (0..N-1) を基にループ反復を開始し、各反復の N 番目を計算する

動的スケジュールにおける考察

タスクの実行時間が可変で予測できない場合、
動的にタスクをスケジュールために利用する方法

- タスクを保持するため共有構造を利用する
  - タスクを構造にカプセル化できるなら、グローバル・キューを作成してスレッドが必要とするタスクを一元管理
  - タスクがファイルに保存されるなら、スレッドはアクセスを共有しファイルからタスクを読み込む
- いくつかのタスクはスレッドに計算を割り当てる前に、事前処理が必要かもしれない
  - スレッドへ割り当てるタスクを取得し、スレッドから要求があれば割り当てを行う別スレッドを利用する (マスター/*ワーカー* パターン)
  - 別スレッドでタスクを共有キューへ配置し、スレッドが必要に応じてキューからタスクを取得する (生産/*消費* パターン)
- タスクがインデックス付けされている場合、共有カウンターを作成し次に実行するタスク割り当てをトラックする

例: 航空機事故を避ける

空港周辺の飛行機の状況をスナップショット

お互いに隣接して危険な状況の組み合わせはあるか?

- もしあれば、即座に警告しイベントを記録
- スナップショット中の航空機によって生成された警告数を把握する
  - どの飛行が最も危険であるか判断するために利用される

航空機はフライト番号と現在位置を 3D 空間に割り当てる

- X と Y は地面に(地図)に関する位置
- Z は高度

つぎのスライドに疑似コードを示す

- 警告の可能性があれば、簡単な距離計算で判断

例: 航空機の位置判断コード

```
function AirplanePositions (N, Flights)
  int const N;                          // 航空機番号
  Array of Airplane_t :: Flights(N); // 航空機構造の配列

  Real :: distX, distY, distZ, distance

  loop [j = 0, N]
      loop [k = j+1, N]
        distX = (Flights[j].Xcoord – Flights[k].Xcoord) ^ 2
        distY = (Flights[j].Ycoord – Flights[k].Ycoord) ^ 2
        distZ = (Flights[j].Zcoord – Flights[k].Zcoord) ^ 2
        distance = sqrt(distX + distY + distZ)
        if (distance < SAFETY_MARGIN)
          PrintWarning(Flights[j],Flights[k])
          Flights[j].numWarnings++
          Flights[k].numWarnings++
      end loop[k]
    end loop[j]
end function AirplanePositions
```

例: 航空機の位置判断コード – タスクを定義

```
function AirplanePositions (N, Flights)
  int const N;                          // 航空機番号
  Array of Airplane_t :: Flights(N); // 航空機構造の配列

  Real :: distX, distY, distZ, distance

  loop [j = 0, N]
     loop [k = j+1, N]
       distX = (Flights[j].Xcoord – Flights[k].Xcoord) ^ 2
       distY = (Flights[j].Ycoord – Flights[k].Ycoord) ^ 2
       distZ = (Flights[j].Zcoord – Flights[k].Zcoord) ^ 2
       distance = sqrt(distX + distY + distZ)
       if (distance < SAFETY_MARGIN)
         PrintWarning(Flights[j],Flights[k])
         Flights[j].numWarnings++
         Flights[k].numWarnings++
     end loop[k]
   end loop[j]
end function AirplanePositions
```

各航空機間の距離計算は
1 つのタスク

例: 航空機の位置判断コード – 依存性

```
function AirplanePositions (N, Flights)
  int const N;                          // 航空機番号
  Array of Airplane_t :: Flights(N); // 航空機構造の配列

  Real :: distX, d

  loop [j = 0, N]
    loop [k = j+
      distX = (Flights[j].Xcoord – Flights[k].Xcoord) ^ 2
      distY = (Flights[j].Ycoord – Flights[k].Ycoord) ^ 2
      distZ = (Flights[j].Zcoord –
      distance = sqrt(distX + dist
      if (distance < SAFETY_MARGI
        PrintWarning(Flights[j],Fl
        Flights[j].numWarnings++
        Flights[k].numWarnings++
    end loop[k]
  end loop[j]
end function AirplanePositions
```

各スレッドはループインデックス j と k のローカルコピーを必要とする

共有変数の更新は保護されなければならない

例:航空機の位置判断コードのタスクはどのようにスケジュールしなければならないか?

```
loop [j = 0, N]
  loop [k = j+1, N]
    distX = (Flights[j].Xcoord – Flights[k].Xcoord) ^ 2
    distY = (Flights[j].Ycoord – Flights[k].Ycoord) ^ 2
    distZ = (Flights[j].Zcoord – Flights[k].Zcoord) ^ 2
    distance = sqrt(distX + distY + distZ)
    if (distance < SAFETY_MARGIN)
      LogWarning(Flights[j],Flights[k])
      Flights[j].numWarnings++
      Flights[k].numWarnings++
    endif
  end loop[k]
end loop[j]
```

