# 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[†] 飯野彰子[†]
中田秀基[††,†] 松岡聡[†,†††]

多次元パラメータ関数の最適値を求める組合せ最適化問題の解法としては、分枝限定法や遺伝的アルゴリズムなどが知られている。これらの解法はグリッド上での実行に適しているが、グリッド上での分散並列プログラミングは煩雑である上、実行時にも実行ファイルや設定ファイルをユーザがインストールしなければならないといった問題がある。我々はこれらの問題を解決し、最適化問題解法のグリッド上での実行を容易にするシステム jPoP を開発している。本稿では、jPoP の分枝限定法用のテンプレートクラスである jPoP-BB の設計およびマスタ・ワーカ方式を用いたプロトタイプの並列実装について述べる。また、0-1 ナップサック問題を用いた評価に関しても報告する。評価の結果、本実装では、他ノードの影響による限定操作が生じない、並列化に理想的な問題に関しては並列効果が得られるものの、一般的な問題に関しては負荷が不均等になり並列効果が得られないことがわかった。

# Parallel Combinational Optimization System for the Grid: jPoP With Applying Branch-and-Bound method

Shingo Nakagawa ,[†] Akiko Iino ,[†] Hidemoto Nakada [††,†]
and Satoshi Matsuoka[†,†††]

For combinatorial optimization problems, which compute the optimal value of a multi-dimensional parameter function, several methods are known to be effective, such as Branch-and-Bound methods, Genetic Algorithm, etc. They are considered to be suitable for executing on the Grid, but distributed parallel programming on the Grid is quite complicated and furthermore setting up the Grid-wide computing environment is a heavy burden. Here, we propose a system called jPoP, which makes it easy to develop and execute optimization-problem solvers on the Grid. In this paper, we focus on the Branch-and-Bound method and describe design and implementation using Master-Worker method in detail. We also report on the evaluation of the system using 0-1 knapsack problems. While we observed remarkable speedup for a certain kind of problem, it could not show speedup for general problems owing to load-inbalance.

## 1. はじめに

我々は現在、グリッド上のアプリケーションとして組合せ最適化問題[1] に注目している。組合せ最適化問題は実社会において、スケジューリング、設計問題、生産計画など広大な応用範囲を持つ問題であり、かつ自明な並列性が大きく実行粒度の調整の自由度も高い。これまでも我々は分枝限定法や遺伝的アルゴリズムに対して Ninf-1 システム[2] を適用し、これらの問題に対するグリッド技術の有効性を確認してきた[3),4)]。

しかし、一般的にグリッドアプリケーションを実装することは、1) 広域に分散したアーキテクチャや OS、性能などが異なる計算リソース (PC クラスタ、スパコン等) 群の取り扱い、2) グリッド上の異なるサイト間の安全な通信、リソースの保護が必要、3) 通信、同期、負荷分散などの並列プログラミングの知識が必要、といった問題のため困難である。さらに、組合せ最適化アプリケーションでは、4) 組合せ最適化問題のアルゴリズム、データ構造などを一から分散実装する、という煩雑さがある。

上記に挙げた 1)、2)、3) に関しては、Ninf-1 等のグリッド RPC システムを用いることで、それまでの既存の方法に比べて大きく負担が軽減されることが確認されている[3),4)]。しかし、グリッド RPC システムを用いてもアルゴリズムプログラマにとっては 4) のような問題は解決されず、その負担は依然として大きい。

この問題を解決するために、我々は並列組合せ最適化システム jPoP[5] を提案している。jPoP は、代表的な数種の並列組合せ最適化アルゴリズムのテンプレー

† 東京工業大学 Tokyo Institute of Technology
†† 産業技術総合研究所 National Institute of Advanced Industrial Science and Technology
††† 国立情報学研究所 National Institute of Information

トを提供し、プログラムの並列化、安全性などの問題をプログラマから隠蔽する。プログラマは問題領域依存なデータ構造や操作を定義するだけで、グリッド上で組合せ最適化アプリケーションを容易に開発でき、かつ安全に実行することができる。

