並列組合せ最適化システム jPoPの分枝限定法の実装

中川 伸吾[*1], **飯野 彰子**[*1],
中田 秀基[*2*1], **松岡 聡**[*1*3]

[*1]**東京工業大学**
[*2]**産業技術総合研究所**
[*3]**国立情報学研究所**

本研究の背景

- 組合せ最適化問題
  - 多次元パラメータ関数の最適値を求める
  - スケジューリング、設計問題、生産計画など、実社会において応用範囲が広い
  - 実用上の問題では膨大な計算量が必要
  - 自明な並列性が大きく、実行粒度の調整が容易
    - グリッドとの親和性

グリッド上で組合せ最適化問題を解く研究が多く行われている

既存の研究

- Ninfシステムを用いた実装
  - [**夏目ら**,02],[A.Takeda et al,02],etc
    →最適化問題のアルゴリズムを一から実装する
- **並列化ライブラリ**
  - **分枝限定法**:
    PUBB[Y.Shiono,96],ZRAM[A.Brungger,99],etc
  - **その他**:Popkern[**横山ら**,01]
    →ポータビリティが低い(実装言語の制約)
    →計算機のヘテロ性に対応してない

jPoPの機能

- グリッド環境における大規模並列組合せ最適化支援環境
- ユーザは問題依存領域のみを記述することでグリッド上で最適化アプリケーションを実装、実行可能
  - 並列･分散実行をユーザから隠蔽
- 標準的なグリッド技術を使用
  - グリッド上の計算資源の選択
  - 高いセキュリティを持った通信、計算資源の保護
- ポータビリティの確保(環境に依存しない汎用性)

本研究の目的

- jPoPの分枝限定法用クラス群jPoP-BBの設計、実装、性能評価
  - →0-1ナップサック問題を用いた評価
    - 遺伝的アルゴリズム用クラス群jPoP-GAはすでに実装されている[秋山ら,03]

発表の流れ

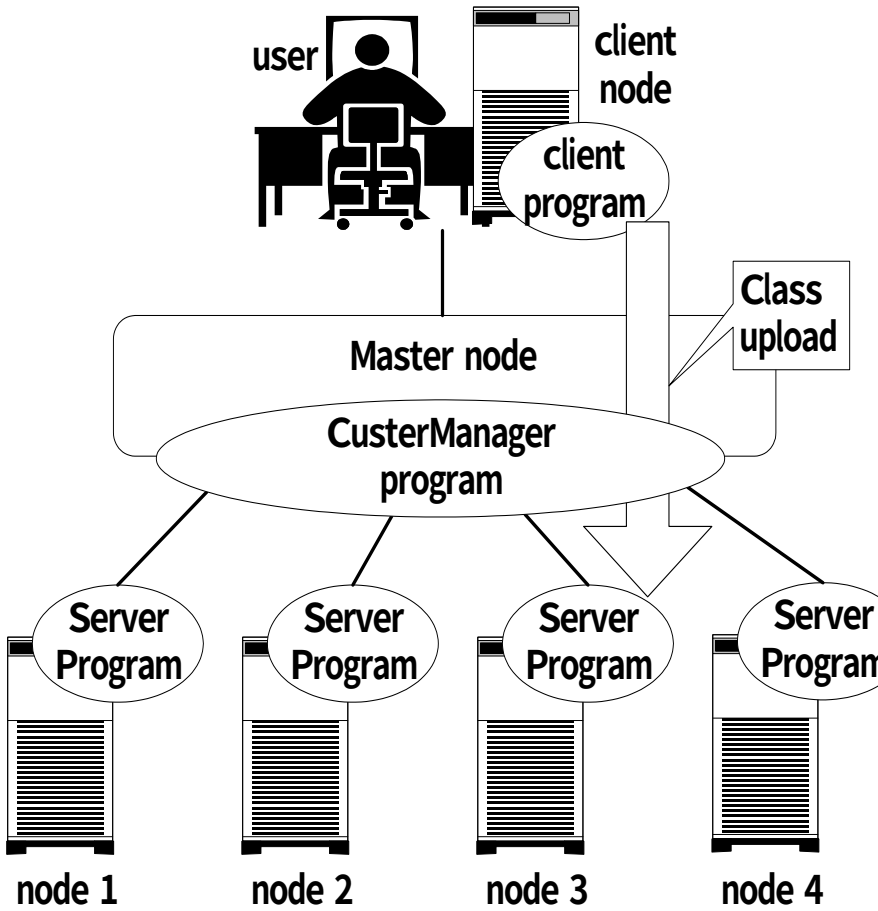
1. 本研究の背景、目的
2. jPoPの概要
3. 分枝限定法用クラス群jPoP-BB
4. jPoP-BBプロトタイプの実装
5. 性能評価実験
6. まとめと今後の課題