静的スケジュール

N の航空機に分割しスレッドに割り当てる

- N/(スレッド数) の航空機をスレッドへ割り当て
- 各 (スレッド数)番目 の航空機をスレッド ID で始まる各スレッドへ割り当てる

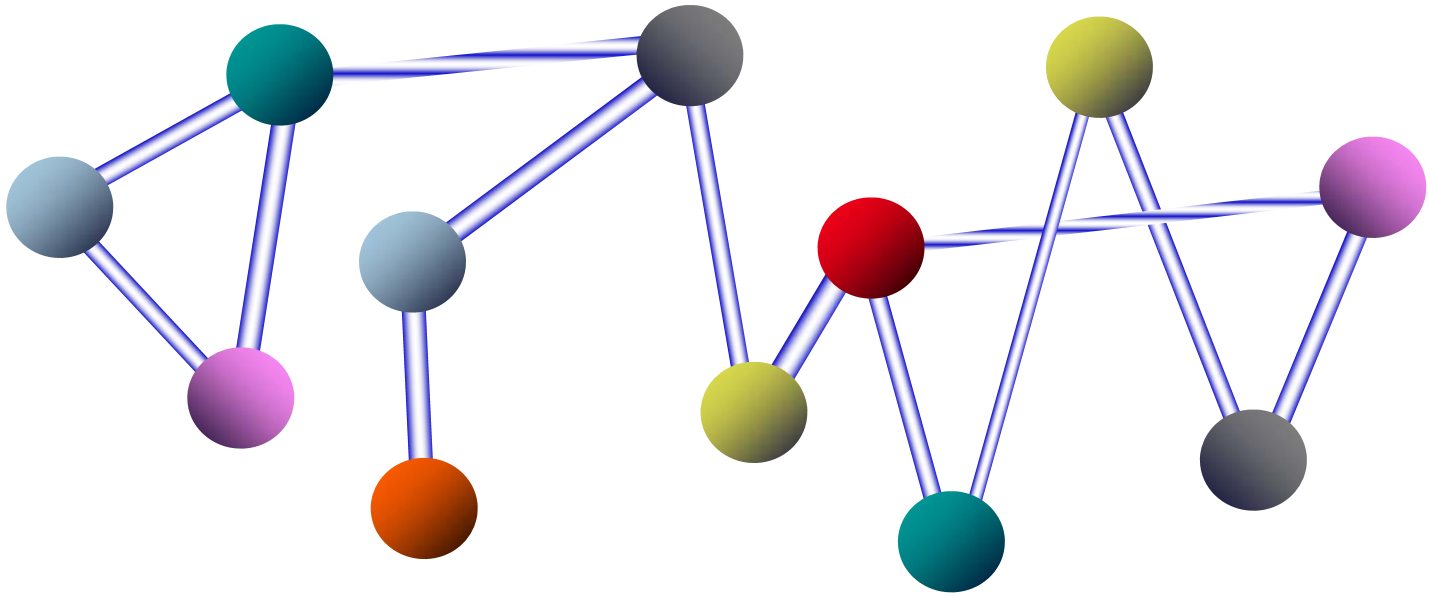
動的スケジュール

グローバルカウンターをセットアップ (0..N-1)

スレッドは次の [j] のためのカウンターをアクセスし、カウンターをインクリメント

- アクセスとグローバル・カウンターのインクリメントは保護されなければならない

例: 分子動力学コード



疑似コード内のタスクは?
それらに依存性はあるか? あればどのように解決するか?
タスクをスレッドで実行するためどのようにスケジュールするか?

例: 分子動力学のコード

原子の収集では対の原子間における非接触の力を計算する

範囲内で対になった原子だけの限界をあらかじめ設定する

- 各原子の隣接するリストで範囲内に他の原子がどこにあるかを判断する
- ニュートンの第 3 法則:Atom[j] への Atom[i] の力は、Atom[i] では Atom[j] の力を打ち消す
  - Atom[j] が Atom[i] の隣接リストにあるならば、Atom[i] は Atom[j] のリストには無い

次のページの疑似コード

- 物理計算はこの例では必要ない
- 隣接リストがどのように更新されるかは示されない

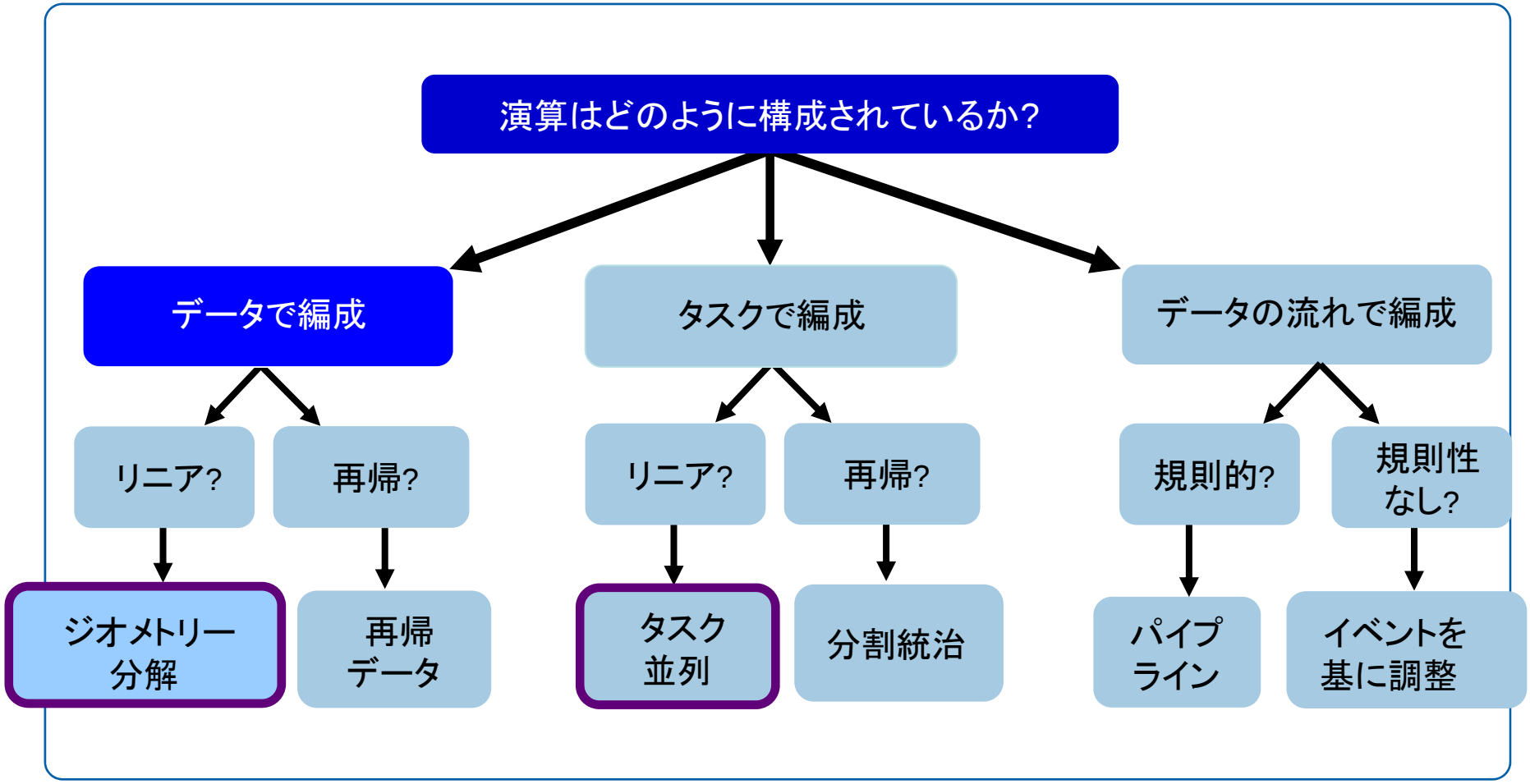
例:分子動力学の疑似コード

```
function non_bonded_forces_all (N, Atoms, neighbors, Forces)
  Int const N                          // 原子の数
  Array of Real :: atoms(3,N)          // 3D 座標
  Array of Real :: Forces(3,N)         // 各次元の力
  Array of List :: neighbors(N)        // 隣接する原子
  Real :: forceX, forceY, forceZ

  loop [i] over atoms
    loop [j] over neighbors(i)
      forceX = non_bonded_force_pair(atoms(1,i), atoms(1,j))
      forceY = non_bonded_force_pair(atoms(2,i), atoms(2,j))
      forceZ = non_bonded_force_pair(atoms(3,i), atoms(3,j))
      Forces(1,i) += forceX;   Forces(1,j) -= forceX;
      Forces(2,i) += forceY;   Forces(2,j) -= forceY;
      Forces(3,i) += forceZ;   Forces(3,j) -= forceZ;
    end loop[j]
  end loop[i]
end function non_bonded_forces_all
```

