❖ ホモロジー検索の並列最適化 ❖

東京大学情報理工学系研究科
田浦研究室 M1 原健太朗

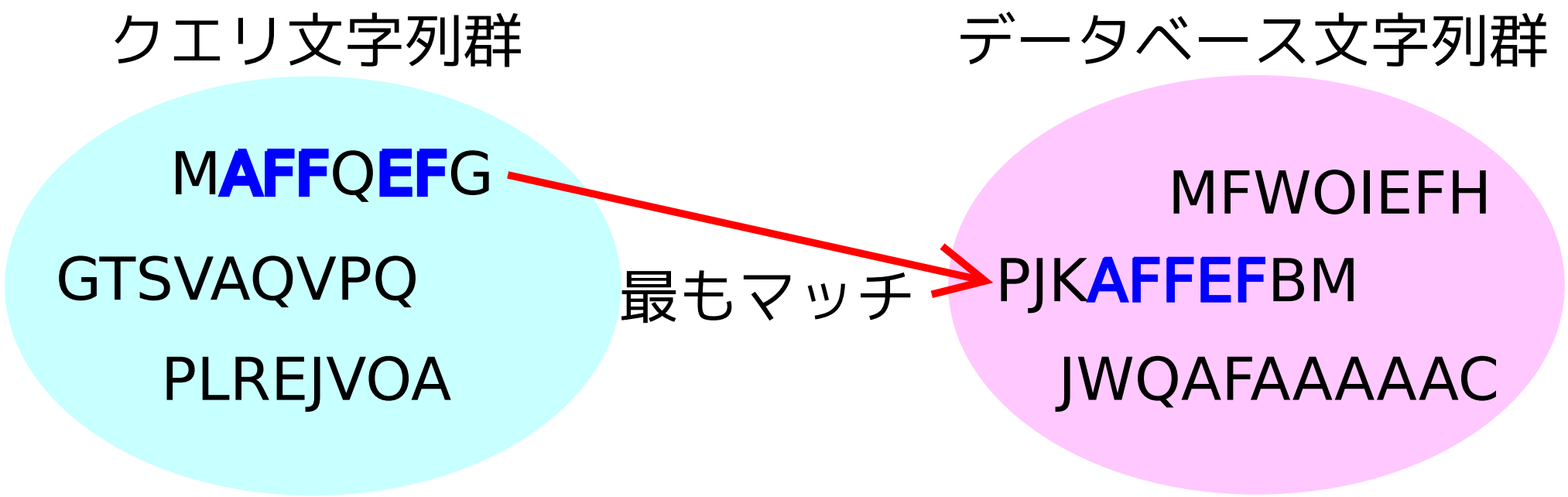
2009.5.29

目次

1. 課題

問題

- アミノ酸配列のホモロジー検索
  - → クエリ文字列とデータベース文字列がたくさん与えられる
  - → 各クエリ文字列に対して，最もマッチするデータベース文字列を探し，マッチ文字列を出力せよ



処理の流れ

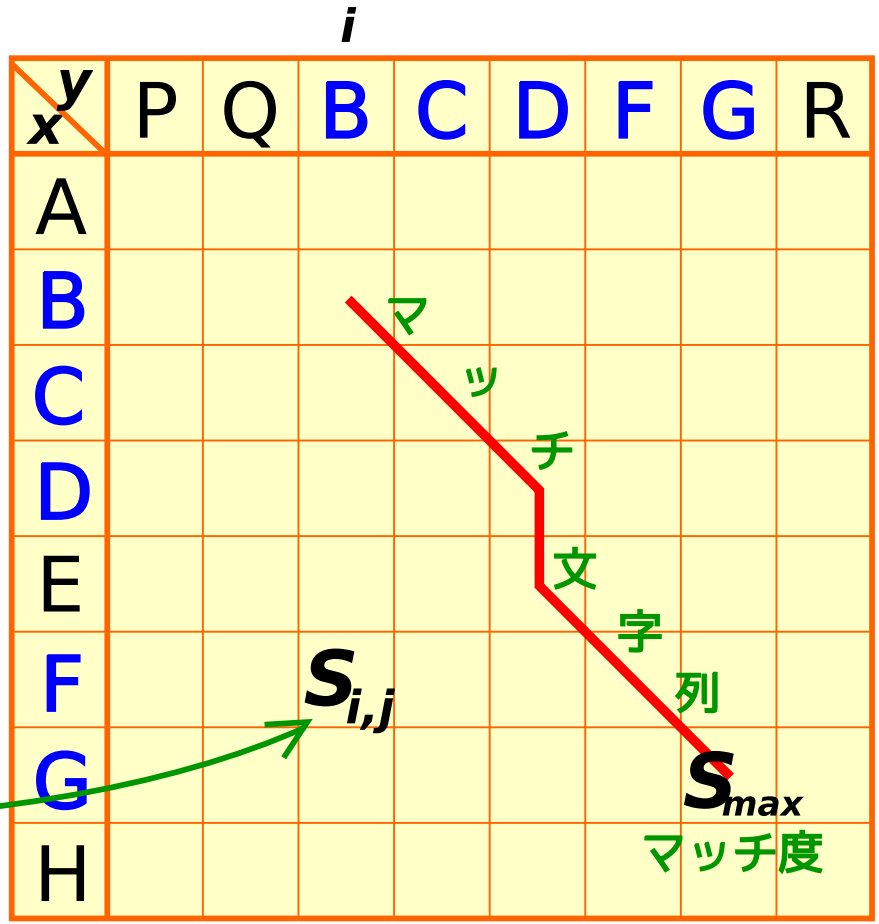
```
for q in クエリ {
    for d in データベース {
        qとdのマッチ度を計算;
    }
    最大マッチ度を持つd(たち)とqとのマッチ文字列を計算;
}
```

- **最大マッチ度を持つ d は複数存在する可能性あり**

マッチ度の計算

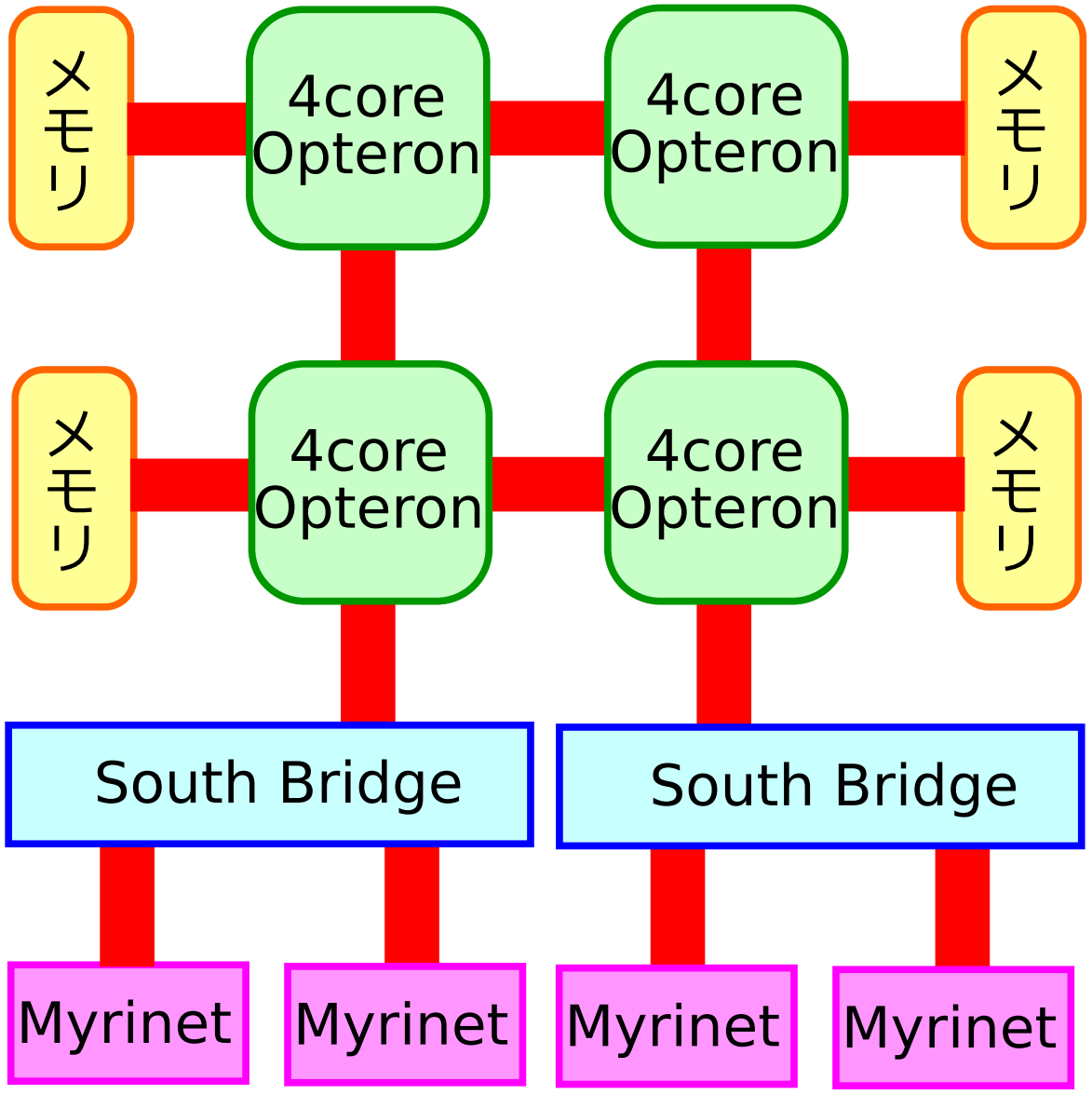
- Smith-Waterman アルゴリズム
  - → O(クエリ文字列長 × データベース文字列長) の動的計画法
  - →「文字列の編集距離計算の複雑バージョン」

$$L_{i,j} = \max(S_{i-1,j} - d,\ L_{i-1,j} - e)$$
$$U_{i,j} = \max(S_{i,j-1} - d,\ U_{i,j-1} - e)$$
$$S_{i,j} = \max(S_{i-1,j-1} + M_{x_i,y_j},\ L_{i,j},\ U_{i,j})$$
M：スコア行列，　d, e：定数