jPoPのしくみ

- Javaで実装

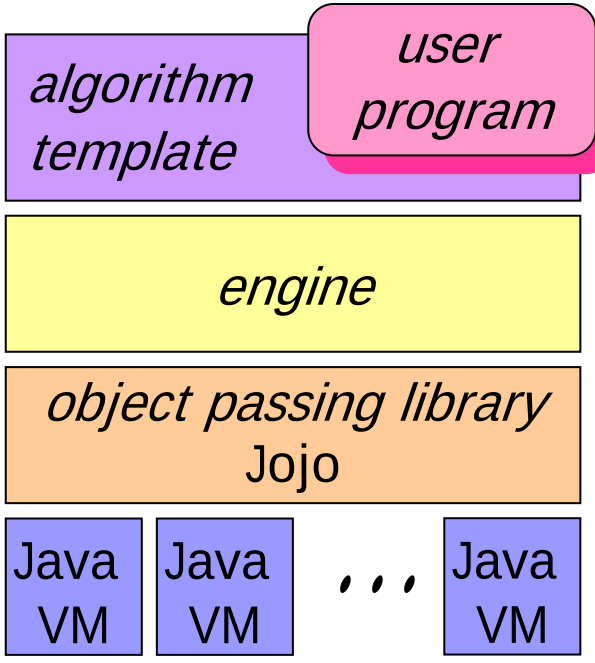
→任意のプラットフォームで実行可

- 自動的に実行プログラムをアップロード
- リソースの保護
  - サイト内で認証されたコードのみを実行
  - セキュリティマネージャの設定



jPoPアーキテクチャ

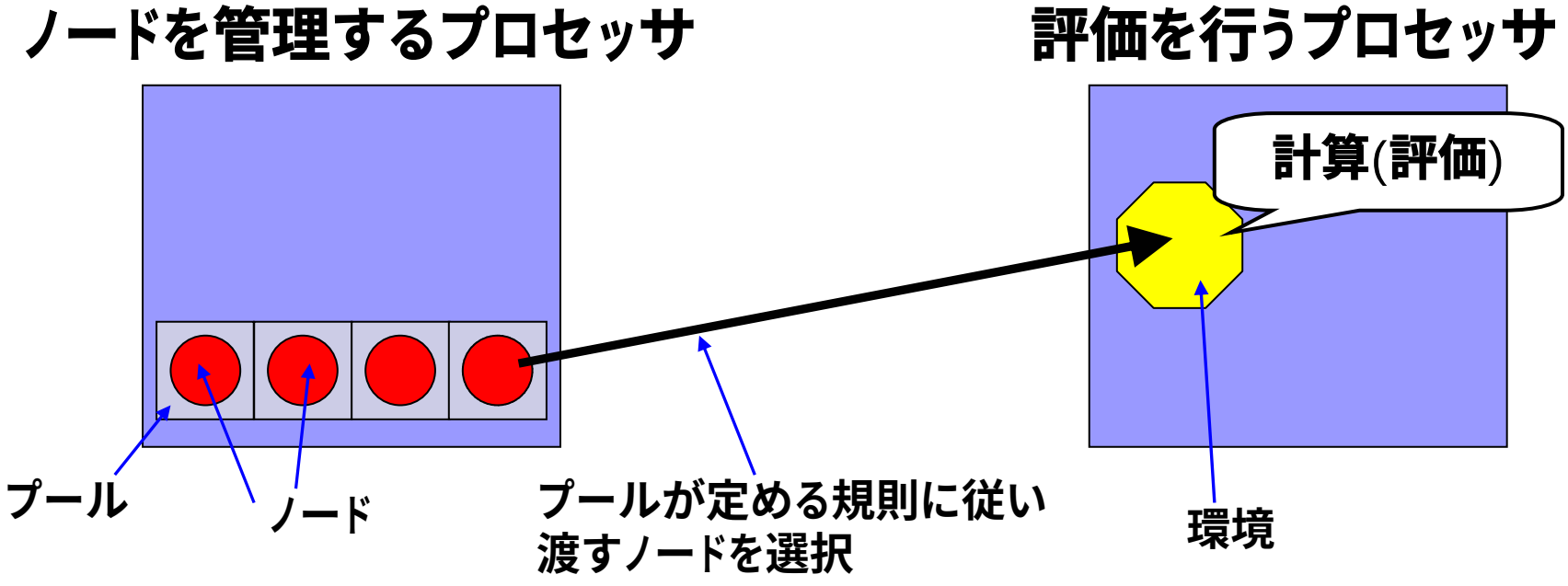
- 最適化アルゴリズムのテンプレート(抽象クラス)
  - アルゴリズムに特化したデータ構造・操作を定義
- エンジン
  - テンプレートを操作し、アルゴリズムを実行
- Jojo[中田ら,02]
  - オブジェクトパッシング通信ライブラリ
  - 階層制御を実現(将来的に役に立つ)



jPoP-BBの設計

- jPoP-BB利用にあたり、ユーザは以下の情報(問題依存領域)を記述すればよい
  - ノード(部分問題)
    - (部分)問題をあらわすクラスのテンプレート
    - 適用問題のデータ構造などを定義
  - 評価環境
    - 下界値、許容解などの計算の方法を記述
  - プール(探索方法)
    - ノードオブジェクトの保持
    - 指定された探索法で次ノードの選択を行う