内容

アルゴリズム構造の設計領域



ジオメトリー分解パターン

問題:

- 独立したもしくは同時に更新可能な “チャンク” に分割可能なデータ構造を、どのように並列アルゴリズムとして組織化できるか?

例:

- **天気予報** – 予測する領域を重なり合う領域が無いように個々に分割; 風、陸、太陽、湿度、隣接する領域の天候などの計算を繰り返す
- **ニュートン・ラフソン法** – 素数平面中の各ポイントで、多項式の根に達するのに必要な反復数を計算する

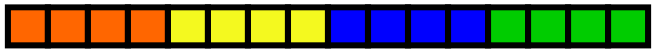
ジオメトリー分解パターンの背景

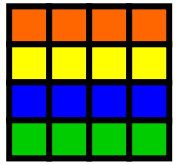
多くの問題は、コアデータ構造に対する操作の連続として理解できる

- データ構造のすべての要素は計算によって更新されるか、利用される

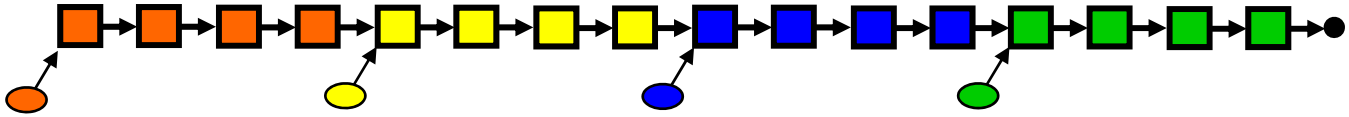
データ構造は隣接するサブ構造やサブ領域に分割される

- 配列: 次元ごとに分割する

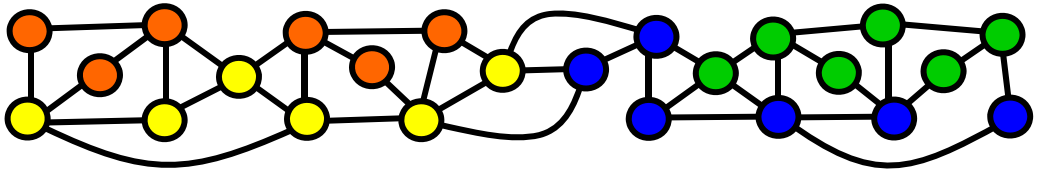




- リスト: 個々の要素のサブリストを定義する



- グラフ: サブグラフを作成する



ジオメトリー分解パターンの背景

サブ領域を説明するにあたり、“チャンク” という一般的な用語を利用する

データをチャンクに分割することは、各チャンクから要素を操作する計算をタスクに分割することを暗示する

- タスクは同時に実行される
- 各タスクは関連するチャンクを更新する
- タスクは共有される「隣接する」チャンクのデータを必要とするかもしれない
  - 隣接するチャンクは、オリジナルのデータ構造体の「近くに」あったデータを含む

ジオメトリー分解パターンの効果

目標: 簡単で、移植性が高く、スケーラビリティーがあり、効率よい

データのチャンクをスレッドに割り当てなければならない

- 特に可変サイズのチャンクを利用する場合、負荷バランスは重要
- 関連する計算はスレッドで実行可能であること

データがチャンクを更新する必要がある場合、必要なデータが利用可能であることを確実にする

- チャンクからのデータは容易に利用可能
- 隣接するチャンクのデータをアクセスもしくは検索する場合、スレッド間の調停が必要かもしれない

ジオメトリー分解パターンのソリューション

次の作業を考慮する必要がある:

1. データ分解
   - グローバルデータ構造をチャンクに分割
   - チャンクの粒度と型を考える
2. 交換操作
   - 各タスクが更新に必要なすべてのデータにアクセスできることを確実にする
   - “外部データ(extern)” は利用可能である必要はあるか？
3. 更新(計算)操作
   - チャンク内の要素を更新する
   - 通常は計算の大部分
4. データ分散とタスクのスケジュール
   - チャンクと関連する計算をどのようにスレッドへ割り当てるか

1a. データ分解 – 粒度

データ分解の粒度はアプリケーションの性能と効率に重大な影響を与える

- 粗い粒度の分解
  - 少ない数の大きなチャンク
  - データ共有を最小限にすることが必要（同期/通信）
- 細かい粒度の分解
  - 小さなチャンクを多く呼び出す
  - スレッド数より多いチャンクを生成；負荷バランスを容易にする
  - 大量のデータ共有と同期

