並列スケルトンライブラリSkeToにおける可変長リストの実装と評価

岩崎研究室
丹野治門

2009年2月6日（金）

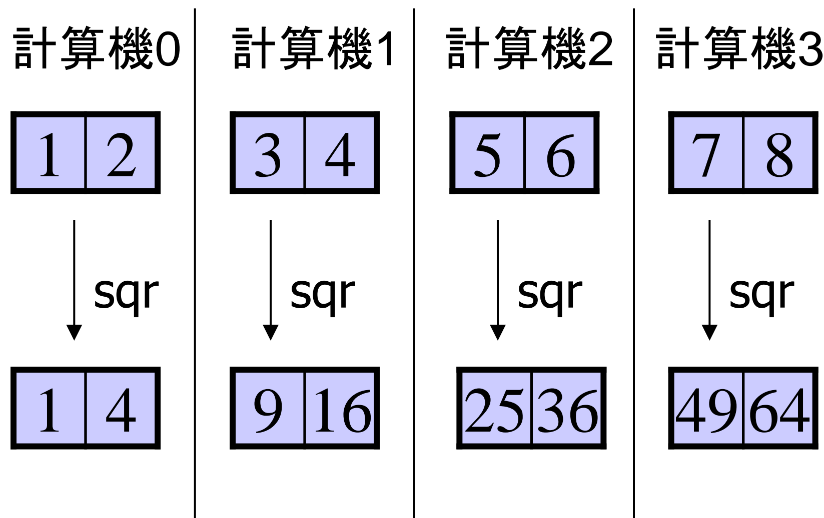
背景

- 並列プログラミングは難しい
  - データの分散、計算機間通信、同期制御
- 並列スケルトンが考案された
  - 一般的な並列処理のパターンを実装したもの
    - C++上の実装 SkeTo [06] (分散メモリ環境が対象)
  - 逐次的な感覚で簡潔に並列プログラムを記述できる
  - 様々なデータ構造を扱える
    - リスト(固定長一次元配列)、行列、木

並列スケルトンの例

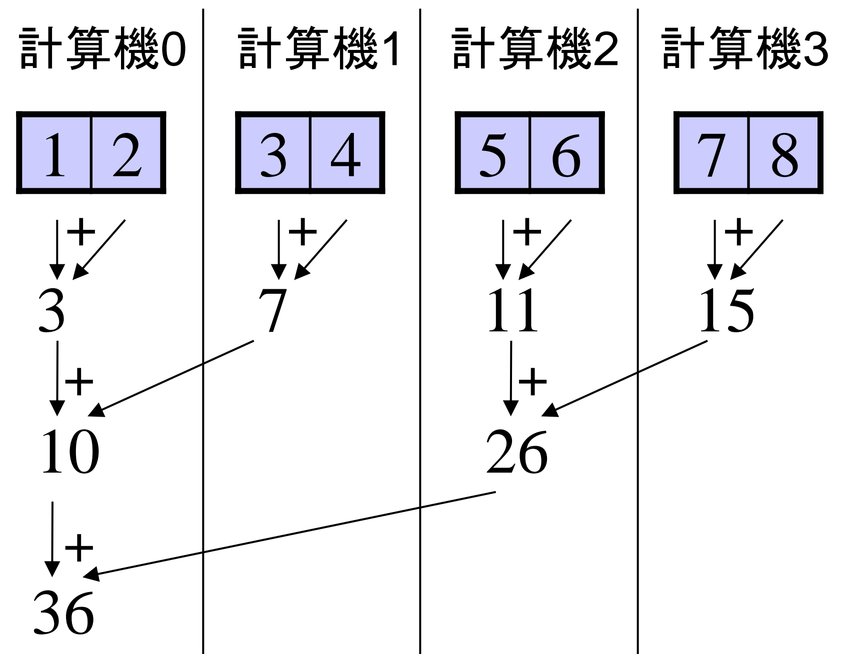
- リスト上のmapスケルトン
  - 全要素に関数を適用

map sqr [1,2,3,4,5,6,7,8]
= [1,4,9,16,25,36,49,64]



- リスト上のreduceスケルトン
  - 全要素を結合的な二項演算子で結合

reduce (+) [1,2,3,4,5,6,7,8]
= 1+2+3+4+5+6+7+8 = 36



固定長リストの問題点

- 自然数から双子素数を求める問題
  - 双子素数とは差が2である素数の組 (3,5) (5,7) (11,13)…

```
numbers = [2, 3, ... , 13, 14, 15]
do{  //エラトステネスの篩
  prime ← numbersから先頭要素を取り出す
  numbersからprimeで割り切れる要素を全て除く
  primesにprimeを追加する
}while(prime <= sqrt_size);
primesにnumbersを連結する
//primesは[2,3,5,7,11,13]
twin_primes = primesの中で隣同士のペアをつくる
twin_primesから差が2でないものを除く
//couple_primes = [(3,5), (5,7), (11,13)]
```