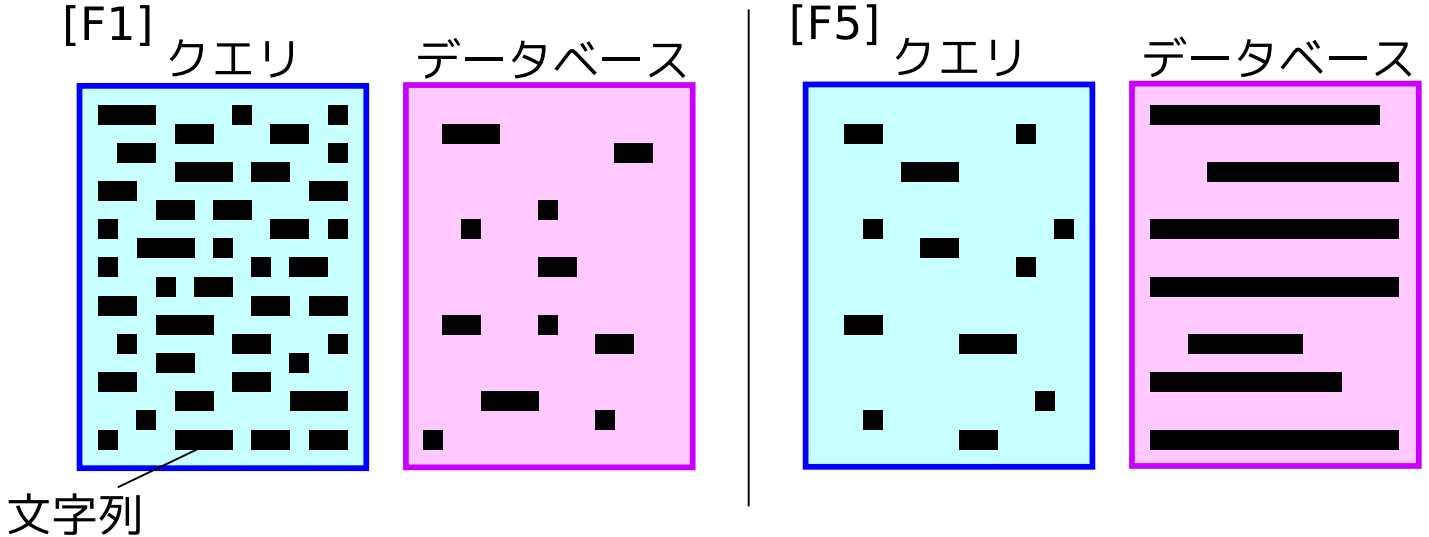
計算環境

- T2K 東大の 32 ノード 512 プロセッサ



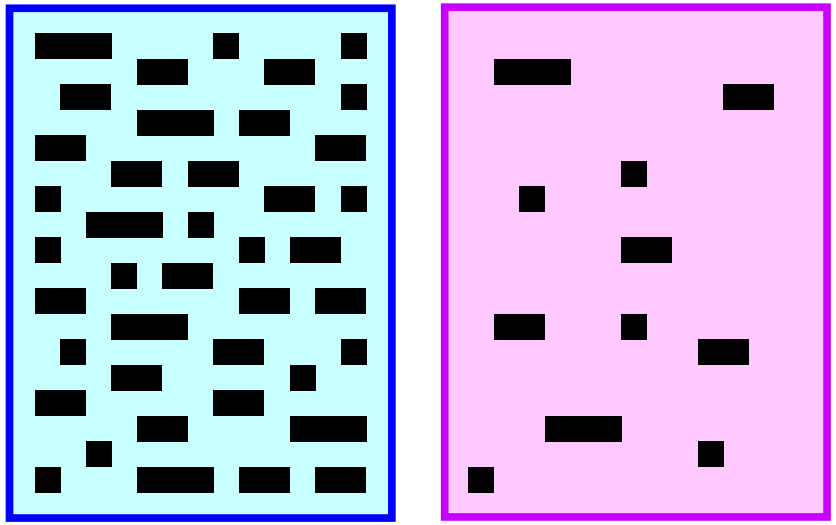
データセット

データセット	クエリ数	データベース数	クエリ文字列長	データベース文字列長
F1	50000	500	61〜3824	61〜3333
F2	5000	5000	61〜3565	61〜3816
F3	500	50000	61〜3512	61〜3838
F4	500	500	61〜3593	15373〜61407
F5	100	100	102〜740	364853〜1374478



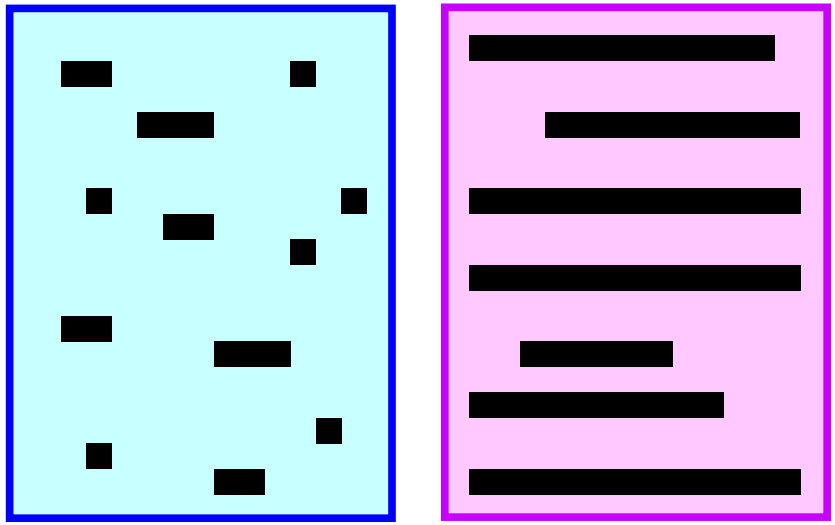
並列化への方針

[F1]



クエリ数：50000
プロセッサ数：512

[F5]