最適な粒度は計算を始める時点で数学的に導き出すのは困難かもしれない

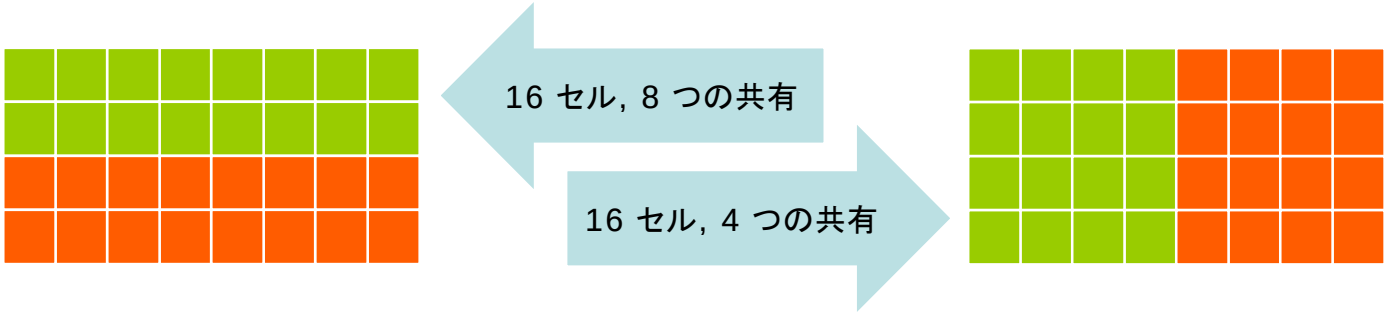
- 最適な分解サイズを求めるため試行する
- 実行時に負荷が変化する場合、分解サイズは調整できるようにする
  - 最良のスケーラビリティー

1b. データ分解 – 形状（配列）

チャンクの形は、通信と必要となる同期の量に影響する

- 共有されるデータは通常チャンクからの境界値
- 共有データはチャンクの上辺領域にスケールする
- 計算はチャンクの量にスケールする（チャンク内のデータ要素数）

面からボリュームへの比率を最大限にし、計算の同期比率を最大にする



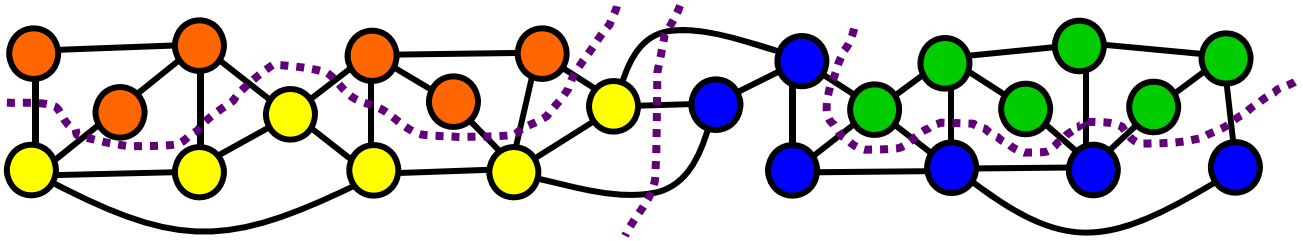
チャンクのサイズと形状は他の要因の影響を受ける

- キャッシュラインのサイズ、行優先 vs 列優先、前後のソースコード

1c. データ分解 – 形状(グラフ)

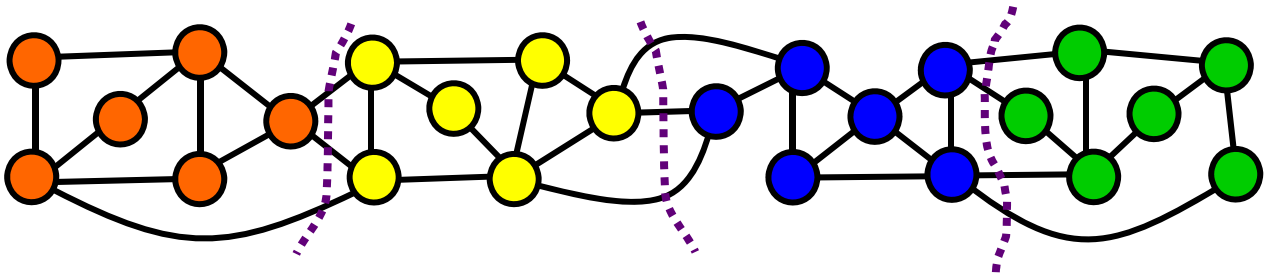
サブグラフの面はサブグラフを分割する切断境界であるかもしれない

- 境界にまたがってデータを共有しなければならないと仮定



ノード	隣接境界
6	9
6	12
6	11
6	8

最良のノード分割は、異なったスレッドに割り当てられる隣接境界を最小にすること



ノード	隣接境界
6	3
6	6
6	7
6	4

例: 分解の目的

以下のデータセットの分解スキームを説明:

- オーストラリアの海岸線と陸地を示す正方形グリッド
- 図書館の書籍目録
- 3000 の米国の都市(ノード)とそれらの間の道路(境界)の地図を示すグラフを作成

2. 交換操作

ジオメトリー分解パターンを正しく利用する重要な要素

- 隣接するチャンク間の効率よいデータ交換を確実にする
- データをローカル構造にコピーする、もしくは必要な時にデータにアクセスするか?

データをローカル構造にコピーする

- データが更新操作前に参照可能で、コピー中に変更されない場合
- スレッドごとに追加のメモリーが必要だが、コピーされたデータに競合は起こらない

必要な時にアクセスする

- スレッド間で共有メモリーの利点を活用できる
- データが必要になるまでスレッド間のアクセスを遅らせる

スレッド間の調停を行い、データが利用可能になるまで
変更されないことを確実にする

3. 更新操作

関連するタスクの実行は同時にデータ構造を更新する

交換と更新のためコード記述における考察

- タスクの開始時にすべてのデータが利用可能で演算中に変更されないならば、並列化は容易で効率的である
- 非ローカルデータが更新計算の前に他のチャンクからコピーされるならば、更新前にデータ収集フェーズを追加する
- 非ローカルデータが更新操作中にアクセスされるならば、適切なアクセスプロトコルを利用して(相互排他？)、正しいデータが検出できることを確実にして、
  - 交換と更新計算の混在はアプリケーションのロジックを複雑にする。特に正しいデータを取得するのを確実にする場合