各情報のはたらき

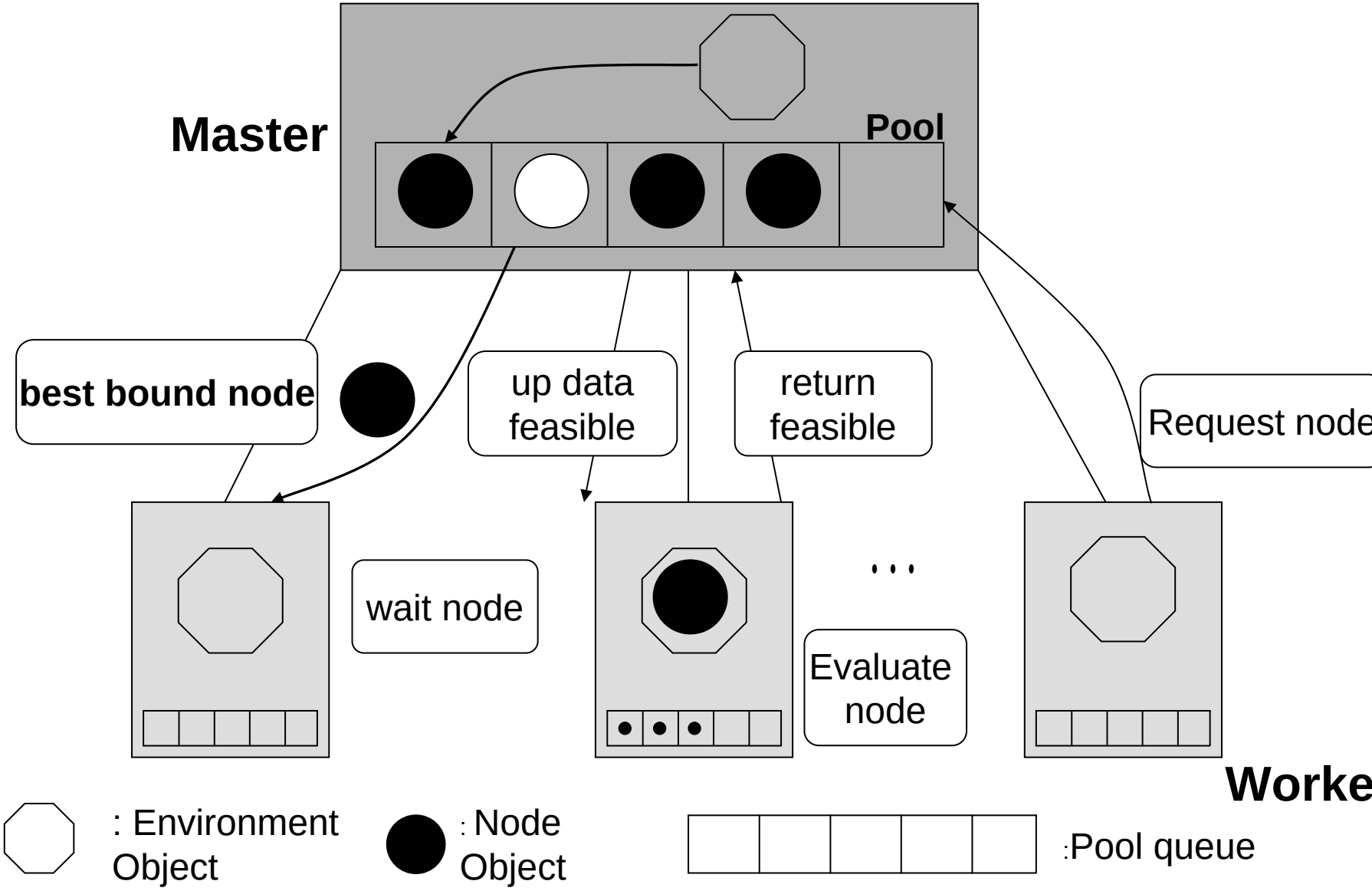


- ノードと環境の分離により通信コスト削減
  - 環境は起動時に各プロセッサに配布

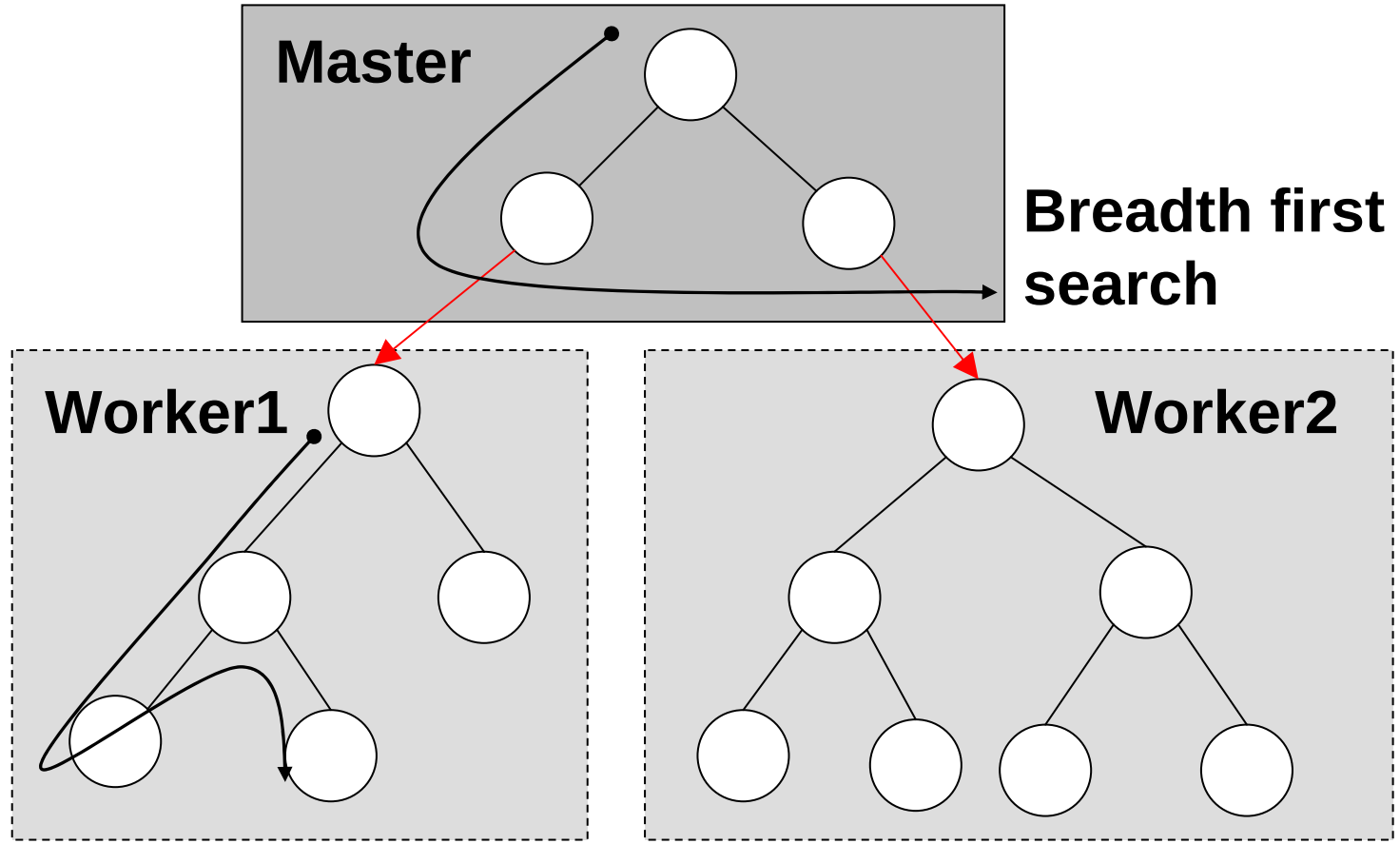
プロトタイプエンジンの設計方針

- 非階層的マスタ･ワーカ方式を採用
  - ワーカ間の通信は無い
- 探索方法も定義
  - 並列実行に幅優先･深さ優先混合探索を採用
- 負荷分散機構
  - マスタ内で問題をワーカ台数の数倍の子問題に分割し、1つずつワーカに与える
- 暫定解･下界値を伝播
  - 更新の度に全ワーカにマスタ経由で情報を伝達

非階層的マスタ･ワーカ方式



探索方法



Depth first search

定義例

- 0-1ナップサック問題

  目的関数: $z = -\sum_{j=1}^{n} c_j x_i \rightarrow$ 最小

  制約条件: $\sum_{j=1}^{n} a_j x_i \leq b,\ x_j = 0,1 (1 \leq j \leq n)$

- ノードクラスと環境クラスを定義
  - 用意されたテンプレートに従い、それぞれのクラスを記述
- プロパティファイル

```
jpop.bb.BBNode.classname = KnapNode
jpop.bb.Environment.classname = KnapEnvironment
jpop.bb.Environment.ItemFileName = knap.dat
jpop.bb.Driver.searchInSorted =false
```