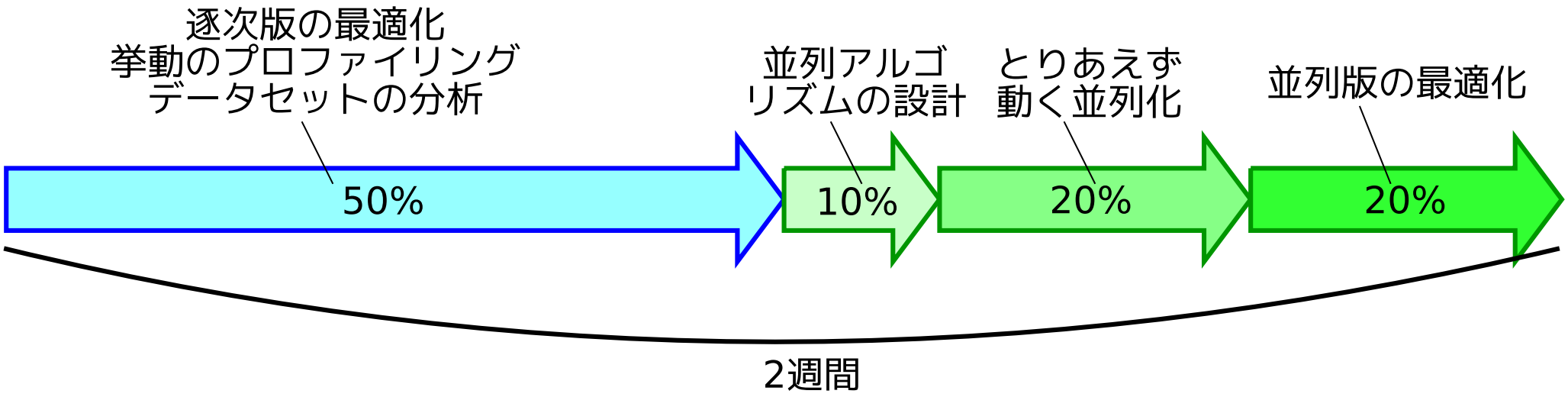


クエリ数：100
プロセッサ数：512

- F1 **は単純にクエリを分散させれば効率的に並列化可能**
- F5 **は単純にクエリを分散させるだけでは負荷バランスが大きく崩れる**

→ **いかに効率良く負荷バランスさせるか**が焦点

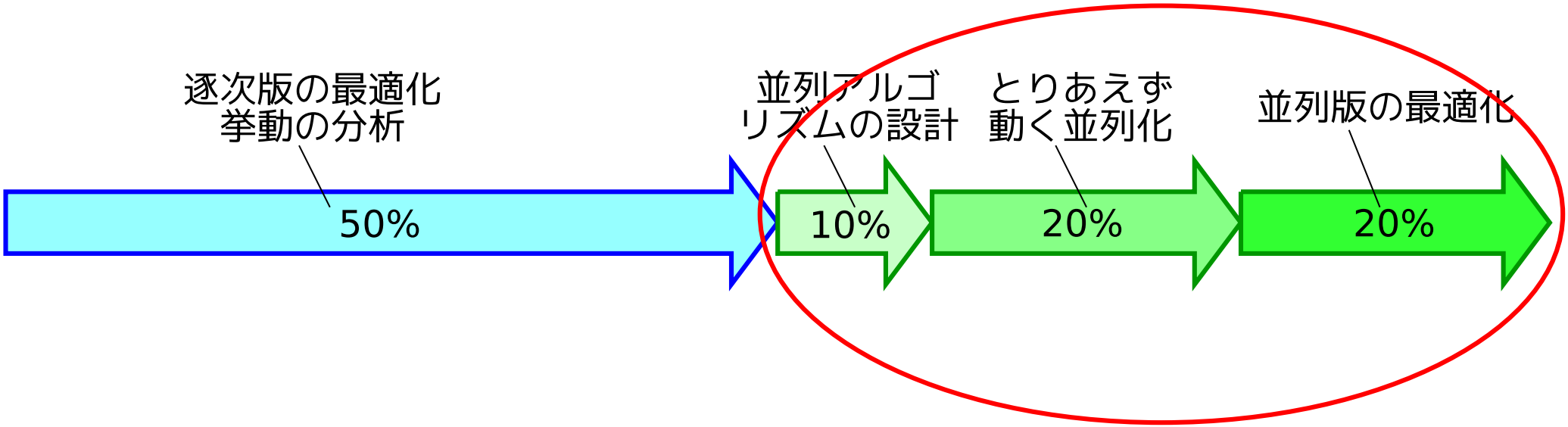
開発工程



- **逐次プログラムの最適化と分析に力を入れた**
- **C 言語・MPI で 1700 行**

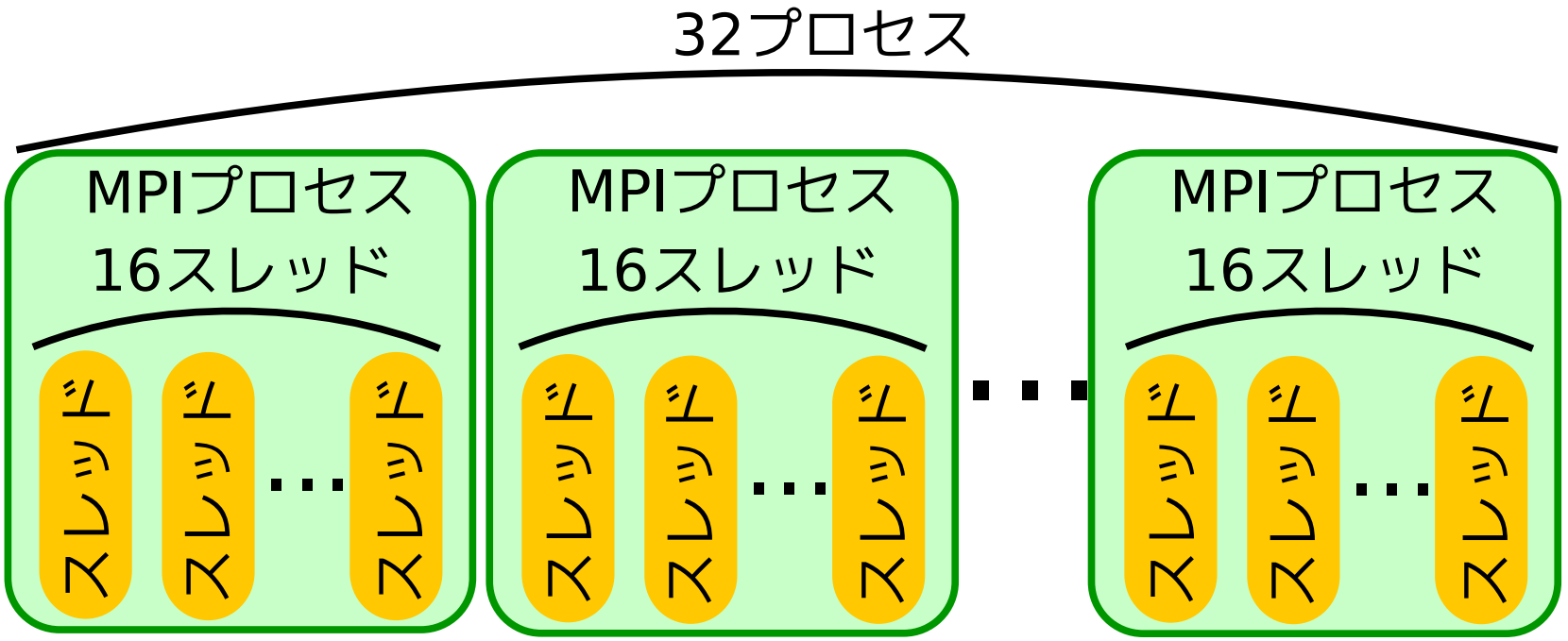
2. 並列アルゴリズム

開発工程 (再掲)



プロセス構成

- 各ノードには 1 個の MPI プロセスと 16 スレッド

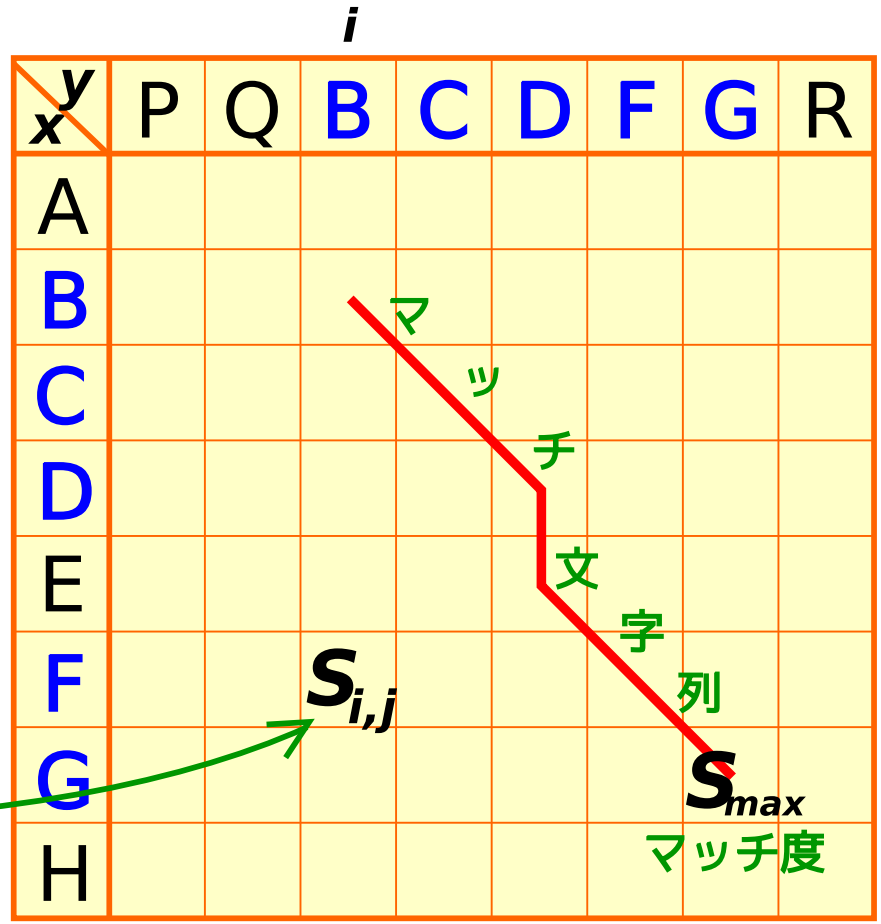


※16コアマシン×32ノード

負荷分散時のポイント

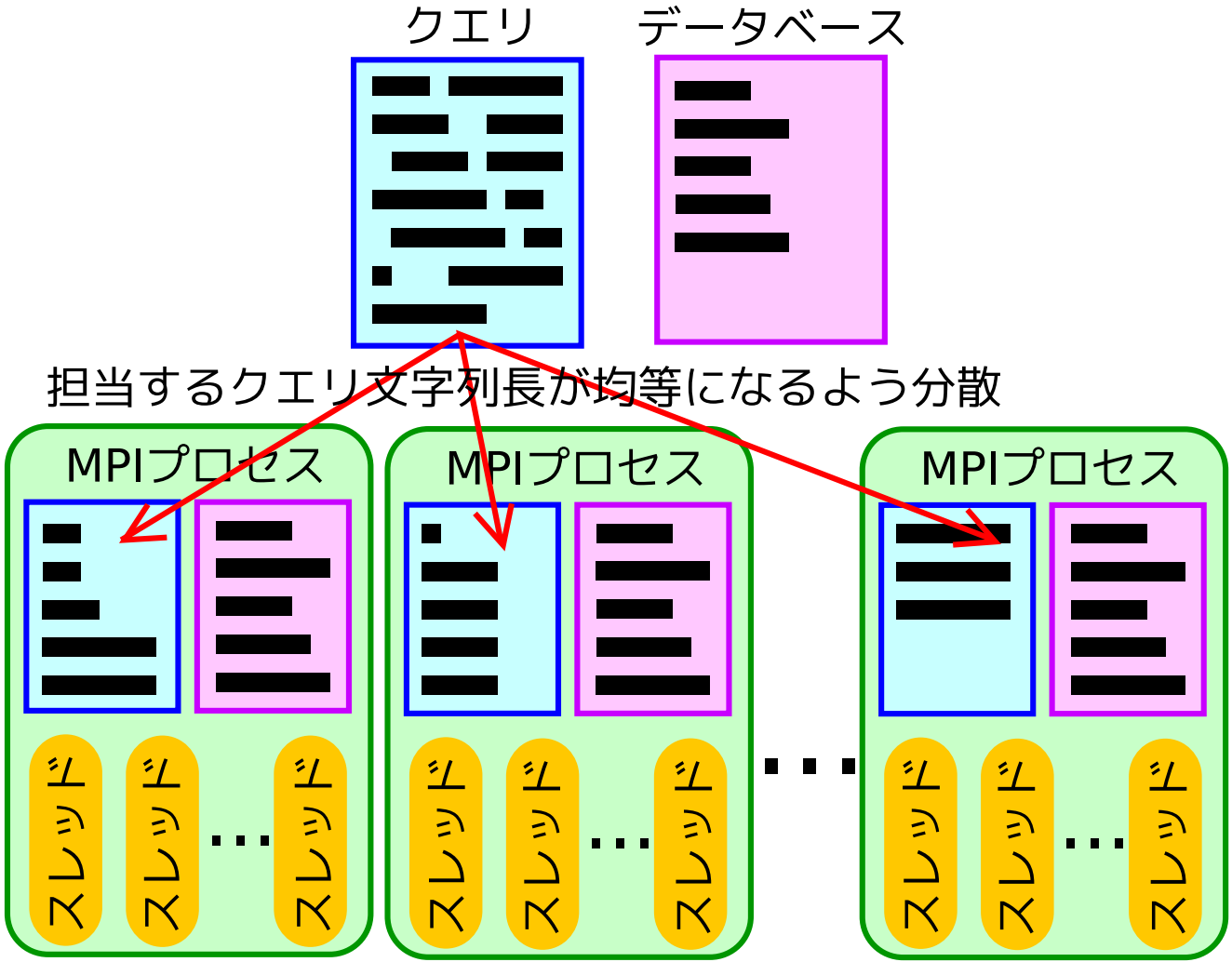
- マッチ度の計算時間は (クエリ文字列長)×(データベース文字列長) にほぼ完全に比例
  - → 文字列長によって静的に負荷が予測可能

$$L_{i,j} = \max(S_{i-1,j} - d,\ L_{i-1,j} - e)$$
$$U_{i,j} = \max(S_{i,j-1} - d,\ U_{i,j-1} - e)$$
$$S_{i,j} = \max(S_{i-1,j-1} + M_{x_i,y_j},\ L_{i,j},\ U_{i,j})$$
M：スコア行列，　d, e：定数