リストが縮む操作

リストが伸びる操作

リストが伸縮する操作は無く、このような記述はできない

固定長リストでは解きにくい問題

Type Ⅰ : 集合から，部分集合を取り出す問題

- （例）双子素数問題，包装法（点集合の凸包計算）

➔リスト要素の追加削除，リストの連結が必要

Type Ⅱ : 全解探索問題

- （例）騎士巡歴

➔解候補を要素とするリストが計算途中で伸縮

Type Ⅲ : 計算量が要素ごとに大きく異なる反復計算

- （例）マンデルブロ集合，ジュリア集合

➔一定間隔で計算が終了した要素を除き、残りだけ計算を続けられれば効率的

本研究の目的と方針

目的

- 可変長リストを扱えるようにする

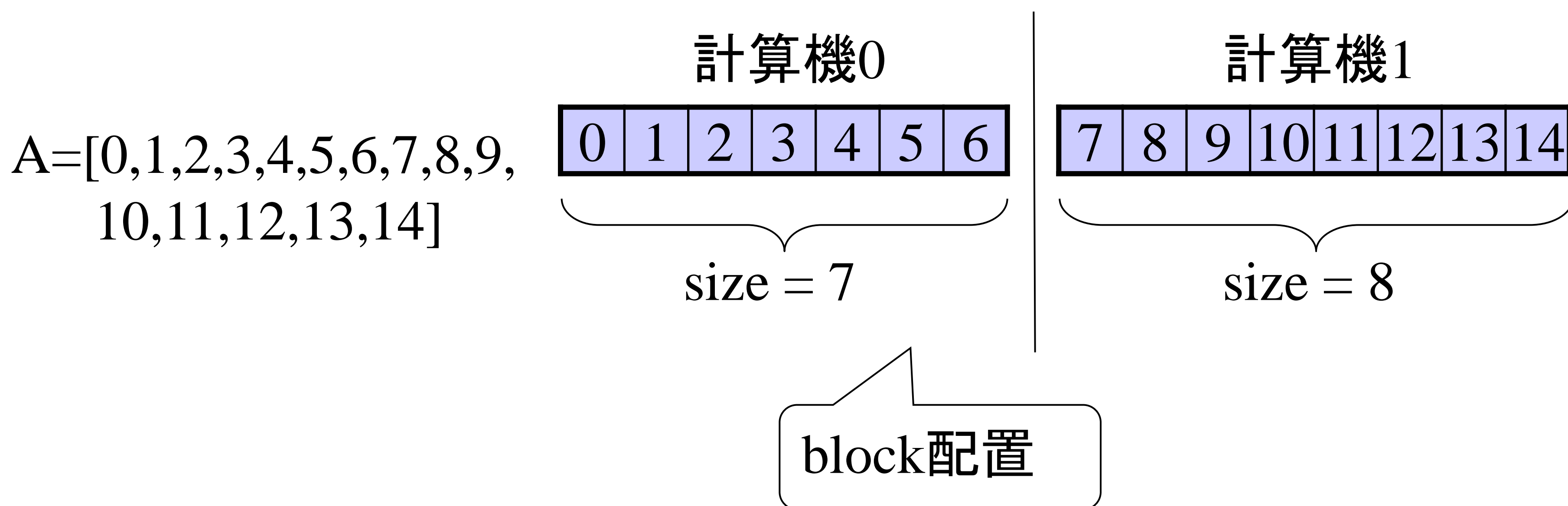
➔より多くの応用問題を効率よく解けるようにする

方針

- SkeToの既存リストを可変長リストへ拡張
  - リストを表現するデータ構造を変更
  - 条件に合う要素を抜き出すfilterスケルトンを追加
    (例)filter odd [1,2,3,4,5,6,7,8] => [1,3,5,7]
  - 2つのリストを連結する操作appendを追加
    (例)append [1,2] [3,4,5,6] => [1,2,3,4,5,6]
  - データ配置不均衡時の自動データ再配置機能を追加