4. データ分散とタスクスケジュール

どのスレッドがどのデータチャンクを更新するか決定する

静的な分散は簡単

- 交換の際の調整は決定しており実装は容易
- タスク中の計算量が一定である場合が最も適切

動的な分散はロードバランスに優れる

- スレッド数よりも多くのタスクが必要
- 交換操作は複雑になる
  - 非ローカルデータの更新が確実に完了していることが必須

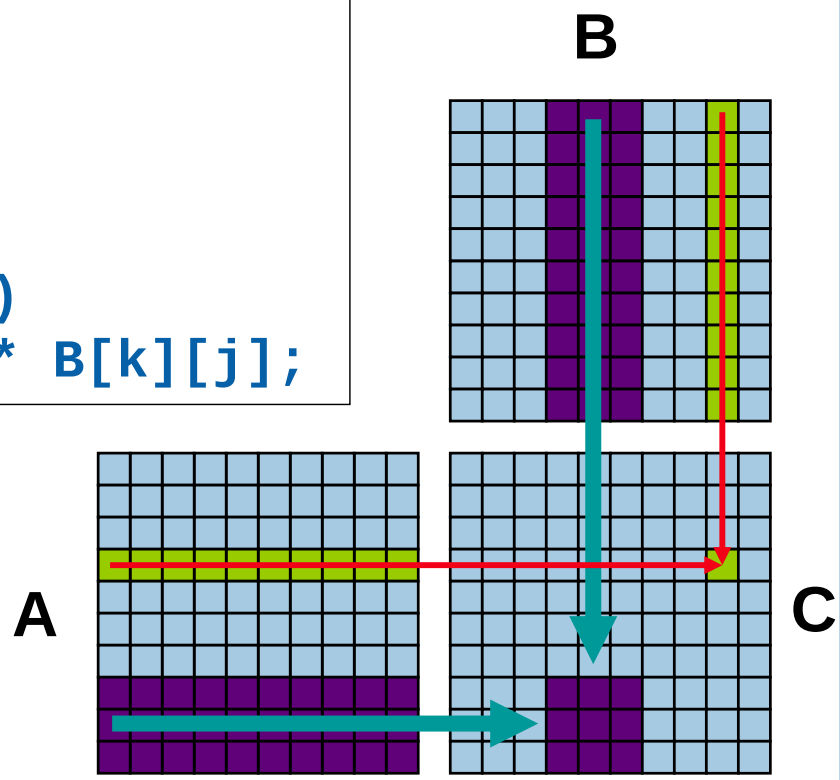
初期の静的分散を動的に再分散するのは?

- チャンクあたりの仕事量が計算の途中で変化するなら
  - 例: 列/行を解決する行列計算
- アプリケーションは負荷分散のためタスクやデータの分散を再構成できる
- 性能上の利得は再分配のオーバーヘッドに打ち勝たなければならない

例: マトリクス乗算

```
for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      C[i][j] = 0.0;

for (i = 0; i < M; i++)
   for (j = 0; j < N; j++)
      for (k = 0; k < L; k++)
         C[i][j] += A[i][k] * B[k][j];
```



独立して演算できるのは？

- C のそれぞれの要素
- C の列（B のすべてを利用）
- C の行（A のすべてを利用）
- C のブロック

例: マトリクス乗算

どれくらいの大きさのデータ構造が更新されるか?

データ構造はどのように分解されるべきか? 最適な粒度は?

どのデータがデータチャンクに関連するタスク間で共有する必要があるか? 交換が必要な非ローカルデータはあるか?

チャンクをどのようにスレッドに割り当てるか? 静的か動的か?

例: マトリクス乗算のソリューション

どれくらいの大きさのデータ構造が更新されるか?

- C の行列

データ構造はどのように分解されるべきか? 最適な粒度は?

- 個々の要素は粒度が細かすぎるように思われる
- 列のグループは列アクセスされ、より粗い粒度となる
- キャッシュサイズとその活用の視点からブロックが利用された

どのデータがデータチャンクに関連するタスク間で共有する必要があるか? 交換が必要な非ローカルデータはあるか?

- 配列 A と B の要素は共有される(しかし更新されない)
- メモリーに収まれば交換することなく共有できる

チャンクをどのようにスレッドに割り当てるか? 静的か動的か?

- データを交換する必要が無いなら、静的な分割は最も簡単

例: マトリクス乗算

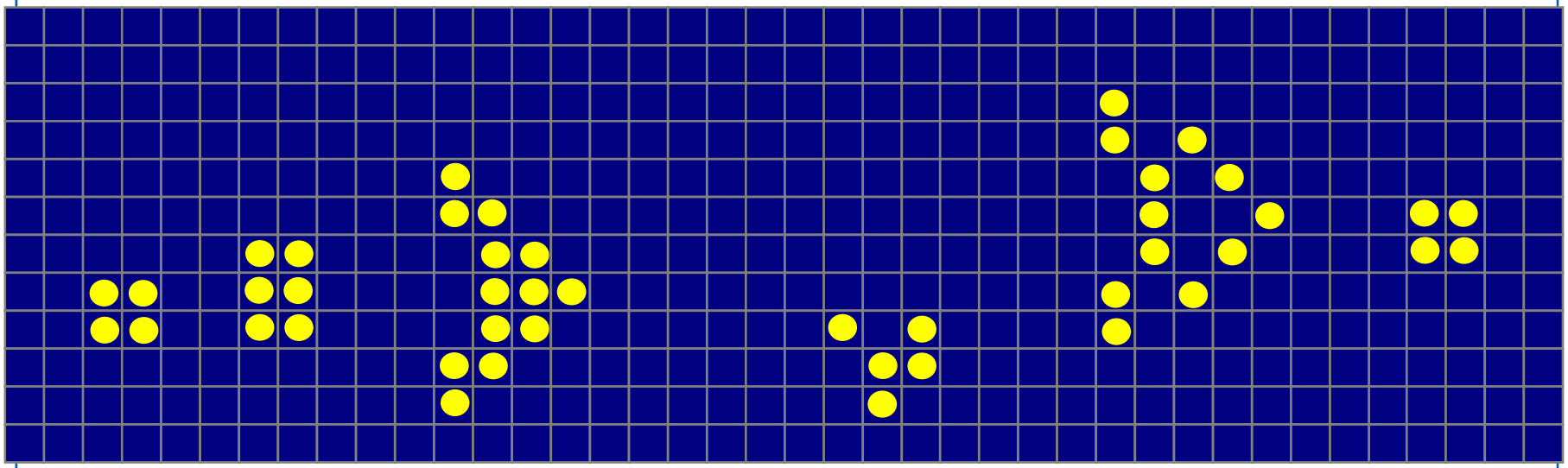
列のグループを個々のスレッドに割り当てるように、一意なスレッド ID を基に “i” のループ領域を変更する