ノードクラスの定義例

- getLowerBound(**下界値計算メソッド**)
- getFeasible(**許容解計算メソッド**)

```
import silf.jpop.bb.*;
import silf.util.*;
import java.util.*;
public class KanpNode extends BBNode{
   public double array[];
   public int branch[];
   public int feasible;
      :
    KnapEnvironment env = new KnapEn...;
      :
   public double getLowerBound(){
      return env.calcLower(this);
   }
      :
```

```
   public BBNode getFeasible(){
       KnapNode node;
       return env.calcFeasible(this);
   }
      :
   public BBNode [] branch(){
      :
      KnapNode a = (KnapNode)this.clone();
      KnapNode b = (KnapNode)this.clone();
      a.branch[i] = 1;
      b branch[i] = 0;
      :
      return new BBNode []{(BBNode )a,
                           (BBNode)b};
    }
}
```

環境クラスの定義例

- calcLower, calcFeasibleの内容に基づきノードの評価が行われる

```
mport silf.jpop.bb.*;
mport silf.util.*;
mport java.util.*;
ublic class KanpEnvironment  implements ...{
  final double capacity;
  final int number;
  final int [] value;
  final int [] weight;

  public void init(SilfProperties prop){
      :
  }

  public double calcLower(KnapNode node){
     double g;
     double tmp;
      :
```

```
      for(int =0; I<node.array.length;i++){
         switch(node.array[i]){
         case 1:･･･
         case -1:･･･
         case 0:･･･
         }
       }
       return g;
    }

    public KnapNode calcFeasible(Knap node){
       :
       for(int i=0; i<node.branch.length;i++){
          node.feasilbe+=value[i]*node.･･･
       }
       return node;
     }
}
```

評価実験

- 0-1ナップサック問題を用い、並列処理による台数効果の有無を調べる
  - 様々な性質の問題について実験するため、まず逐次実行で問題自体の性質を調べる
    - パラメータの違いによる実行時間の変化

評価対象の問題

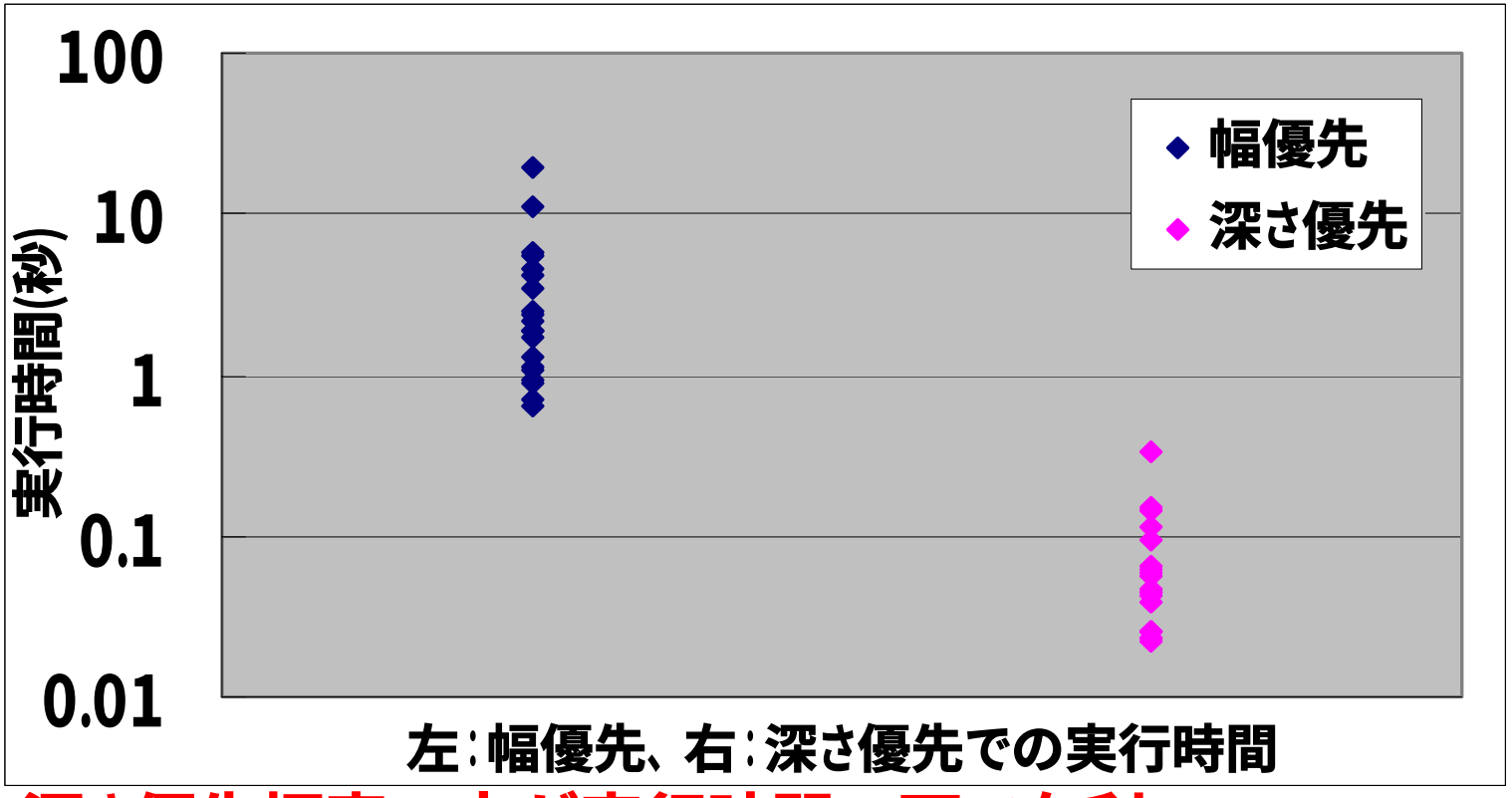
- 0-1ナップサック問題
  - 荷物の個数:100,150
  - 荷物の価値の最大格差:1.1倍,1.5倍,2倍
  - 荷物の重量の最大格差:2.5倍(固定)
  - ナップサックの容量:総重量の50%,80%
  - 各荷物の価値、重量はランダムに決定
- パラメータの組み合わせ12×乱数10通り
  →120問