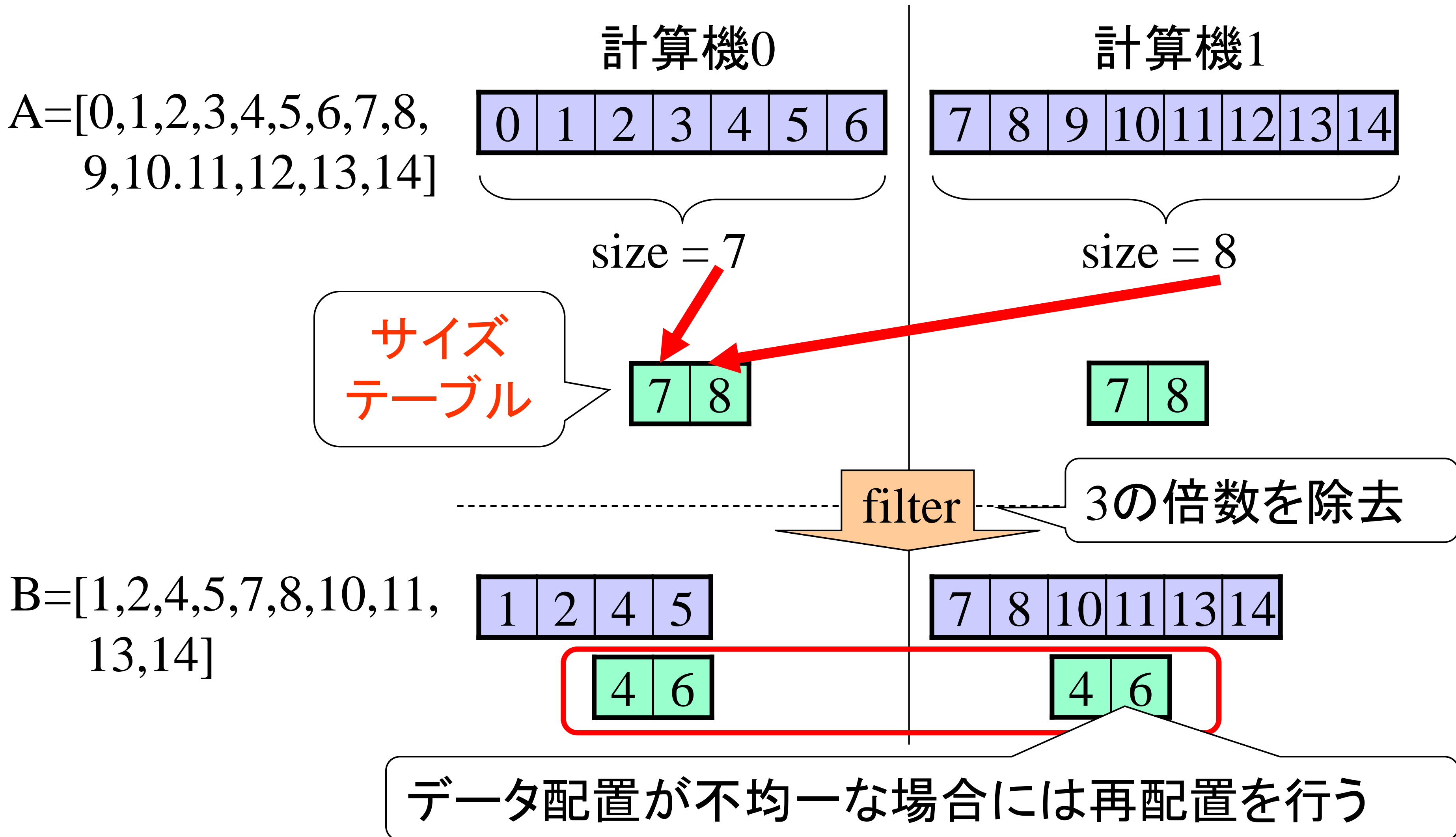
固定長リスト(既存)のデータ構造

- リストの各要素はリスト生成時にほぼ均等に配置される
- 計算途中でこのデータ配置が変わることはない



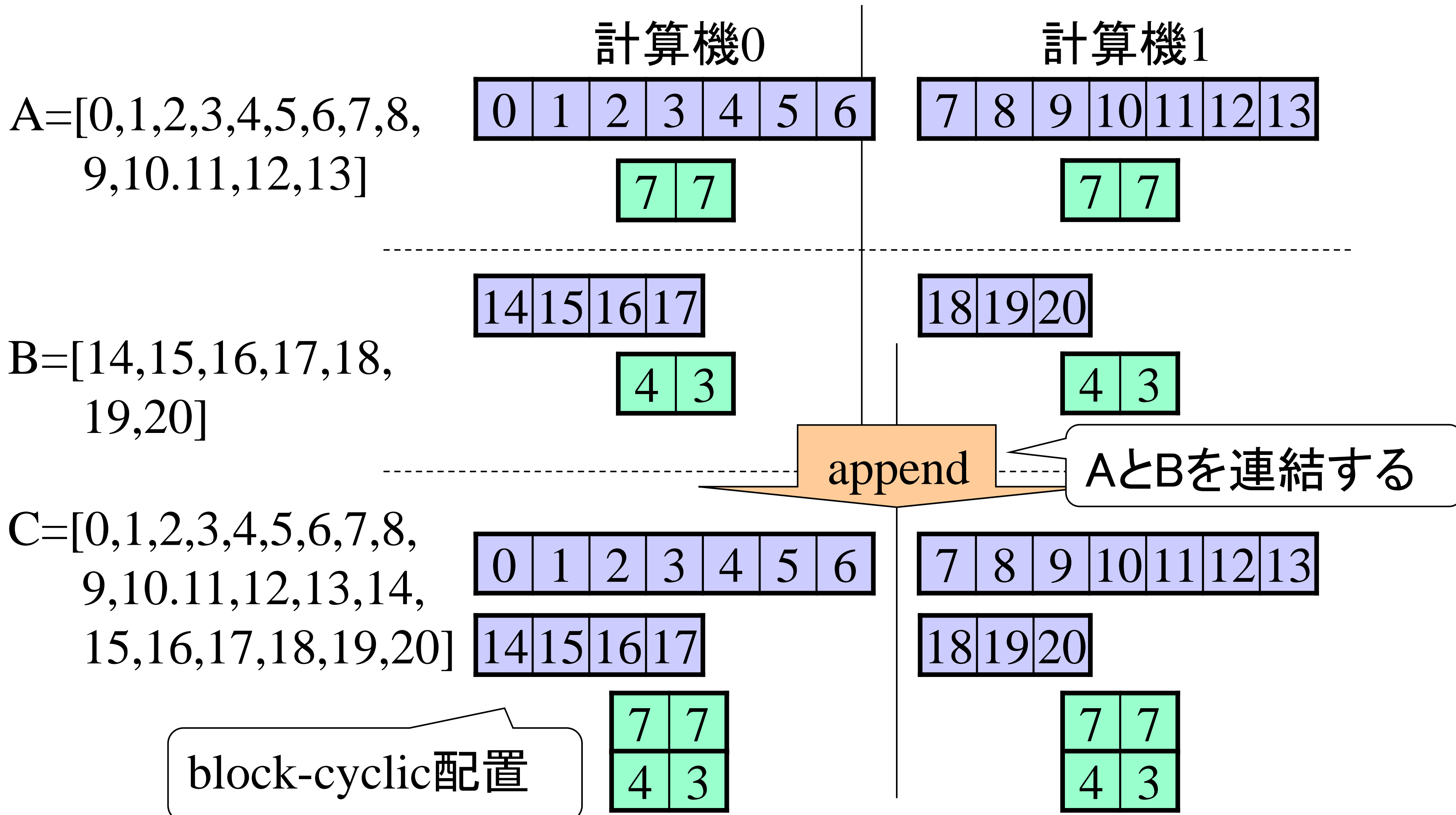
可変長リストのデータ構造(1)

- 各計算機は全計算機がもつデータのサイズを保持



可変長リストのデータ構造(2)