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        C[i][j] = 0.0;

for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < L; k++)
            C[i][j] += A[i][k] * B[k][j];
```

各スレッドは 3 つのループのインデックス変数のローカルコピーを必要とする

例: Game of Life



同時に更新可能な大きなデータセットはどれか?
このデータはどのように分解されなければならないか?
データとタスクをスレッドで実行するため、どのようにスケジュールするか?
負荷バランスのためどの様な問題を考慮しなければならないか?

内容

パターン言語構造

タスク並列パターン

ジオメトリー分解パターン

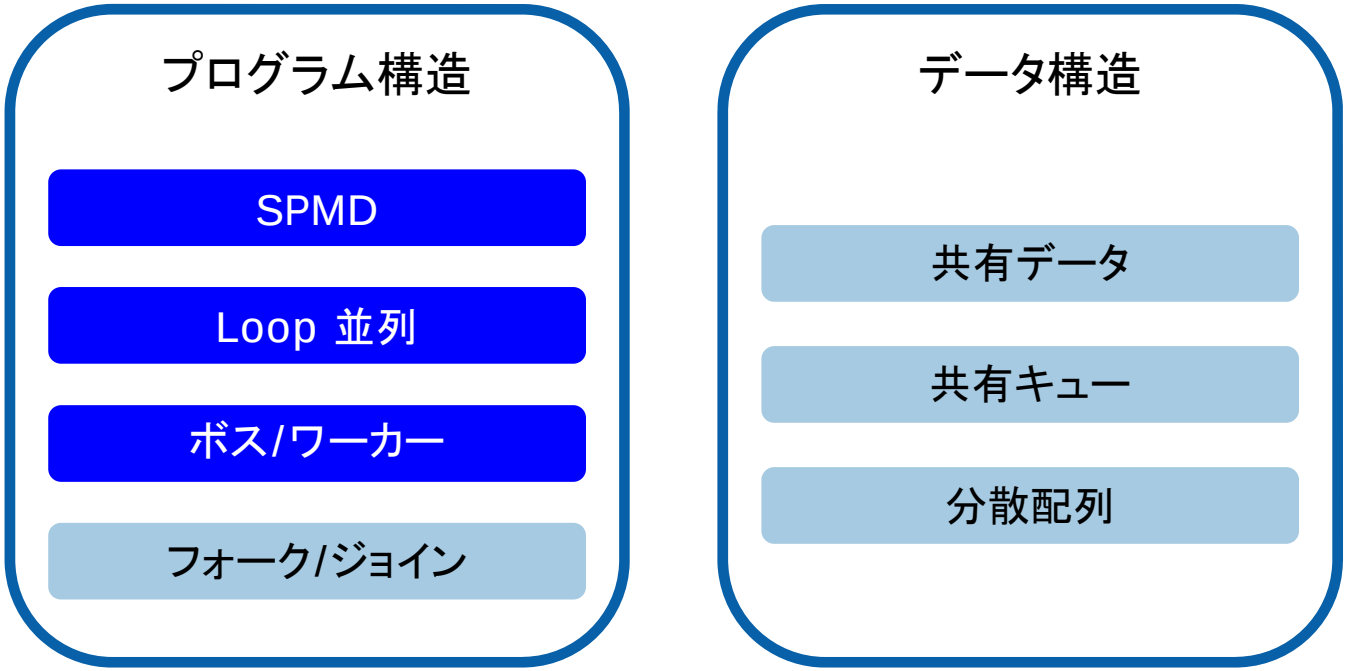
構造のサポート

- SPMD
- Loop 並列
- ボス/ワーカー

まとめ

設計領域でサポートする構造化

ソースコードを組織化し、プログラム構造とデータ構造を分類するため、ハイレベルの実装要素が利用されてきた



SPMD パターン

問題

- コア計算を容易に実装するため、スレッド間の相互作用をどのように並列プログラムに構造化するか?

関係

- スレッド間のループ反復は効率よく、そして正しい結果を導くように調整されなければならない。
- 多くの並列アルゴリズムではスレッド間で同様の操作が行われるが、スレッドとデータのサブセットでは少し異なった要件が求められる
- 複雑なアルゴリズムでは、スレッド間で大きく異なった命令とデータストリームが求められる

効力

- 単独のソースコードでは、複数のソースコードにまたがる場合よりもスケーラビリティーと保守性に求められる矛盾するバランスを取るのは容易である

SPMD パターン – ソリューション

シングル・アプリケーションでは、ソース中でコードと機能を実行するスレッドを生成する

- マルチスレッド・プログラミングの本質において、SPMD パターンはデフォルトである

SPMDパターンを実装するコーディング要件

- **ユニークな識別子を取得**
  - 各スレッドに ID 番号(ランク)を割り当てる。通常 0 から N-1 の番号
- **異なるスレッドの振る舞いを区分するために ID を利用する**
  - コードブロックを割り当てるため分岐ステートメントを利用できる
  - ループのインデックス計算に ID を利用して、スレッドに割り当てるループ反復を分割できる
- **データ分散**
  - グローバルデータをチャンクに分割し、ID を基にチャンクをアクセスする
- **ファイナライズ**
  - グローバルに関連するデータがスレッド間に分散された場合、各部分的な演算結果は再結合する必要がある

SPMD パターン – Discussion

SPMD プログラムは、より密接にメッセージパッシング環境分散に適合する

スレッドをスポーンするシングル・プログラムは、スレッドプログラミングのデフォルトである

- 同一プロセス内のスレッドは、コードとグローバルデータを共有する
- 一般的なスレッド化モデルであり OpenMP* より関連深い

ループ並列パターン

問題

- 演算主体の実行時間を占有するループを持つシリアルプログラムを並列化する場合、どのように構造化するか?

関係

- 膨大な科学および技術的な数値を扱うアプリケーションでは、繰り返し構造(ループベース)を持つ
- 各スレッドで異なるループ反復を実行するように、ループを変換することによりシーケンシャルコードを並列コードに進化させることが目的
  - ループ反復は並列タスクのように動作する
    - 演算主体
    - 明確な十分な同時性
    - 大部分が独立
  - (もしあれば)ループ反復間の依存性を排除する

このパターンは、OpenMP や インテル TBB を利用する場合と関連する

ループ並列パターン – 効力

シーケンシャルと等価
- 1 もしくは多くのスレッドで実行した場合でも同じ結果をもたらすプログラム
  - 四捨五入によるいくつかのエラーは許容される
- シーケンシャルと等価なコードは、書きやすく保守しやすい

インクリメンタルな並列性
- ループ変換が行われ同時にテストされているならば、正当性の保守は容易に行える
- 一般的なスレッド化モデルとデータのローカル・チャンクへの分割では困難な場合がある（“コード全体”の変換が必要）

メモリー効率
- データのアクセス・パターンはメモリー階層に合っていなければならない
- スレッドへのループ反復の分散は、シーケンシャルなアクセス・パターンと異なるかもしれない
  - 適切にアクセスするためループの再構成が必要

ループ並列パターン – 解決策

プログラミング・スタイルは、OpenMP や インテル TBB と一致する必要がある

- 一般的なスレッド化モデルでは、ループ反復の割り当てはプログラマーによって行われる

ループ並列パターンを実装するコーディング要件

- **ホットスポットを見つける**
  - コードの調査、アルゴリズムの理解、もしくはプロファイル・ツールを利用することで、最も演算主体のループを特定する
- **ループ反復間の依存性を排除する**
  - ループ反復間の依存性やデータの競合を見つける
  - 依存性を排除するか、同期によってデータを保護する
- **ループを並列化する**
  - ループ反復を各スレッドに分割する
- **ループのスケジュールを最適化**
  - 負荷バランスが適切になるようにループ反復をスレッドに割り当てる

ループ並列パターン – 他の考察

各ループ反復の演算時間は、スレッド管理のオーバーヘッドと反復をスレッドへ割り当てる時間を打ち消すのに十分でなければならない

- 解決方法: ループを統合する
  - 同じインデックス範囲を持つ複数のループシーケンスを 1 のループに統合するとこで、反復あたりの仕事量を増やすことができる

ループ反復回数が大きければ大きいほど、スケジューリングの柔軟性が高まる

- 解決方法: 入れ子のループを融合する
  - ネストしたループを 1 のループに統合することで、大きなループカウントを得ることができる
  - 大きなループカウントは、オーバーヘッドを打ち消す可能性を高める

ループ統合の例

素数値回数のループ反復は効率よく負荷バランスを取れない