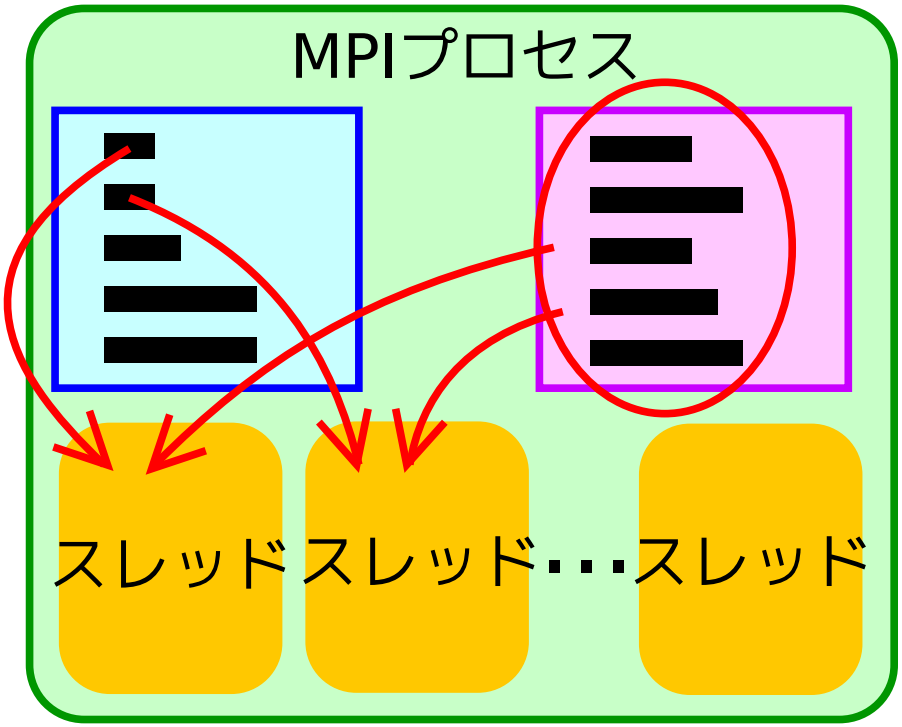
Step1：クエリ文字列の静的負荷分散

- 各ノードが担当するクエリ文字列長合計が等しくなるようにクエリを分散
  - → この処理に通信は不要



Step2：粗粒度な動的負荷分散

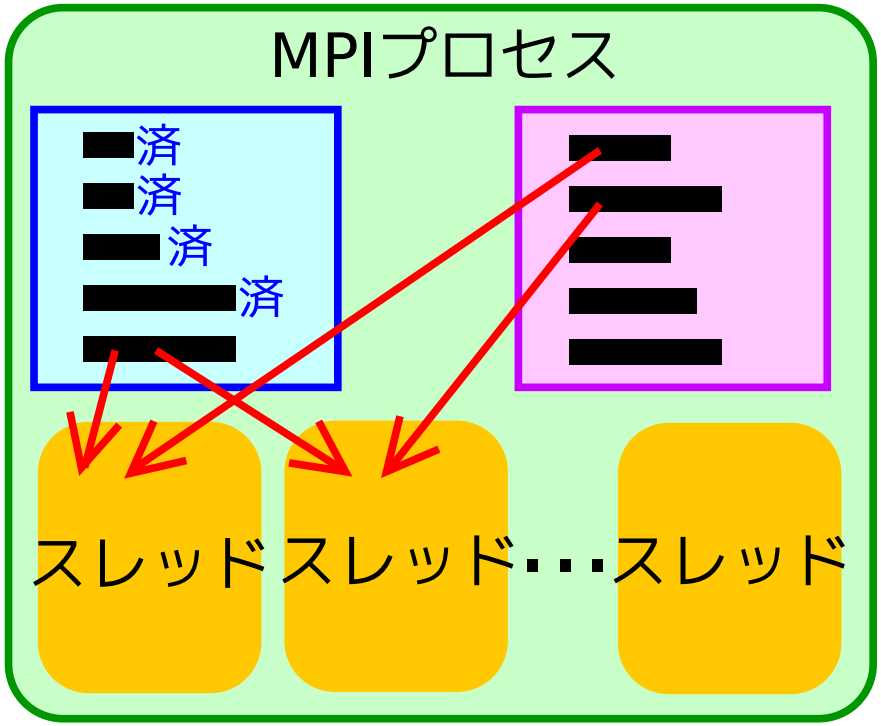
- 各スレッドはクエリ文字列に関してマスタワーカ方式で処理



(1) クエリ文字列を1個取ってくる
(2) データベース文字列全部とマッチ度計算してマッチ文字列を求める

Step3：細粒度な動的負荷分散 (1)

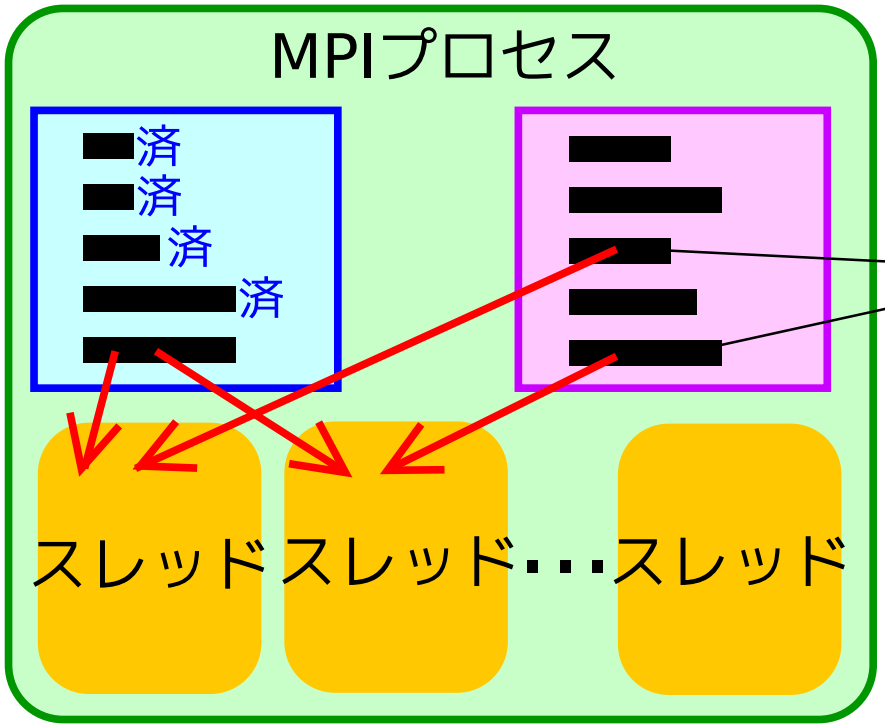
- **各スレッドは (クエリ文字列，データベース文字列) の組合せに関してマスタワーカ方式で処理**



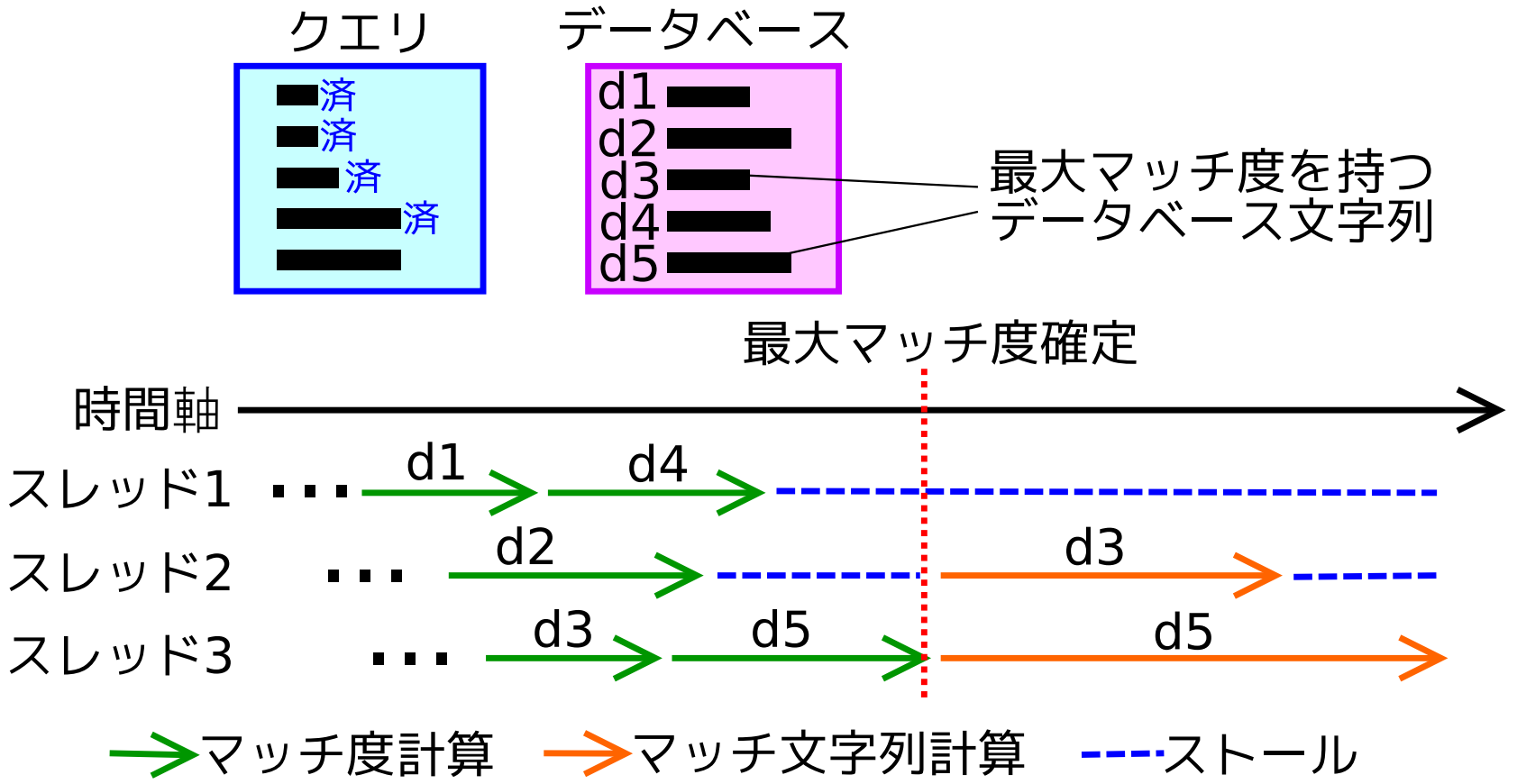
(1) クエリ文字列1個とデータベース文字列1個を取ってきてマッチ度計算

Step3：細粒度な動的負荷分散 (2)