- リストを連結しても、すぐにはデータの移動を行わない



双子素数を求めるプログラム

```
…
do{ //エラトステネスの篩
    prime = numbers->pop_front();
    list_skeletons::filter(IsNotMultipleOf(prime), numbers);
    primes->push_back(prime);
} while(prime <= sqrt_size);
primes->append(numbers);
//primesは素数のリスト
dist_list<int>* dup_primes = primes->clone<int>();
dup_primes->pop_front();
couple_primes = list_skeletons::zip(primes, dup_primes);
list_skeletons::filter(couple_primes, IsCouple());
//couple_primesは双子素数のリスト
```

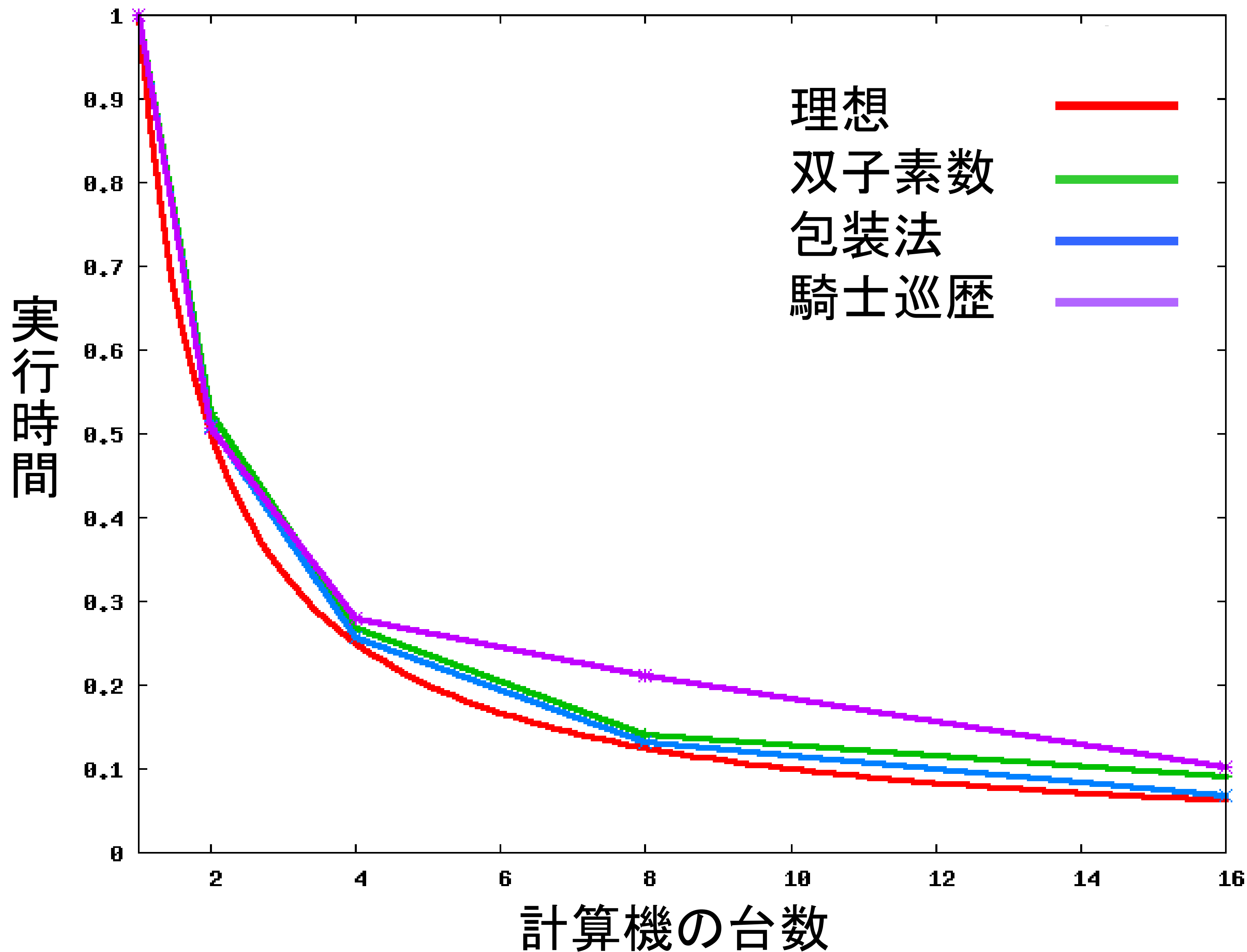
リストが縮む操作

リストが伸びる操作

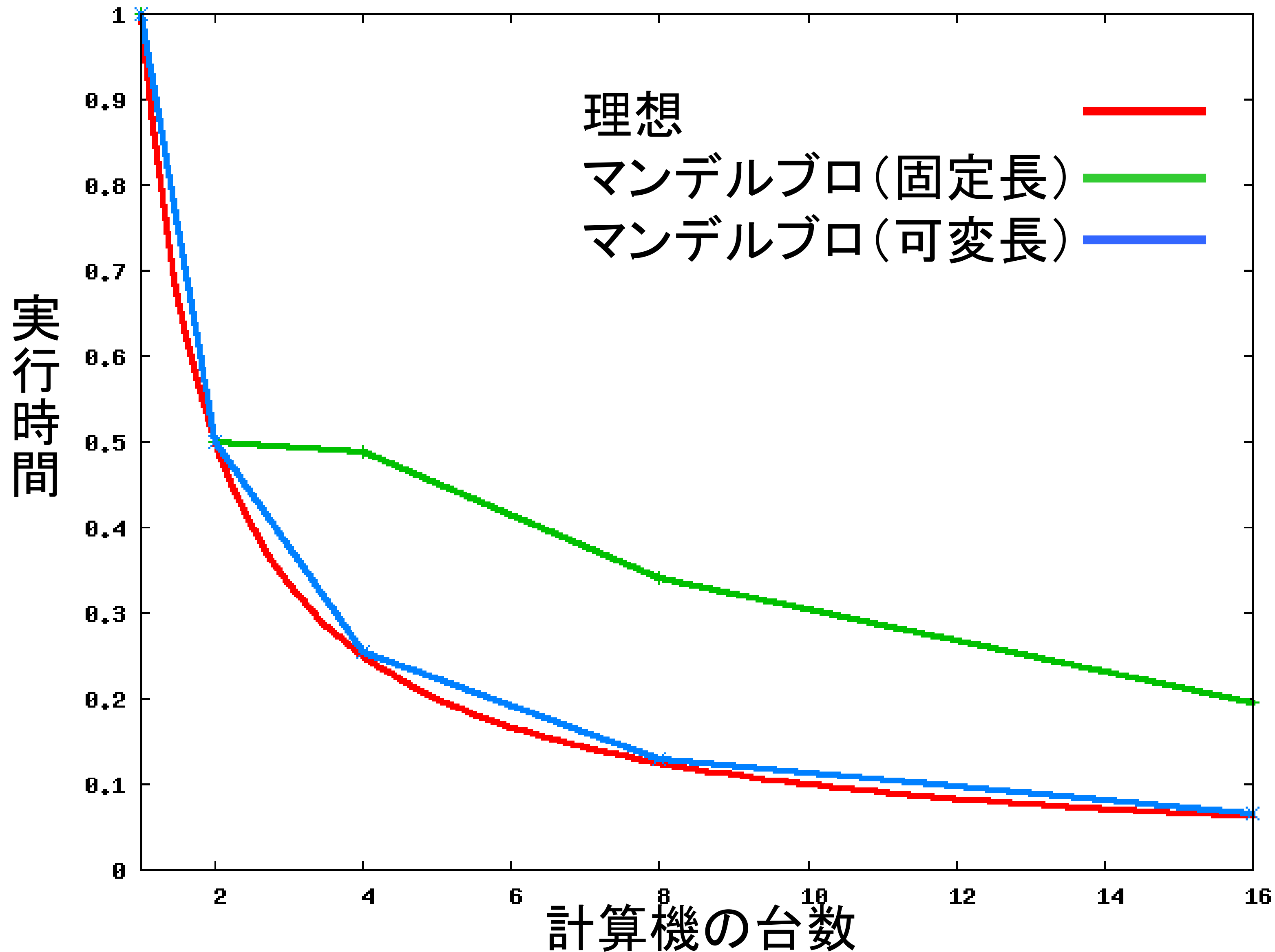
性能評価（1）

- 実験環境
  - Pentium4 3.0GHz、メモリ 1GB、Linux 2.6.8
- TypeⅠ
  - 双子素数問題（1000万までの数）
  - 包装法（100万個の二次元座標の点）
- TypeⅡ
  - 騎士巡歴（5×6マス）
- TypeⅢ
  - マンデルブロ集合（1000×1000要素，10,000回計算）
    - 可変長リストで100回の計算ごとに，計算が終了した要素をfilterで除去
    - 固定長リストで10,000回計算

性能評価(2)



性能評価(3)



関連研究

- Muesli [05]、P3L [96]
  - リスト、行列を対象とした並列スケルトンライブラリ
  - リストサイズは固定長(一次元配列)
- 不均質なクラスタ環境を対象とするデータ再配置による動的負荷分散機構の設計と実装 [野口ら 06]
  - ループの残数情報等を利用して動的に負荷分散
  - 数値計算向きであり、配列のサイズは固定長

まとめ

- 可変長リストを提案、実装
  - 可変長リスト上のスケルトンと操作を提案
    - filter、appendなど
  - リストの伸縮，連結を効率よく行うデータ構造を採用