```
#define N 23
#define M 1000
...
for (k = 0; k < N; k++)
   for (j = 0; j < M; j++)
      w_new[k][j] = DoSomeWork(w[k][j], k, j);
```

内側ループの並列化は? スレッド管理のオーバーヘッドを打ち消すのに十分な反復数か?

大きな値の反復数は、負荷バランスとオーバーヘッドの隠匿によい影響を与える

```
#define N 23
#define M 1000
...
for (kj = 0; kj < N*M; kj++) {
   k = kj / M;
   j = kj % M;
   w_new[k][j] = DoSomeWork(w[k][j], k, j);
```

内側のループの上限値を求める
DIV と MOD

この演算はオーバーヘッドの一部

ループ反復を分割する - OpenMP

```
#define N 23
#define M 1000
. . .
#pragma omp for private(k,j) schedule(static, 500)
for (kj = 0; kj < N*M; kj++) {
   k = kj / M;
   j = kj % M;
   w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

ループ反復を各スレッドに分散するため、OpenMP ワークシェア構文を指示する pragma を追加する

- 必要であれば、データ環境とスケジュールを指示する句を追加する

ループ反復を分割する – 手動でチャンクへ分割

```
#define N 23
#define M 1000
#define numThreads 4

int start = ((N*M)/numThreads) *  myID;
int end   = ((N*M)/numThreads) * (myID+1);
. . .
if (myID == (numThreads-1))  end = N*M;
. . .
for (kj = start; kj < end; kj++) {
   k = kj / M; j = kj % M;
   w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

この変数は各スレッドでプライベート・コピーを持たなければならない

すべての反復が含まれることを確実にする

スレッド数とスレッド ID 番号を基に処理するループの start と end 位置を計算する

ループ反復を分割する – 負荷バランス

23 * 5323 = 122,429 の反復

```
#define N 23
#define M 5323
#define numThreads 16
...
int start = ((N*M)/numThreads) *  myID;
int end   = ((N*M)/numThreads) * (myID+1);
```

122,429 / 16 = 7651.8125

ID	(float) start	(float) end	割り当てられた反復
0	0.000	7651.813	7651
1	7651.813	15303.625	7652
2	15303.625	22955.438	7652
...	...	...	...
5	38259.063	45910.875	7651
...	...	...	...
14	107125.375	114777.188	7652
15	114777.188	122429.000	7652

	割り当てられ[illegible]反復
[illegible]ead	
...	
14	7651
15	7664

13 の追加ループ！

```
oat)(N*M)/numThreads) *  myID;
oat)(N*M)/numThreads) * (myID+1);
ds-1))  end = N*M;
 end; kj++) { ... }
```

ループ反復を分割する – ラウンドロビン

```
#define N 23
#define M 1000
#define numThreads 4


for (kj = myID; kj < N*M; kj+=numThreads) {
   k = kj / M;
   j = kj % M;
   w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

各スレッドは (numThreads) 番目先の反復を処理

myID の範囲は、[0..numThreads-1]

各ループ反復間でほぼ均一な負荷バランス

- 任意のスレッド間で ±1 の誤差
- 容易に記述でき、処理されない反復がない

メモリーアクセスとキャッシュに関する問題の可能性に注意

例: ループ並列

次の演算でスレッドに反復を分割する場合、どのような割り当て方法が利用できるか?