- マッチ文字列の計算もマスタワーカ方式で処理



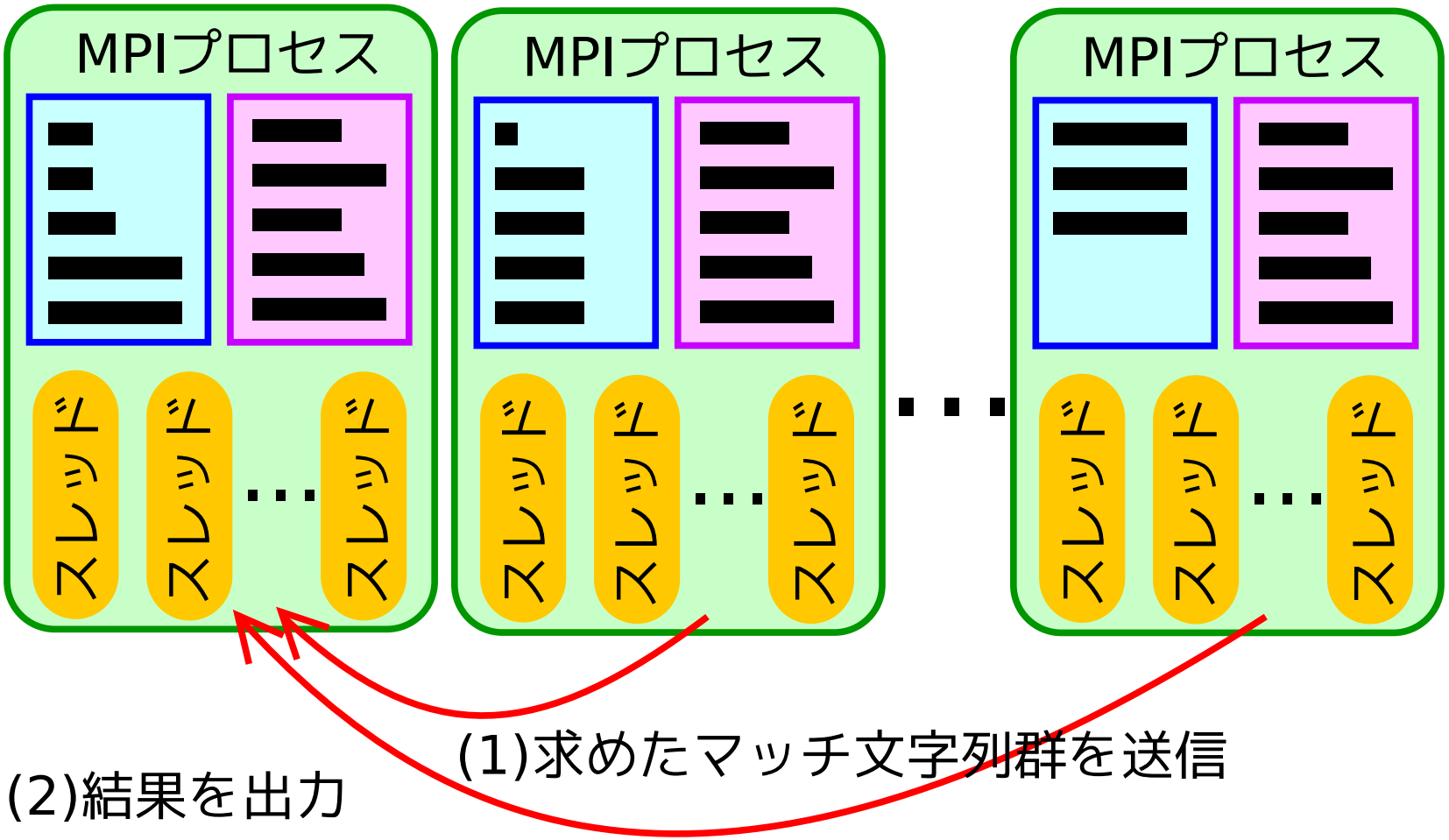
Step3：細粒度な動的負荷分散 (3)



- **クリティカルパスの短縮には「一番最後の」マッチ文字列計算の負荷分散が特に重要**
  - → **マッチ文字列計算はマッチ度計算より約 2 倍重い**
  - → **データセット F5 では 1 つのマッチ文字列計算に最大 14 秒**

Step4：マッチ文字列の収集

- **計算が終了次第，結果を 1 個のプロセスに送信**



アルゴリズムの要約

ノード間で，文字列長に基づいてクエリを静的に分散

↓

ノード内で，粗粒度のマスタワーカ方式で動的負荷分散

↓

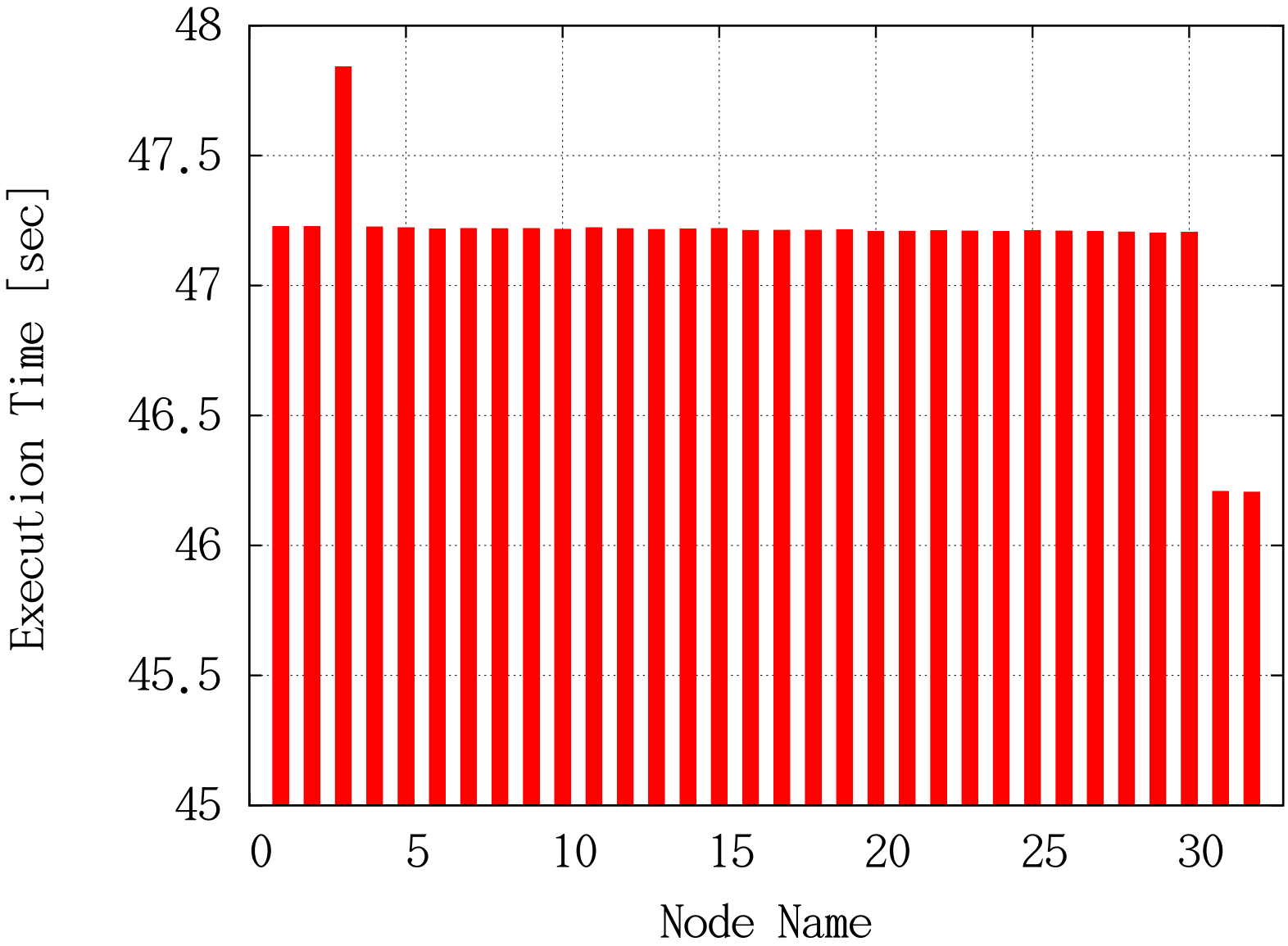
ノード内で，細粒度のマスタワーカ方式で動的負荷分散

↓

マッチ文字列を収集

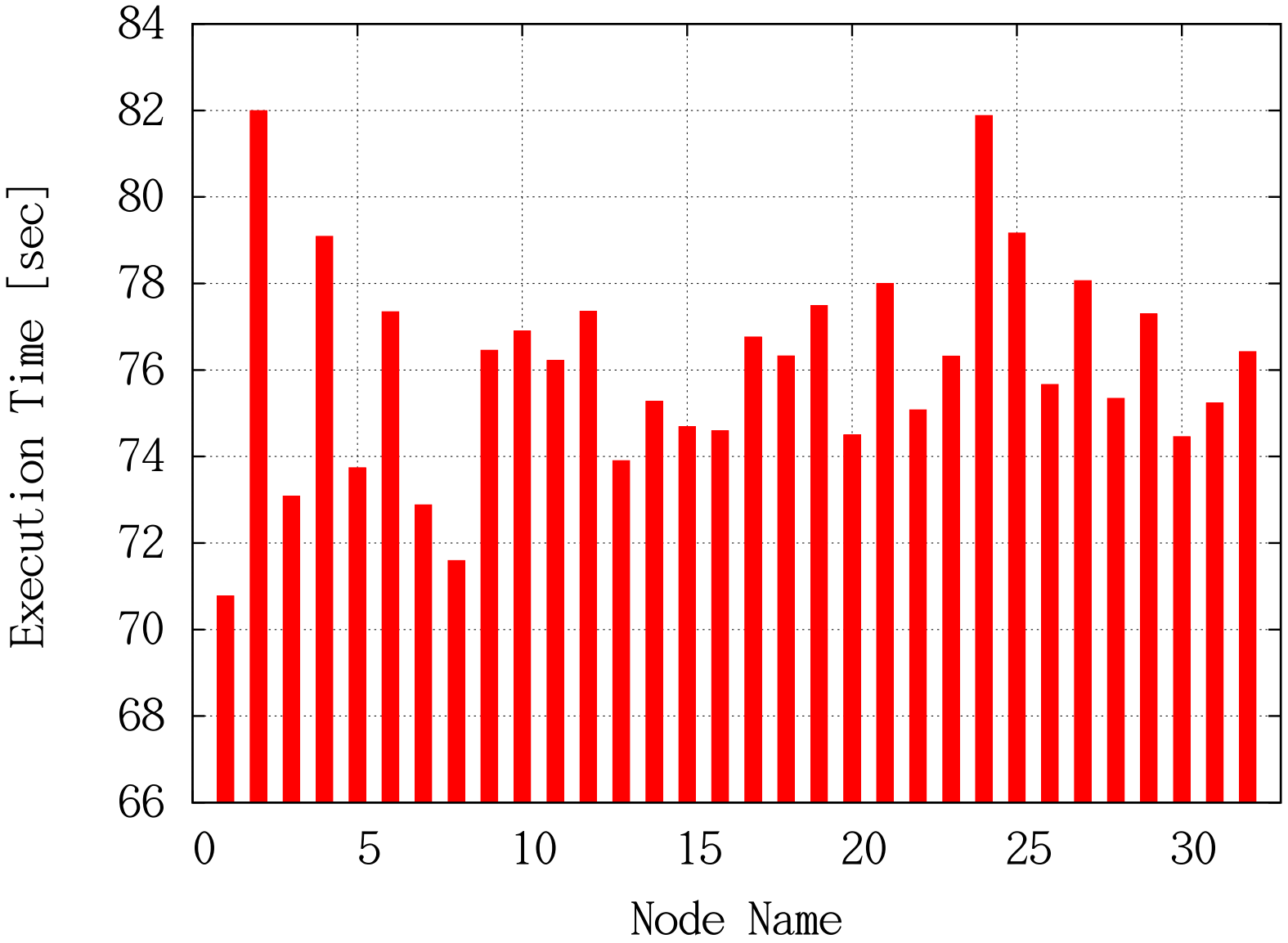
- ノード間では，クエリ文字列を**静的に負荷分散**
- 各ノード内では，マスタワーカ方式で**効率良く動的に負荷分散**
- **通信は最後にわずかに起きるだけ**