    - サイズテーブル、block-cyclic配置
  - 3タイプの問題（Ⅰ、Ⅱ、Ⅲ）を解き有効性を確認
    - 双子素数問題、騎士巡歴、マンデルブロ集合など

性能評価（２）

マンデルブロ集合（1000×1000要素，10,000回反復計算）

固定長リスト　10000回，そのまま反復計算

	1プロセス	2プロセス	4プロセス	8プロセス	16プロセス
実行時間(s)	61.2	30.6	29.9	20.85	11.98

可変長リスト　100回の反復計算ごとに計算不要な要素をfilterで除去

	1プロセス	2プロセス	4プロセス	8プロセス	16プロセス
実行時間(s)	63.4	31.8	16.2	8.2	4.2

性能評価(1)

- 実験環境
  - Pentium4 3.0Ghz、メモリ 1GByte、Linux 2.6.8

双子素数問題（1000万までの数）

	1プロセス	2プロセス	4プロセス	8プロセス	16プロセス
実行時間(s)	16.86	8.84	4.52	2.38	1.52

包装法（100万個の二次元座標の点)

	1プロセス	2プロセス	4プロセス	8プロセス	16プロセス
実行時間(s)	8.59	4.38	2.21	1.13	0.59

騎士巡歴（5 × 6 マス）

	1プロセス	2プロセス	4プロセス	8プロセス	16プロセス
実行時間(s)	11.92	6.05	3.34	2.53	1.22

可変長リストへの操作とスケルトン

– 条件に合う要素を抜き出すfilter

（例）filter odd [1,2,3,4,5,6,7,8] => [1,3,5,7]

– 各要素から新しい要素を生成するconcatmap

（例）concatmap plusminus [1,2] => [1,−1,2,−2]

– 2つのリストを結合する操作append

（例）append [1,2] [3,4,5,6] => [1,2,3,4,5,6]

– リストの先頭や末尾に要素を追加，削除する

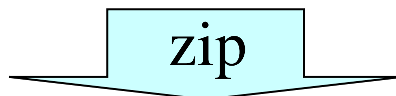
  - push_back，pop_back，push_front，pop_front

エラトステネスのふるい

- 双子の素数(差が2の素数の組)
  - 素数リスト複製し，それらをzipし，filterで抽出

primes = [2,3,5,7,11,13,17,...]

primes2 = [3,5,7,11,13,17,...]