評価環境(逐次実行)

- OS→Linux2.4.2-2
- Java→java 1.4.1_01
- CPU:1GHz,Memory:512MB

探索方法による違い

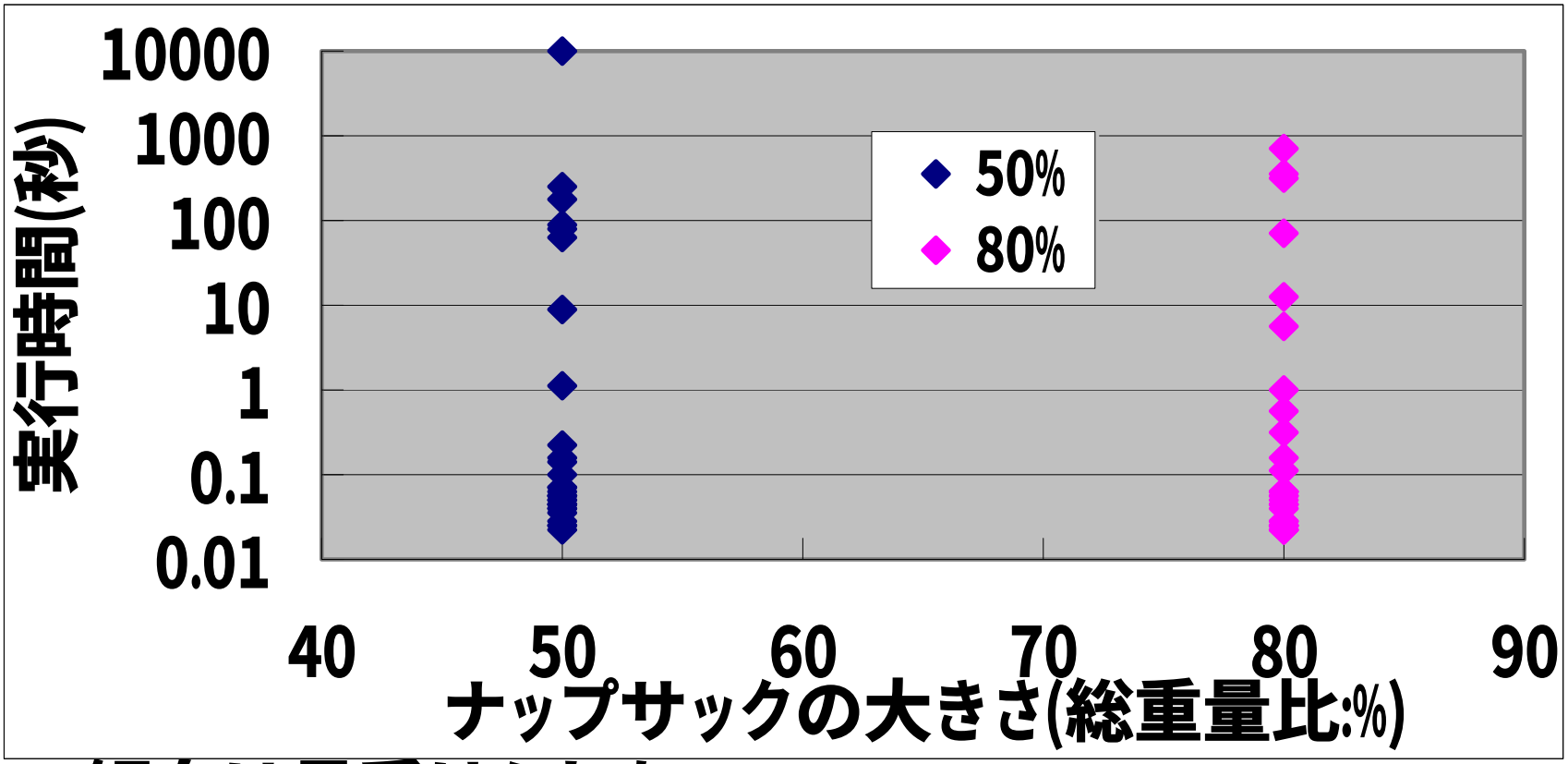
- 代表的な2種類の探索方法で同じ問題を解く



- 深さ優先探索の方が実行時間の面で有利
  →並列実行でさらなる短縮を目指す

ナップサックの大きさによる違い

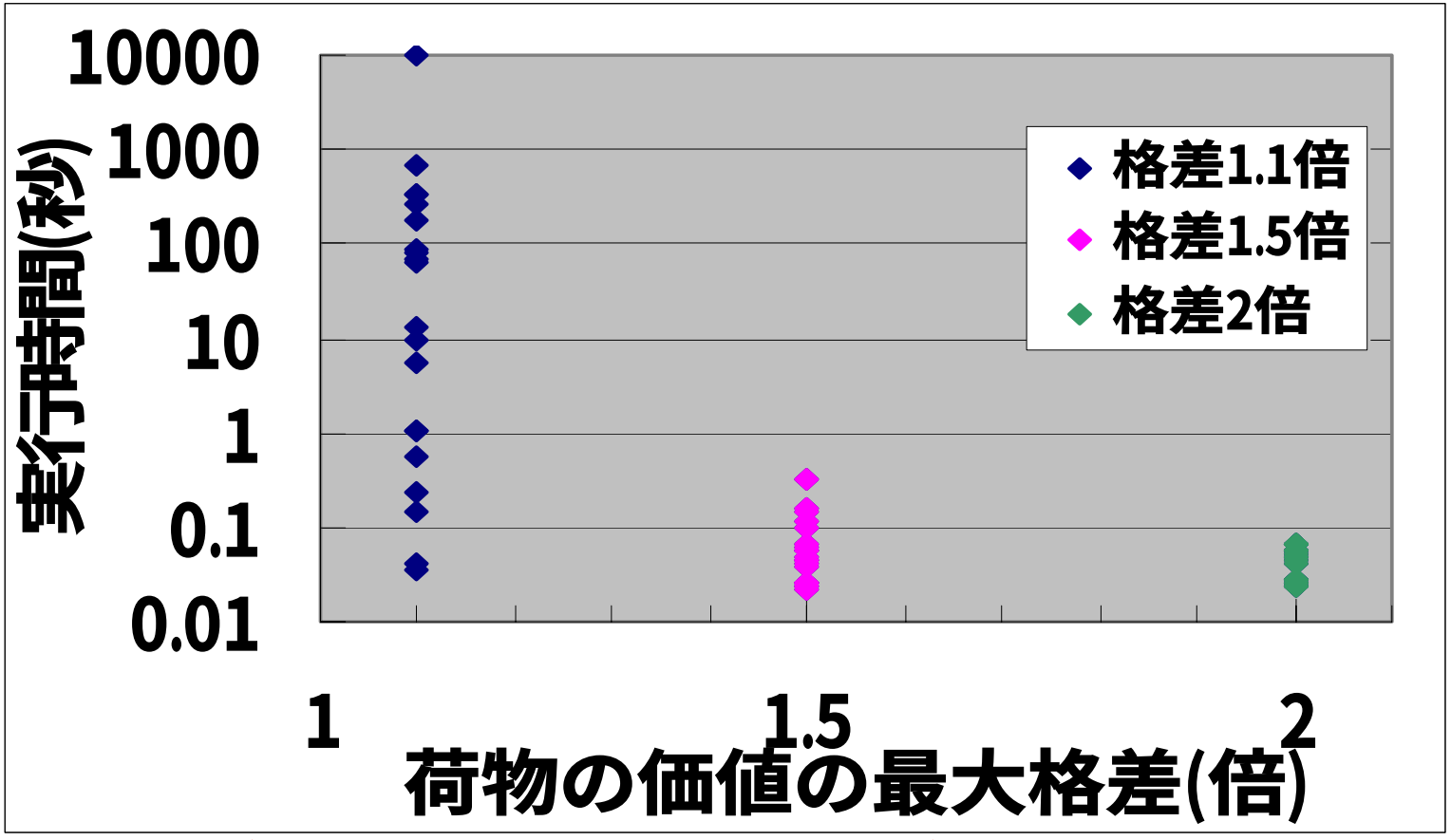
- 100%、0%に近い方が実行時間が短いと予想



- 傾向は見受けられない

荷物の価値による違い

- 格差が大きい方が実行時間が短いと予想



- 格差が小さいほど限定操作がしにくい

逐次実行と並列実行の比較

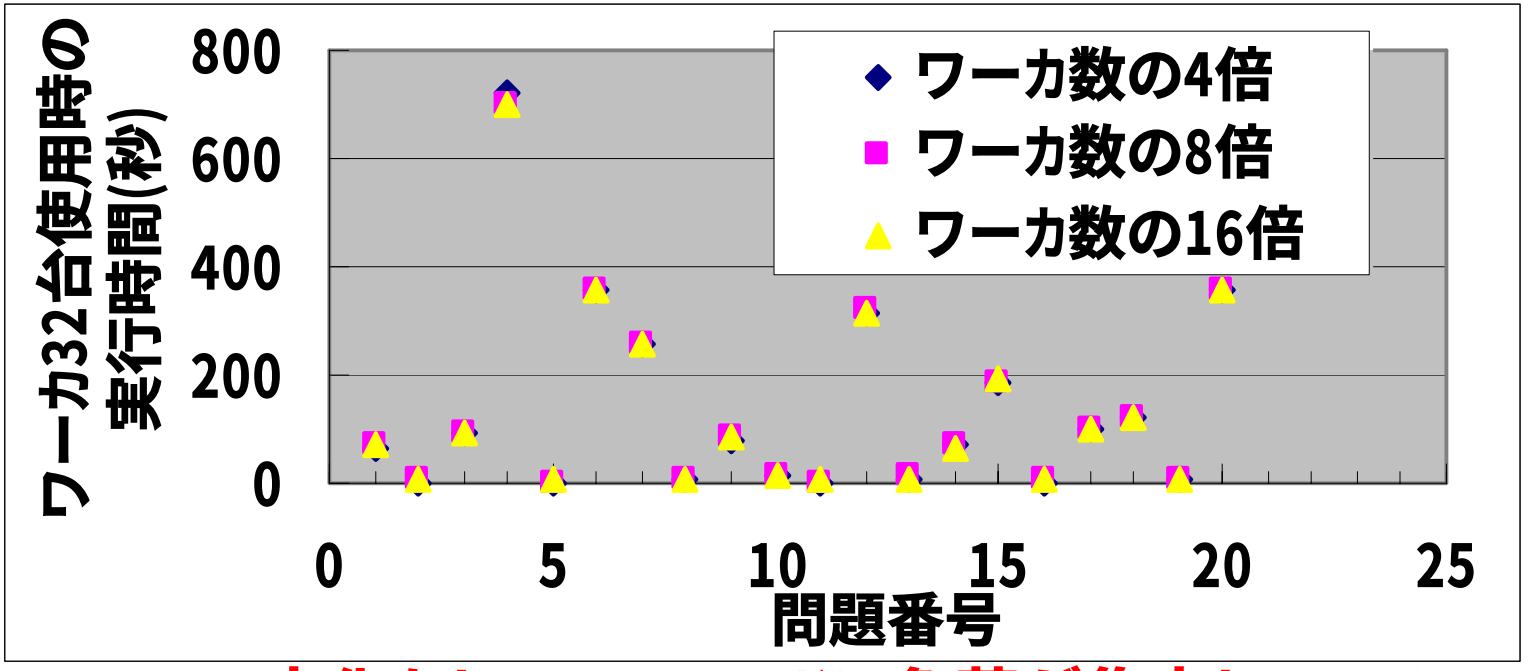
- 逐次実行では深さ優先探索が有利
- 逐次での実行時間が0.2〜10000秒の24問を並列で解き台数効果を評価
  - ワーカを2,4,8,16,32台使用
  - マスタでの展開ノード数は
    ワーカ台数×4(理由は後述)
  - 各3回解いて平均を採る
  - 実行時間は計算時間のみ、アップロード等の時間は含まない

評価環境(並列実行)

- **東工大内の**Grid**である**Titech-Grid**を構成するクラスタの**1**つ**
  - OS→Linux2.4.2-2
  - Java→java 1.4.1_01
  - **マスタ(クラスタの管理ノード)→** CPU:1GHz,Memory:512MB
  - **ワーカ→**CPU:1.4GHz×2,Memory:512MB
  - **通信レイテンシ→**0.075sec.
  - **接続→**100base-T