ノード間の負荷分散状況 (データセット F1)



➤ バランスが取れている

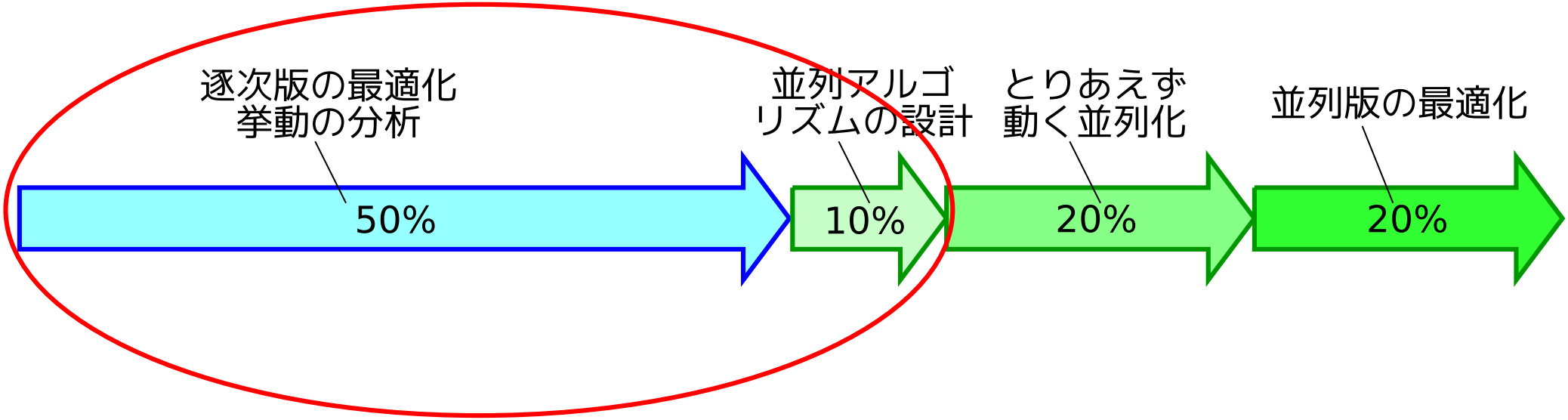
ノード間の負荷分散状況 (データセット F5)



- 多少乱れる

3. 最適化へのアプローチ

開発工程（再掲）



適用した最適化手法

Method0 : 基本的な最適化

→ コーディングレベルの最適化

→ キャッシュヒット率の意識

Method1 : 4KB エイリアシングの抑止

Method2 : コンパイラの選択

Method3 : エンコーディングによる通信量削減

Method1：4KB エイリアシングの抑止 (1)

- a と b のアドレス差が 4KB の整数倍だと著しく遅い

```
[Case 1]
int *a;  /* 0xFFFF1000 */
int *b;  /* 0xFFFF2000 */
for (i = 0; i < size; i++) {
    *a = 0;
    *b;
}
```

遅い

```
[Case 2]
int *a;  /* 0xFFFF1000 */
int *b;  /* 0xFFFF2016 */
for (i = 0; i < size; i++) {
    *a = 0;
    *b;
}
```

速い

単位は [sec]	Opteron	Athlon	Core2Duo	Xeon
Case 1	1.48	2.66	4.91	5.18
Case 2	0.88	2.00	0.85	0.86

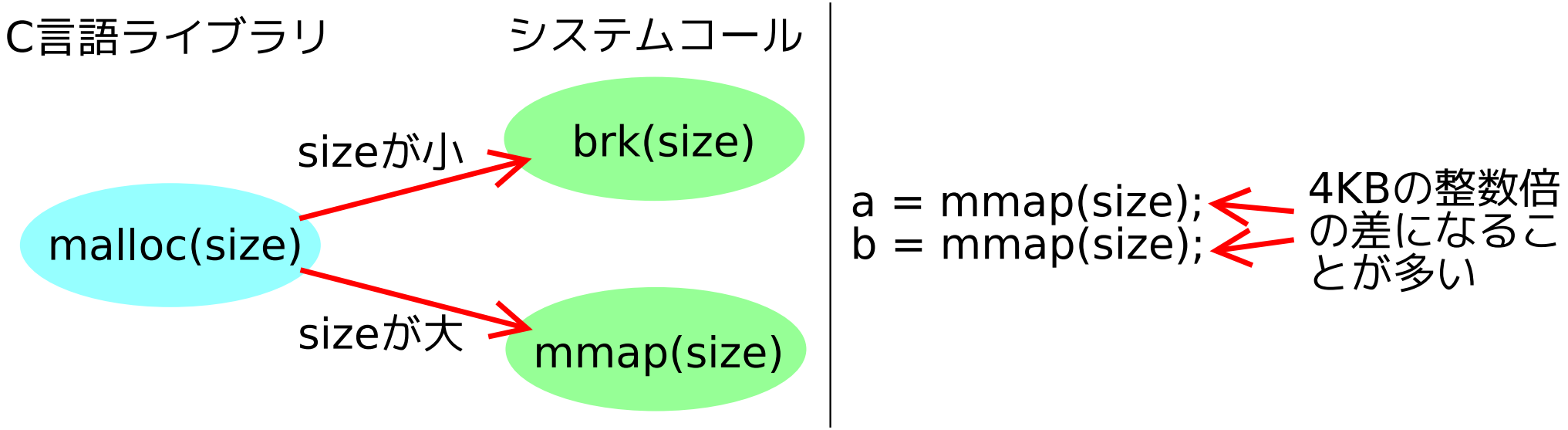
Method1：4KB エイリアシングの抑止 (2)

```
[Case 1]
int *a;  /* 0xFFFF1000 */
int *b;  /* 0xFFFF2000 */
for (i = 0; i < size; i++) {
      *a = 0;
      *b;
}
```

「同じ」アドレスへの
ストア→ロードと見なされる

- Out of Order 実行では，ロード対象のアドレスが，先行するストア対象のアドレスと「一致」している場合，ストアが完了するまでロードがストールする
  - → 「一致」かどうかは下位 12bit の比較で判断 (intel 系)
  - → 4KB の整数倍だけ離れた 2 つのアドレス間に偽依存関係が生じる

Method1：4KB エイリアシングの抑止 (3)



- malloc は内部で brk または mmap を呼ぶが...
  - → ページサイズの関係上，連続した mmap 呼出は 4KB の整数倍だけずれたアドレスを返すことが多い
  - → malloc 任せにすると 4KB エイリアシングの危険性あり
- ホモロジー検索では実行時間が約 2 倍も変わる

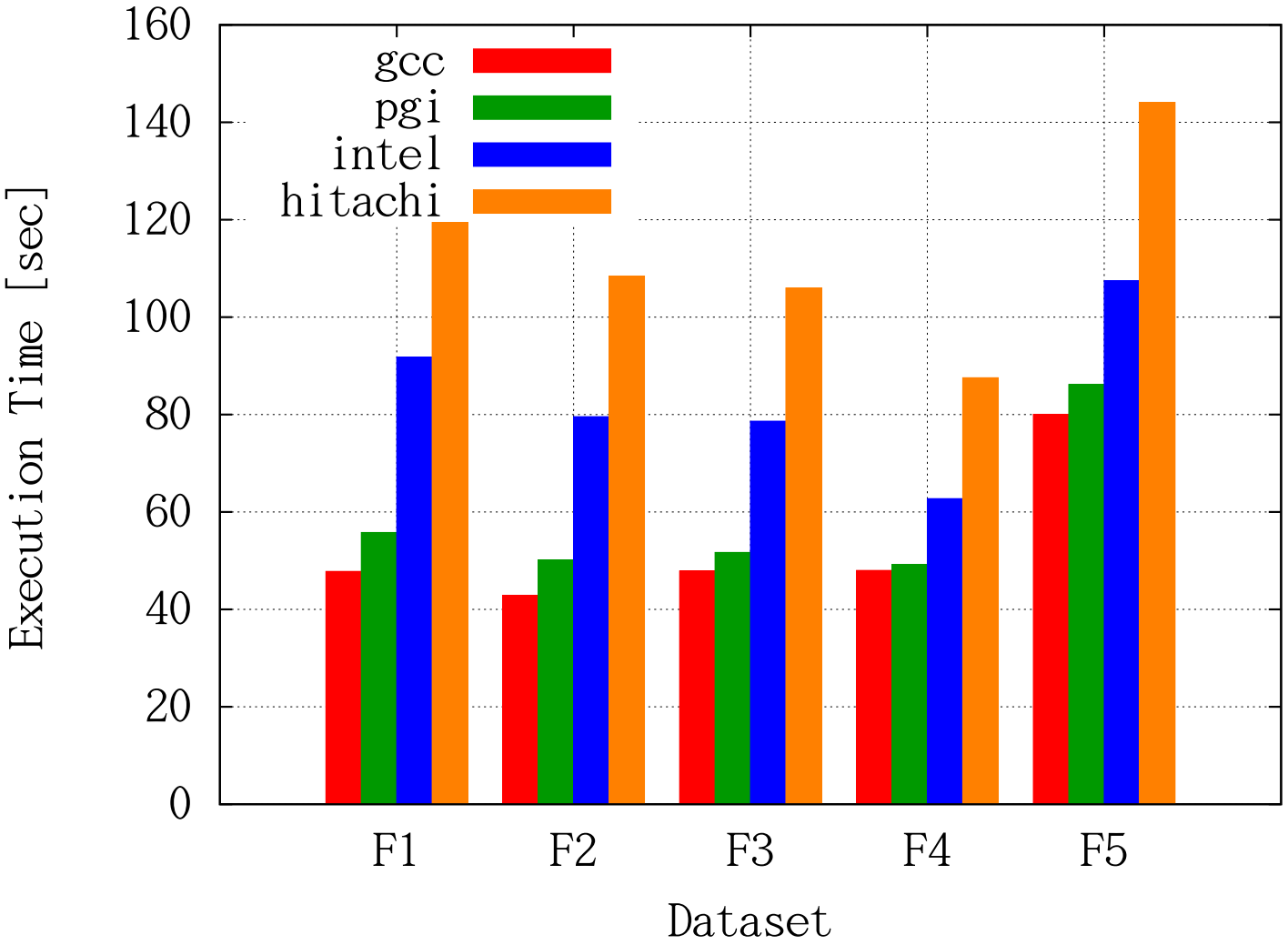
一般論：データ配置のポイント

- データ配列の先頭メモリアドレスは**明示的に適切に管理**する
- 留意点：
  - → **キャッシュヒット率を意識したデータ配置**が最重要
    - ♦ L1,L2,L3 キャッシュラインサイズの整数倍へのアライン
    - ♦ フォルスシェアリングの抑止
    - ♦ キャッシュワーキングセットへのコンフリクトの抑止
  - → 4KB **エイリアシング**も意識する