コンピュータ・アニメーション・フィルムにおいて次のフレームを計算する

Loop 列方向

Loop 行方向のピクセル (カラム)

ピクセルを計算

図書目録を検索してどれくらいの書籍が “ループ” を持っているか見つける

Loop アルファベット文字列

Loop 著者名の最初の文字

Loop 著者別

タイトルが “ループ” を持っていたらグローバル値をカウント

ボス / ワーカーのパターン

問題

- プログラムの設計が、動的に 1 組のタスク上で作業バランスをよる必要がある場合、プログラムはどのように構成されるべきか?

関連

- タスクの負荷バランスを取る場合、シリアル領域とオーバーヘッドに影響を受ける
  - タスクの負荷は可変で予測できない
  - 演算主体のコード領域はシリアルループへ割り当てない
  - システム外のコアの能力は、様々で、変化しやすく、予測できない。
- タスク並列パターンに関連する

効力

- タスクごとの予測できない作業では、動的な割り当てロジックが必要
- 負荷バランスを取る操作は同期オーバーヘッドの要因となる
  - 少ない大きなタスクではこの影響は少なくなるが、オプションを制限する
- 最適な負荷バランスとコード保守の容易度にはトレードオフがある

ボス / ワーカーのパターン – 解決方法

ボススレッド
- 演算を開始
- ワーカースレッドに割り当てられるタスク分割を制御

ワーカースレッド
- ワーカーはタスクが提供される間ループし続ける
  - ボススレッドからのタスクを受け入れ
  - 割り当てられたタスクを実行

プログラミング上の考察
- タスク分割
- ワーカーの完了と配備を検出

ボス / ワーカーのパターンにおけるタスク分割

ランデブー

- 2 のスレッドがデータを交換するため“遭遇”する
  - 相互に待ちあう(最初に到着したスレッドは 2 番目を待たなければならない)
  - 1 の双方向トランザクションが完了する
  - 他のスレッドの相互排他がランデブーを試みる

タスクのバッグ

- タスクの共有データ構造(キュー)
  - ボススレッドは共有構造へタスクをロードする
  - ワーカースレッドは作業が必要になったら、新しいタスクを取り出す(取り除く)
- 生産/消費: ワーカーが実行する間、ボスはタスクを追加し続ける

共有された単純カウンター

- ボスがタスクのインデックスを設定する
- ワーカーは次のタスクを取得するため、カウンターにアクセスする(インクリメントする)

ボス / ワーカーのパターンにおける完了の検知

ランデブー

- 各ワーカーに完了タスク(ポイズンピル)を送信する
- タスクを受け取る側であるワーカーは、新規のタスクか完了かをチェックする

タスクのバッグ

- すべてのタスクが初めにバッグにロードされ、空バッグシグナルで演算を終了する
- すべてのタスクによる演算が終了したのち、キューに完了タスクが追加される
  - 単純にするため、ワーカースレッドごとに 1 の完了タスクが追加される

共有された単純カウンター

- 既知のタスク数よりも大きなカウンターは完了を通知する

例: ランデブーを実装するボスコード

```
LockSyncObject(BWlock);  // num_waiting カウンターへの同期アクセス
num_waiting++;
if (num_waiting !=2) {   // ワーカー待ちがない場合...
  UnlockSyncObject(BWLock);
  SleepUntilWorkerArrives(); // ...待ち
}
else {
  WakeUpSleepingWorkerThread();  // ワーカーを起動
  num_waiting = 0;               // 次のランデブーのためリセット
  UnlockSyncObject(BWLock);
}
```

<データ転送、新しいタスク割り当てもしくは完了を送信>

例: ランデブーを実装するスレッドコード

```
LockSyncObject(WorkerLock);  // 他のワーカーとの排他制御
 LockSyncObject(BWlock);  // num_waiting カウンターへの同期アクセス
 num_waiting++;
 if (num_waiting !=2) {   // ボスが待機していない場合...
   UnlockSyncObject(BWLock);
   SleepUntilBossArrives(); // ...待機
 }
 else {
   WakeUpSleepingBossThread();     // ボスを起動
   num_waiting = 0;                // 次のランデブーのためリセット
   UnlockSyncObject(BWLock);
 }
 <データ転送、新しいタスク割り当てもしくは完了を送信>
UnlockSyncObject(WorkerLock);  // 次のワーカーのランデブーを許可
```

例: ボス / ワーカー

次のような演算処理のループ反復をスレッドに分割する場合、どのようなタスク分割方法が利用できる?

マンデルブロー集合計算

- Loop 列方向
  - Loop 行方向のピクセル(カラム)
    - ピクセルを計算

図書目録を検索してどれくらいの書籍が “ループ” を持っているか見つける

- Loop アルファベット文字列
  - Loop 著者名の最初の文字
    - Loop 著者別
      - タイトルが “ループ” を持っていたらグローバル値をカウント

内容


intel®
Leap ahead™

著作権と商標について

- 本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスを許諾するものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責任を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商適格性、あらゆる特許権、著作権、その他知的財産権の非侵害性への保証を含む)に関してもいかなる責任も負いません。インテル製品は、医療、救命、延命措置などの目的への使用を前提としたものではありません。
- インテル製品は、予告なく仕様や説明が変更されることがあります。
- インテル製品は、予告なく仕様が変更される場合があります。本資料に記載されているすべての製品、日付、および数値は、現在の予想に基づくものであり、計画以外の目的ではご利用になれません。
- 本資料に掲載されているインテル製品は、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公開されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。
- Nehalem、Fox Hollow、Lynnfield, Boxboro、Westmere、Sandy Bridge、Tylersburg およびその他のコード名は、開発中で一般に公開されていない製品を特定するためにインテル内部でのみ使用されているものです。顧客、ライセンシー、その他の第三者により、いかなる製品またはサービスの広告、販促活動、あるいはマーケティングにおいてコード名を使用することは許可されていません。また、かかるインテル内部の開発コード名の使用はユーザー側の責任となります。
- 性能に関するテストや評価は、特定のコンピューター・システム、コンポーネント、またはそれらを組み合わせて行ったものであり、このテストによるインテル製品の性能の概算の値を表しているものです。システム・ハードウェアの設計、ソフトウェア、構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。
- Intel、インテル、Intel ロゴ、Intel Atom は、アメリカ合衆国およびその他の国における Intel Corporation の商標です。
- Windows は、米国 Microsoft Corporation および / またはその関連会社の商標です。