マスタでの展開ノード数による違い

- ワーカ数の4倍、8倍、16倍にして実験
  - 展開ノード数多いほど負荷分散ができると予想

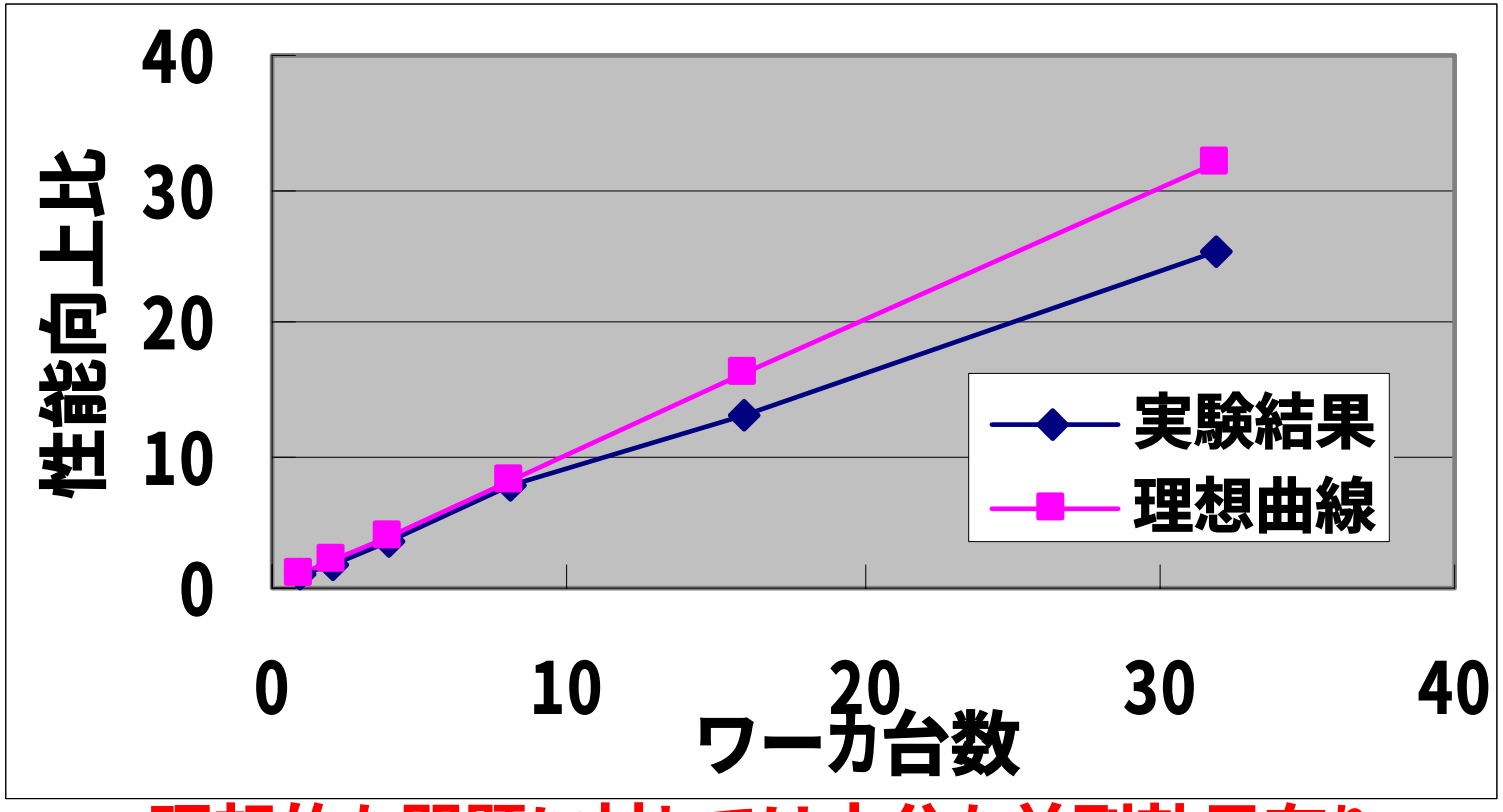


→変化なし(1つのノードに負荷が集中している)

→以下、並列実験はワーカ数×4で行う

理想的な問題での台数効果

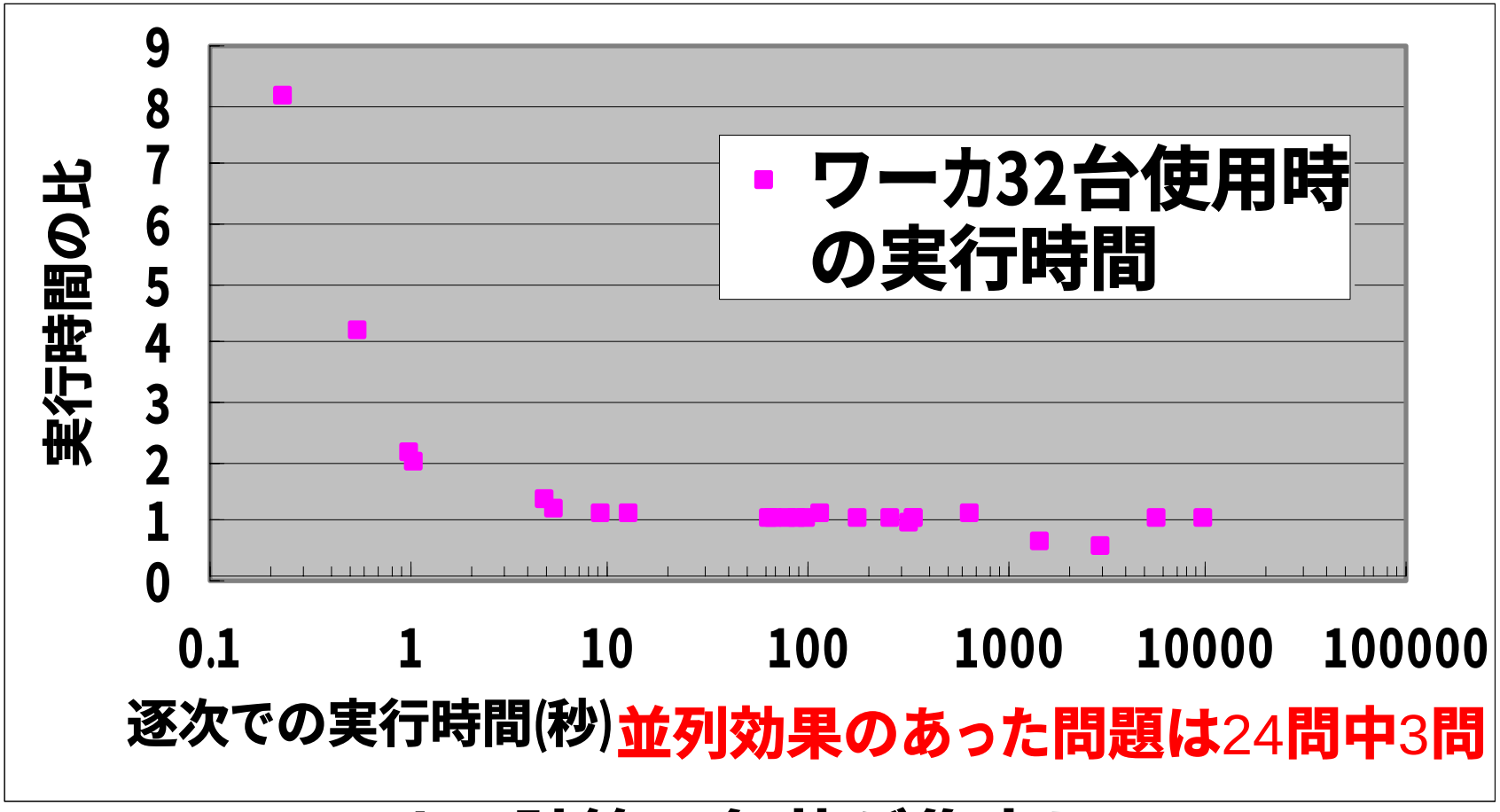
- 30個の荷物全てが同重量、同価値の問題
  - 他ノードの暫定解による限定操作が無い