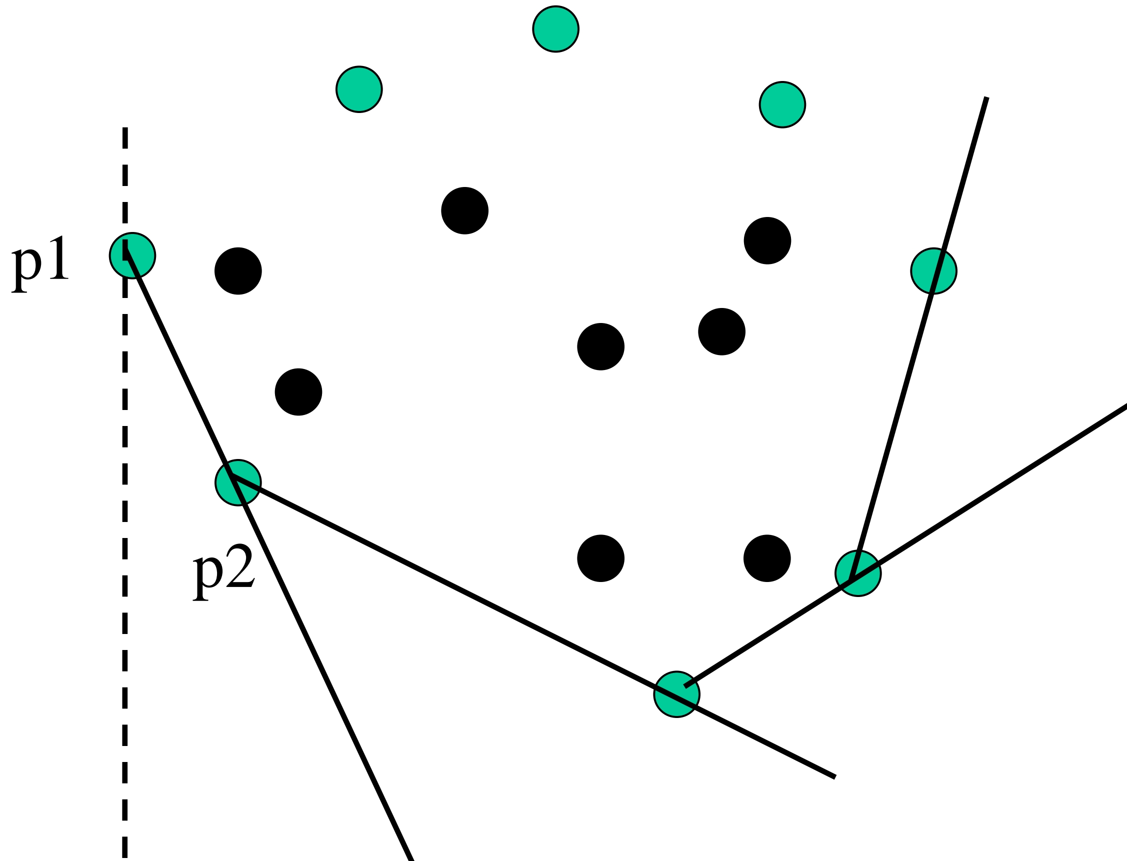
tuple_primes = [(2,3), (3,5), (5,7), (7,11), (11,13),...]

- ゼータ関数
  - 素数リストにreduceを適用

$$\frac{1}{\zeta(n)} = 1 + \left(1 - \frac{1}{2^n}\right)\left(1 - \frac{1}{3^n}\right)\left(1 - \frac{1}{5^n}\right)\cdots$$