Method2：コンパイラの選択 (1)

- 4 つのコンパイラを比較
  - → gcc：-O4
  - → pgi：-fast -O3 -tp=barcelona-64
  - → intel：-O3 -ipo -no-prec-div
  - → hitachi：-Os +Op -noparallel -noischedule
- MPI には mpich 1.2.7 を使用
- プロファイルベースの最適化は行っていない

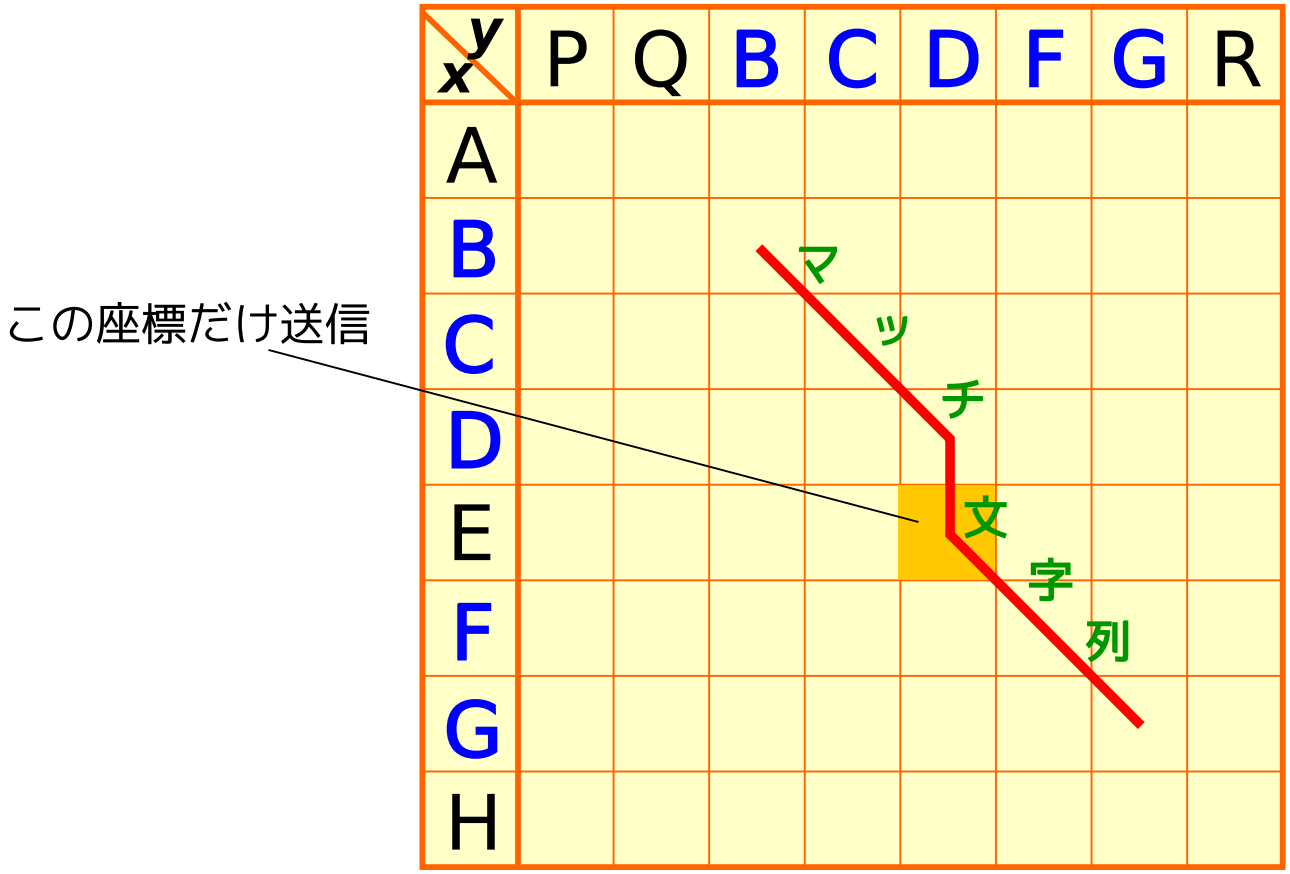
Method2：コンパイラの選択 (2)