→理想的な問題に対しては十分な並列効果有り

ランダムな問題での台数効果

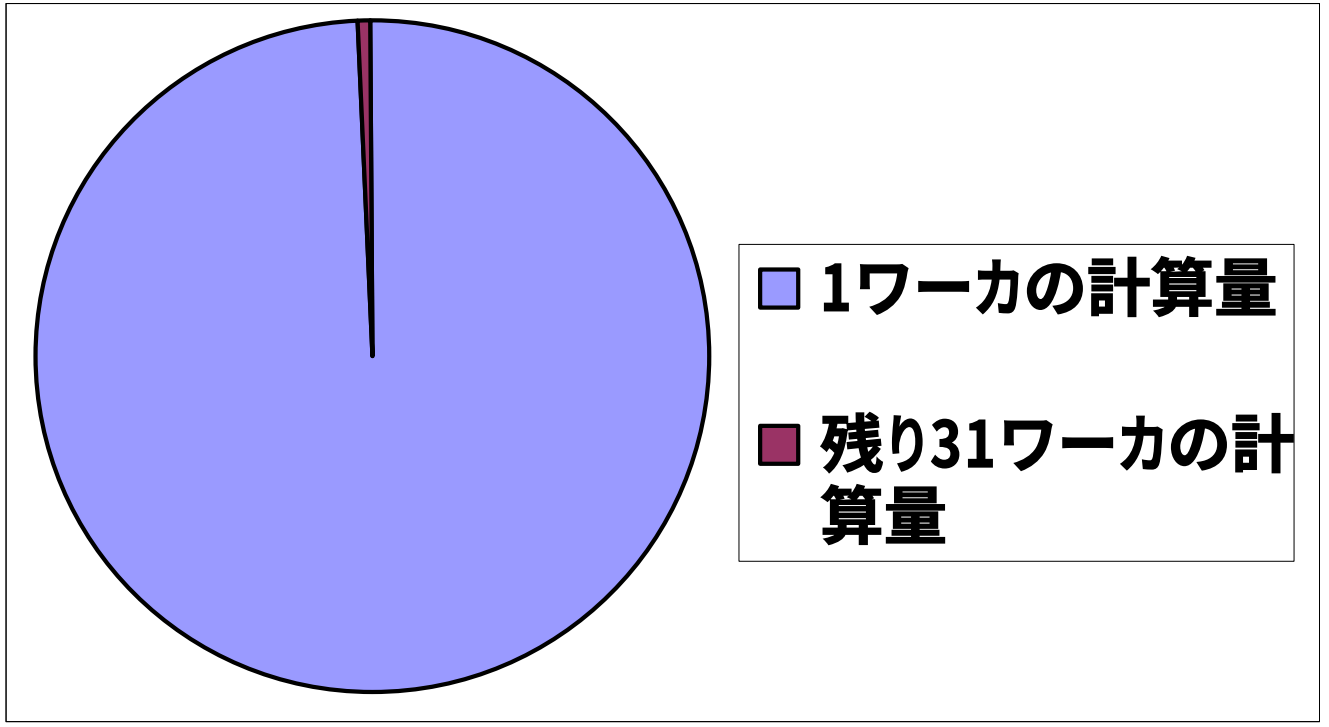
- 逐次での実行時間とワーカ32台使用時の実行時間を比較



- 1つのワーカに計算の負荷が集中している

負荷の偏り

- 計算量の99%以上が1台のワーカに集中したケース



→限定操作がほとんどできないノードの存在

他のノードは暫定解によりすぐに限定操作ができる

考察

- 台数効果の有無＝負荷分散の成否
  - 99%以上が1ワーカに集中したケースも
- 負荷分散の手法
  - マスタでの展開ノード数を増やしても現時点では効果無し
  - 負荷集中を感知しさらなる負荷分散を行う機構の実装(ノードのスティール)

→実現できれば並列効果は飛躍的に向上

まとめ

- jPoPはユーザに以下の利点をもつ並列処理環境を提供する
  - ユーザが並列･分散処理を考慮する必要無し
  - 問題依存領域の定義のみで実行可能
- jPoP-BBのプロトタイプを実装し、0-1ナップサック問題を用いて性能を評価
  - マスタでノード展開を行い負荷分散を図る
    →期待された性能向上は示されなかった
  - しかし理想的な問題に対しては並列効果有り

今後の課題

- 負荷分散機構の実装と評価
- 他の探索法(下界値優先探索など)の実装
  - 実現すれば全般に実行時間を短縮可能
- 階層的なマスタ･ワーカ構造の導入
  - マスタの分割によるマスタの負荷の軽減
  - さらに大規模なシステムでもスケーラビリティと安定性を確保するため