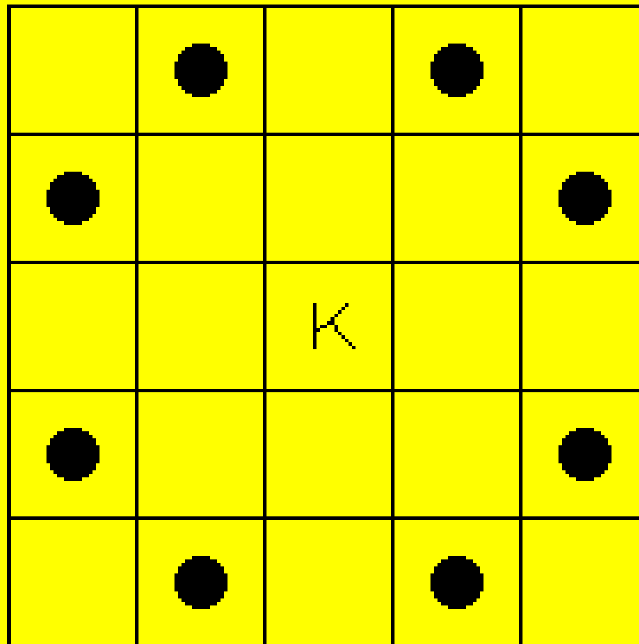
包装法

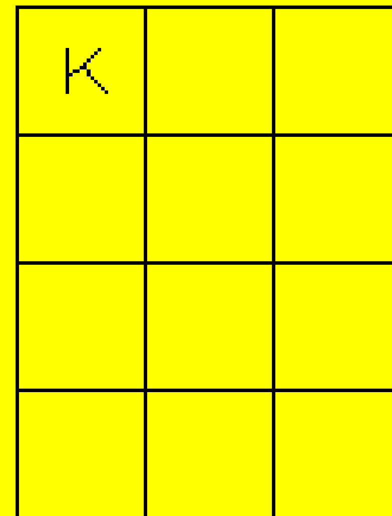
- 点集合の凸包を求める方法



騎士巡歴

図：ナイトの巡歴





●：ナイト（K）が動ける位置

問題

マイクロベンチ

- 要素数8000万個のリスト
- map，reduce，scanで軽い関数と重い関数を実行
- ブロック数1のリストへデータ再配置を実行

クラスタの台数	1	10	100	1000	2000	3000	4000
map(light)	0.0128	0.0129	0.0128	0.0128	0.0129	0.0130	0.0132
reduce(light)	0.0183	0.0182	0.0183	0.0191	0.0194	0.0197	0.0200
scan(light)	0.0407	0.0408	0.0411	0.0443	0.0484	0.0530	0.0580
map(heavy)	16.8	16.8	16.8	16.8	16.8	16.8	16.8
reduce(heavy)	16.9	16.9	16.9	16.9	16.9	16.9	17.0
scan(heavy)	33.8	33.8	33.8	33.9	33.9	34.0	34.1
redistribute	0	3.74	4.64	4.67	4.66	4.62	4.72

Pentium4 3.0Ghz、メモリ 1GByte、Linux 2.6.8

block-cyclic配置のオーバーヘッドは，データ再配置に比べると小さい

block-cyclic配置の性能評価

- 実験環境　Pentium4 3.0Ghz, メモリ 1GByte, Linux 2.6.8
- クラスタ16台
- 8000万個のデータ
- 軽い関数(かけ算)を，1000回ずつ繰り返す

ブロック数	map	reduce	scan	zip
1	13.22	18.19	41.01	21.91
10	13.24	18.33	41.07	22.15
100	13.26	18.39	41.47	23.88
1000	13.19	19.11	44.18	22.70

包装法

ソースコード（おおざっぱな流れ）

```
dist_list< point > points;                //初期集合
dist_list< point > boundary_points; //凸集合

point P0 = reduce(min_x, points); //x座標が一番小さい
boundary_points.push_back(P0);
繰り返し(n=1、2、3・・・){
    point Pn+1=reduce(Pn-1PnとPnPiの内積が最大となるPi、points);
    boundary_points.push_back(Pn+1);//凸集合に追加
    filter(Pn+1と等しくない点, points);//Pn+1を初期集合から除く
    if(P0  ==  Pn+1){
        break;//最初の点に戻ってきたら抜ける
    }
}
```

既存リストの問題点

- 現在のリストはサイズが固定されている

➔記述性、実行効率が悪くなる応用問題がある

（例）エラトステネスのふるい（既存リストを用いて解く）

計算機0					計算機1				
(2,T)	(3,T)	(4,T)	(5,T)	(6,T)	(7,T)	(8,T)	(9,T)	(10,T)	(11,T)

mapで2以外の2の倍数をF

(2,T)	(3,T)	(4,F)	(5,T)	(6,F)	(7,T)	(8,F)	(9,T)	(10,F)	(11,T)

mapで3以外の3の倍数をF

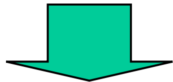
(2,T)	(3,T)	(4,F)	(5,T)	(6,F)	(7,T)	(8,F)	(9,F)	(10,F)	(11,T)