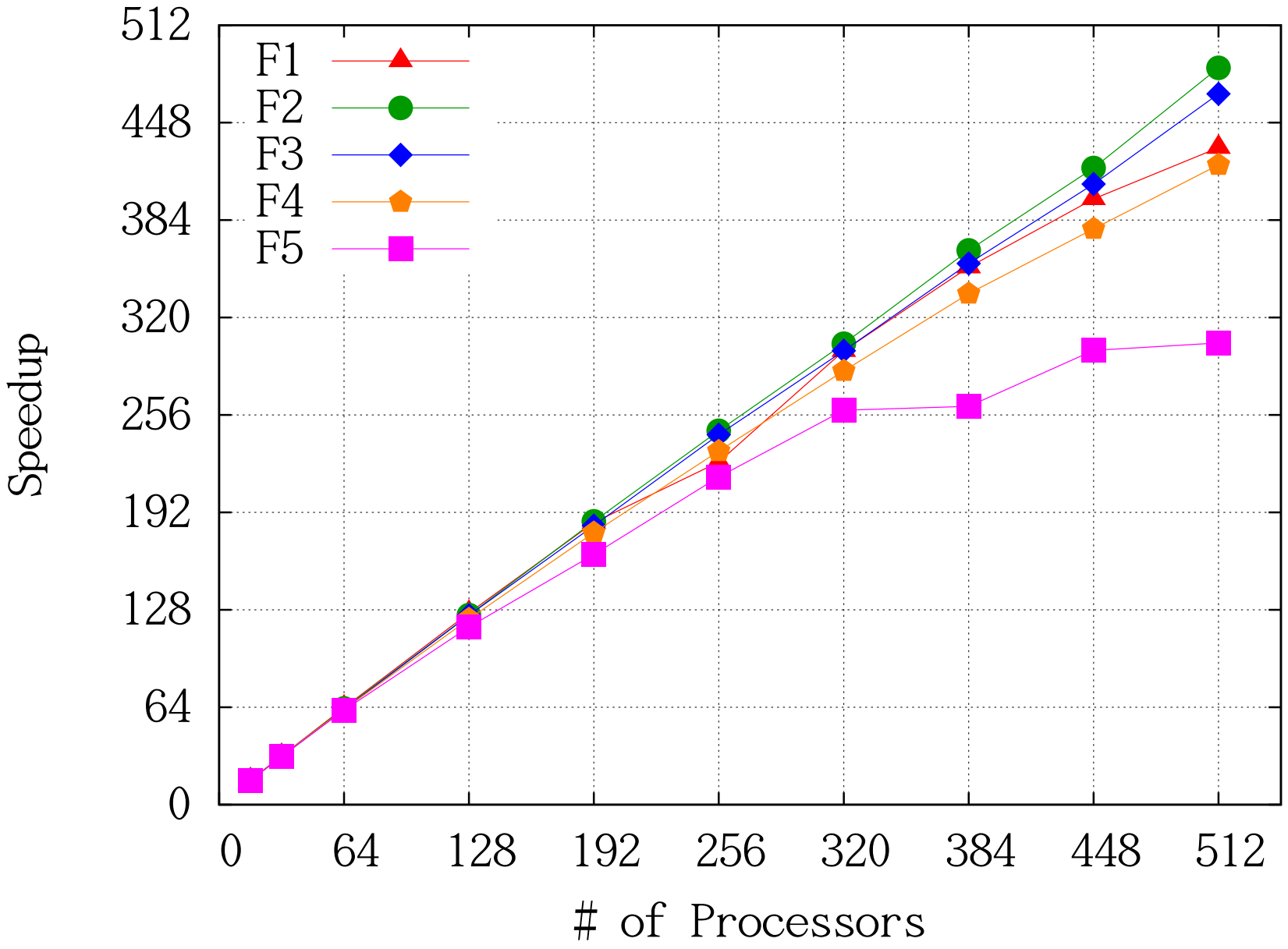
- コンパイラ間の実行時間差の原因は特定できず
- gcc を使用

Method3：エンコーディングによる通信量削減

- マッチ文字列はほぼ対角線になる
  - → 対角線にならない部分の座標のみ送信
- 解答文字列をフルに送る場合より，送信データ量を**平均 95% 削減**

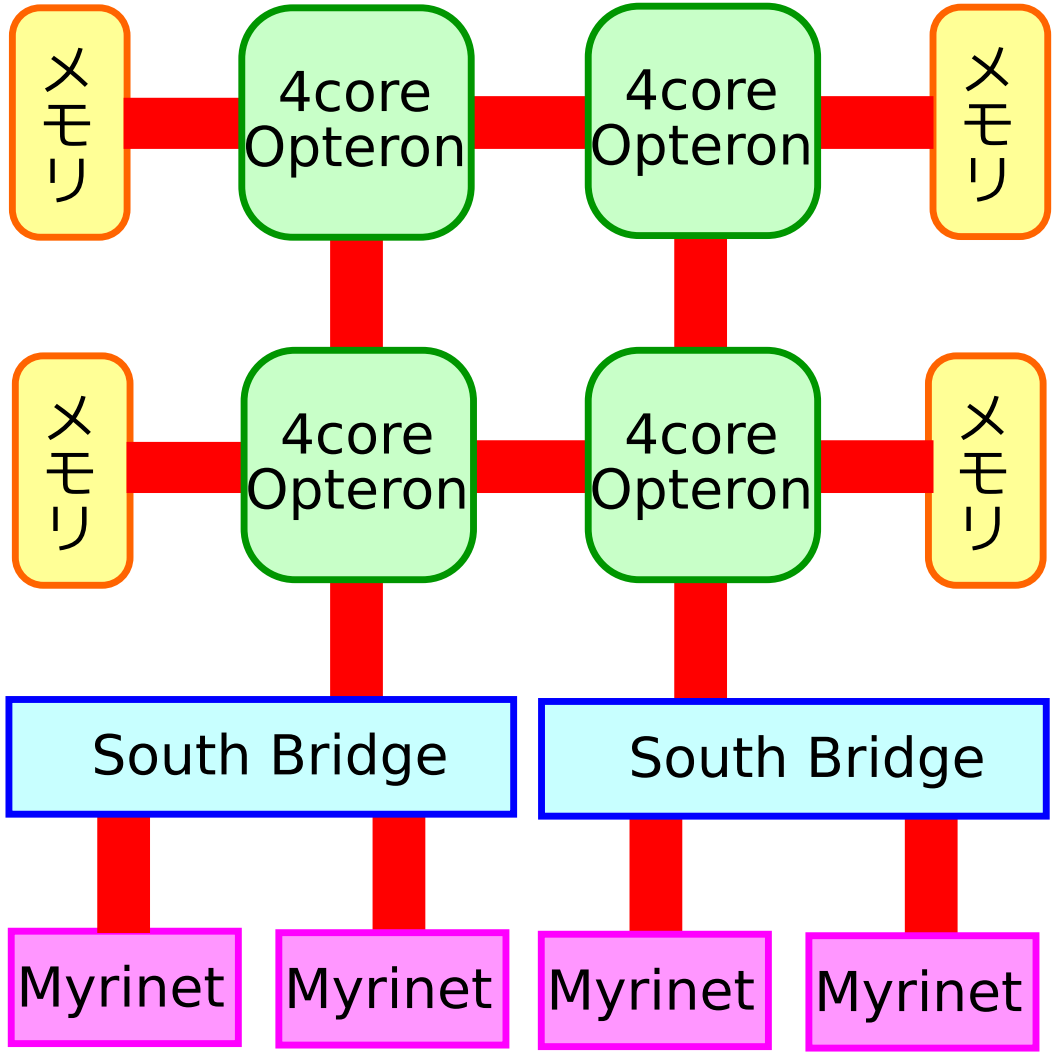


最終結果：スケーラビリティ



未検討事項

- 今回のアルゴリズムは SMP 構成を仮定
  - → NUMA を意識すればマッチ度計算がさらに高速化できた?

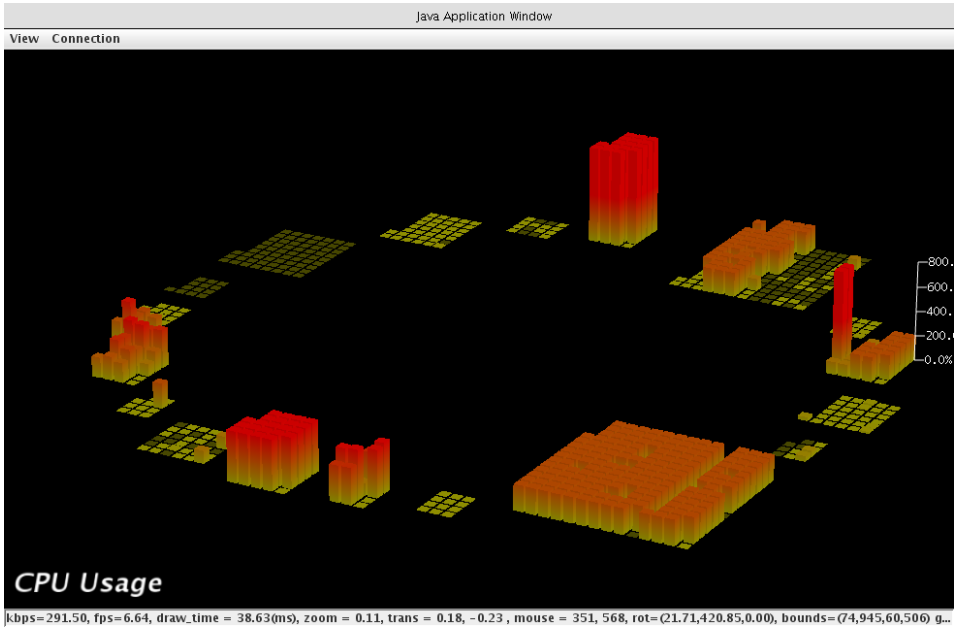


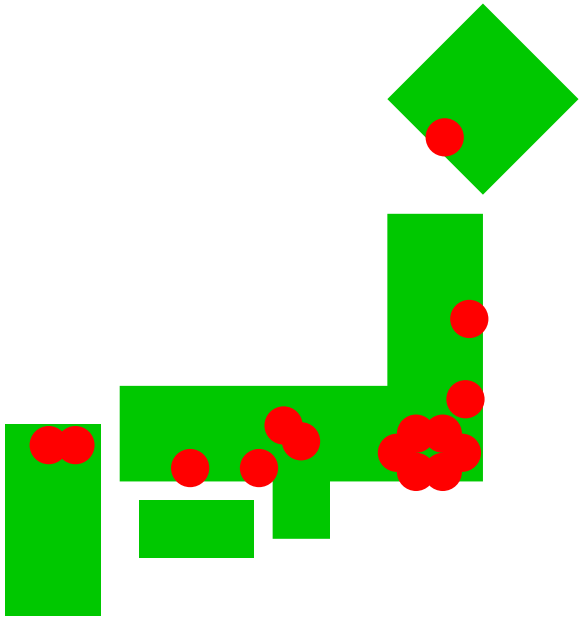
4. スパコンの感想

- スパコン vs 汎用クラスタ

汎用クラスタ (InTrigger)

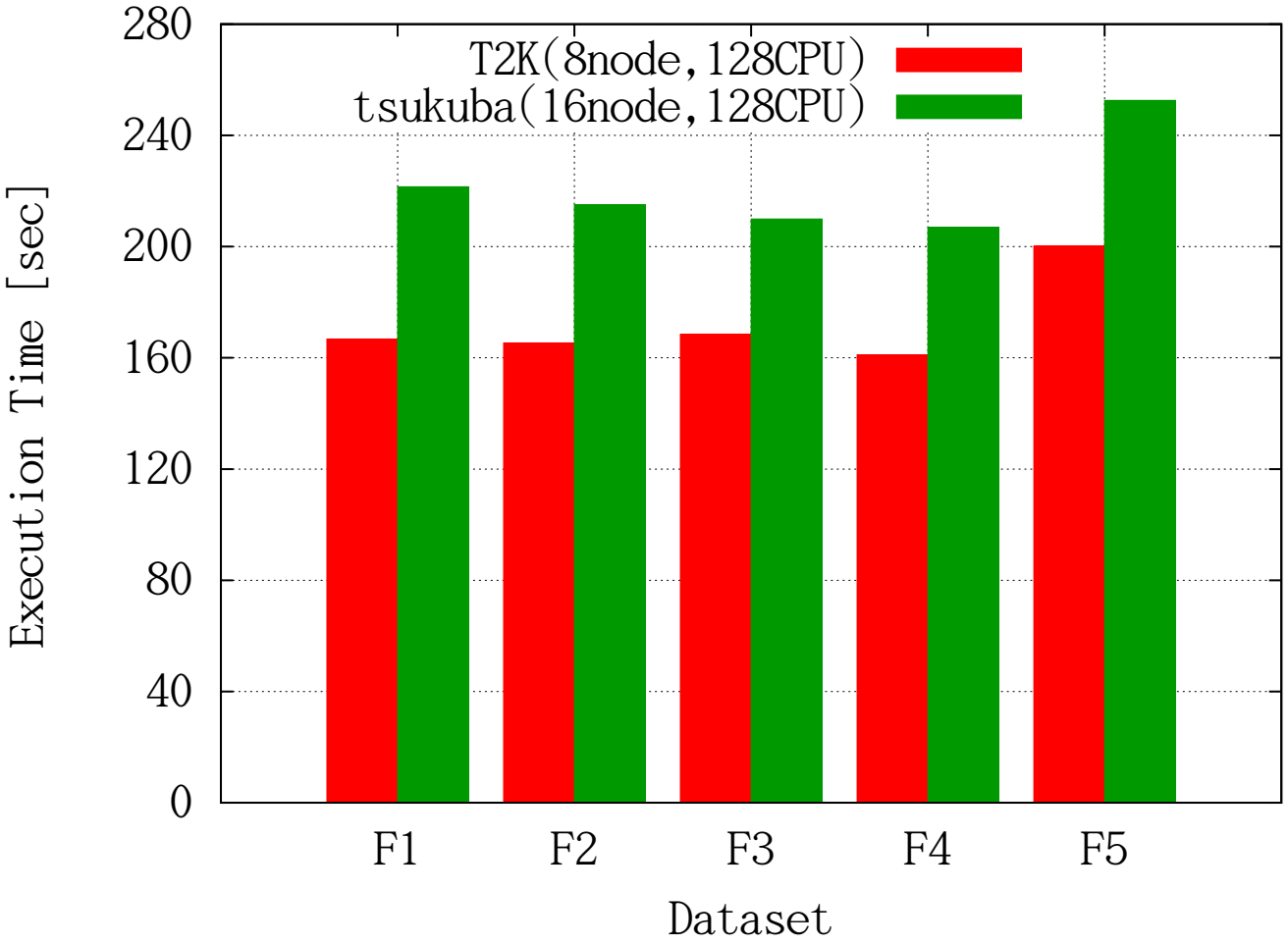
- InTrigger
  - → 全国 15 拠点，390 ノード，1436CPU
  - → 汎用サーバによる広域分散クラスタ
  - → 全ノードに直接 ssh ログインしてインタラクティブにコマンドを実行可能
- 今回は tsukuba クラスタを使用





T2K 東大 > InTrigger

- T2K 東大：Opteron 2.3GHz，16 コア ×8 ノード
- tsukuba：Xeon 2.33GHz，8 コア ×16 ノード



- T2K 東大は tsukuba より平均 22% 速い

T2K 東大 < InTrigger

- T2K 東大 (などのジョブ型システム) は対話性に乏しい
  - → 開発・デバッグは InTrigger で，チューニングは T2K 東大で行った
  - → T2K 東大にも，(デバッグキュー以外にも) 何らか対話的に実行可能なノードがあるとうれしい?

謝辞

- 実行委員会の皆様，ありがとうございました！