要素の有無を示すフラグがいる
➔記述性の低下

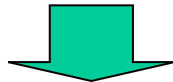
計算機間で計算量が偏る
➔実行効率の低下

データの再配置

必要なデータ送受信の
スケジュールを作成

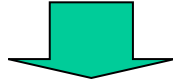


プロセス間の非同期
データ送受信命令を発行

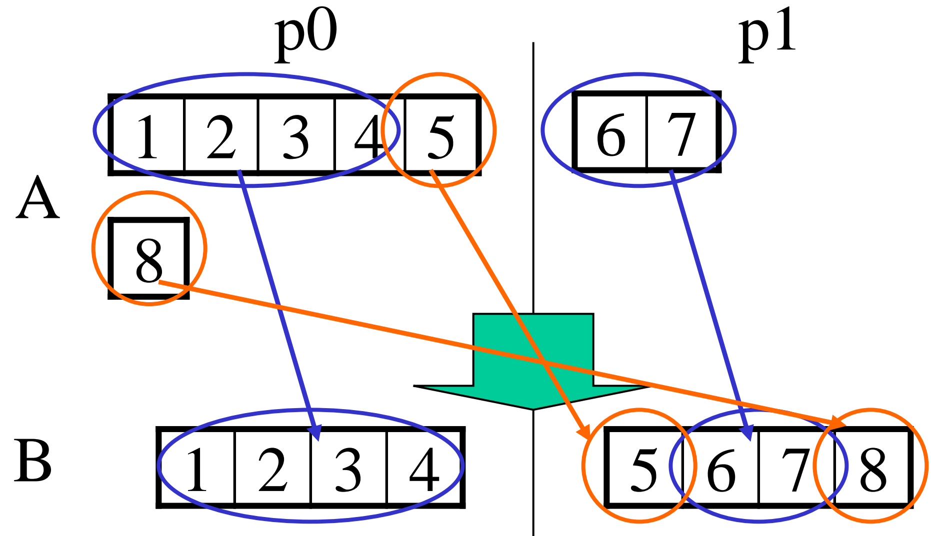


ローカル内での
データコピー

データ送受信



非同期通信が終わるのを待つ

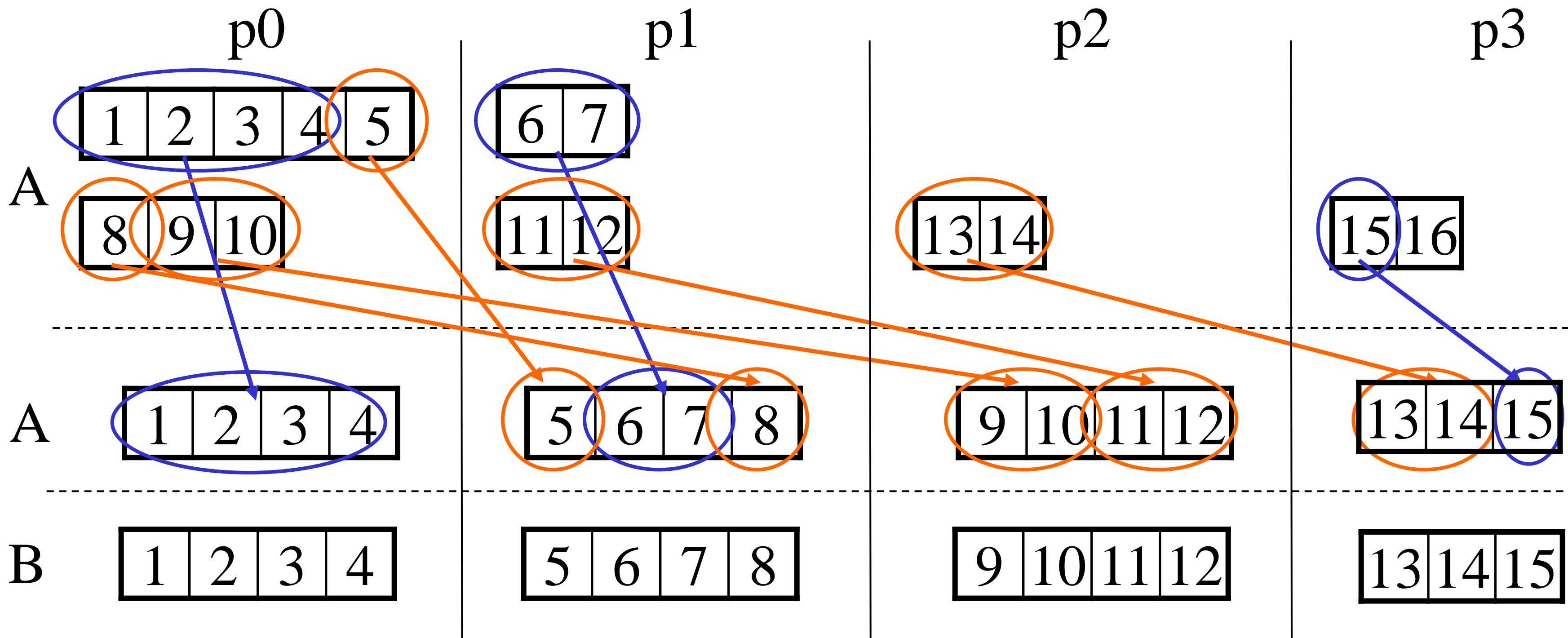


送信元	受信元	
p0	p0	(1,2,3,4)
p1	p1	(6,7)
p0	p1	(5)
p0	p1	(8)

新しいデータ構造とzipwith

- データ配置が異なる場合は、片方に合わせる

zipwith f A B => C

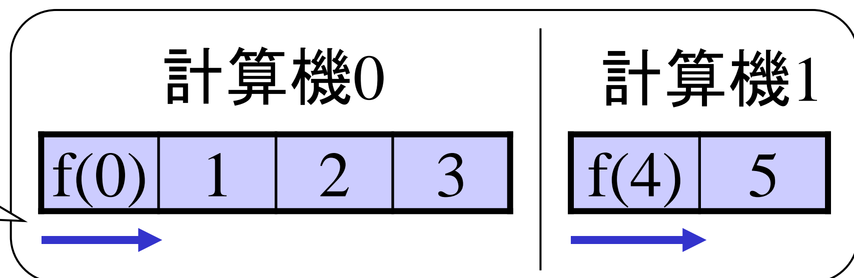


自動負荷分散の手法

- 実行時間の計測により、負荷分散が必要か予測する
  - 単位データあたりの通信時間はあらかじめ計測しておく

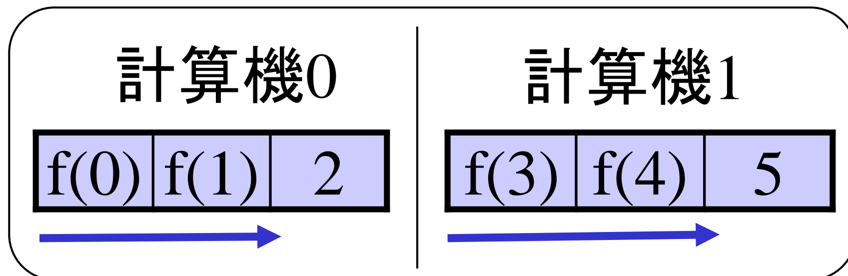
（例）map f [0,1,2,3,4,5]

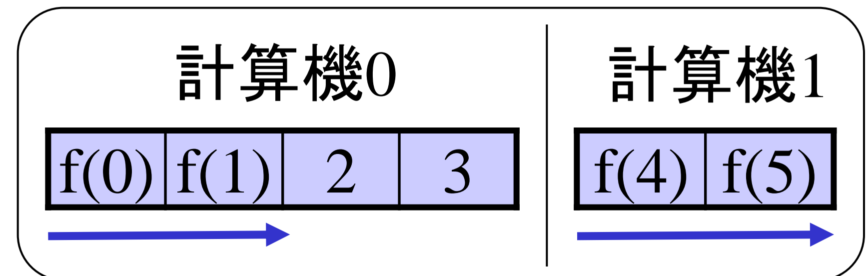
途中までmapを実行し、fの実行時間を計測する



負荷分散してから計算を続行（fの処理が重い場合）

そのまま続行





他のスケルトン適用時も同様