本稿では、jPoP の概要と、最適化アルゴリズムの一つである分枝限定法用クラス群 jPoP-BB の設計、およびそのプロトタイプの実装について述べる。

## 2. jPoP の設計

jPoP は実行環境として複数のクラスタから成るグリッド環境を想定している。これは異なるサイトにおかれた比較的小規模なクラスタを複数結合して形成されるような環境であり、将来のグリッドとして一般的になると考えられる。このような環境におけるアプリケーションには以下のような要請がある。

- 任意のプラットフォームでの実行
- 高いセキュリティをもった通信とリソースの保護
- 並列プログラミング

さらに、組合せ最適化問題を解くとなると、

- 並列度のスケーラビリティの獲得が困難

という問題が挙げられる。既存のグリッド上での組合せ最適化アプリケーションはマスタ・ワーカ方式の実装が一般的であり、これまでの実験ではマスタが一つに対してワーカが数台から数十台規模のローカルな環境におけるものに過ぎなかった。従って、数百台さらには数千台といった大規模な並列環境にスケールできるかどうかは確認できていない。☆

以下では、上記の要請や問題に対して jPoP がとる解決法について述べる。

### 2.1 プラットフォーム独立性

jPoP は実装に Java を用いる。Java バイトコードのポータビリティによってプラットフォームを選ばない実行が可能となっている。さらに、jPoP ではシステムプログラムおよびユーザプログラムのバイトコードを自動的にステージングする。このため、使用する個々のノードにプログラムをインストールする必要がない。

### 2.2 安 全 性

広域に分散する計算資源を安全に活用するために、異なるサイトのノード間においては、Globus[9] の GSI(Grid Security Infrastructure)[10] や ssh といった安全な通信をサポートする。

また、ユーザコードをグリッド上で実行する場合には計算資源をユーザコードから保護する必要がある。Java ではクラスローダ別にセキュリティマネージャを設定することができ、jPoP ではこれを利用してセキュリティサンドボックスを実現している。

☆ 実際、夏目らの実験[3] によって、単体のマスタに対し、ワーカが 16 台程度で通信時間がボトルネックとなり始め、性能向上が飽和してしまうことが報告されている

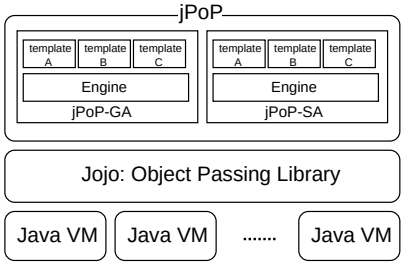


**図 1** jPoP レイヤ

### 2.3 並列プログラミング支援

jPoP ではプログラムを書く際にユーザが記述しなければならないのは、各アルゴリズムの問題依存領域部分であるデータ構造やその操作だけである。そのため、並列プログラミングで一般的に要求される、通信や同期、負荷分散について特別な記述をする必要がない。このため、ユーザが並列プログミングを意識することなく並列プログラムを書くことができる。

### 2.4 アルゴリズム実装支援

jPoP におけるプログラミングは特定のインターフェイス (もしくは抽象クラス) のメソッドを実装していく。これはアプレット、サーブレットなどで用いられている方法と同じである。ユーザは対象となる問題のデータ構造とその操作を定義したメソッドを実装する。それぞれの最適化アルゴリズムでは、当然必要となるデータ操作が異なる。また、あるアルゴリズムの中にも複数のアルゴリズムフレームワークがあり、それぞれ異なるデータ操作が必要となる。このため jPoP では、個々の手法に対してそれぞれに適した抽象クラスやインターフェースを提供する。これにより、ユーザは最低限の定義だけでアルゴリズムを記述できる。

## 3. jPoP の実装

前節で提案されている解決法を実現するため、jPoP はその下位レイヤとして、我々が開発した階層型実行環境 Jojo[11] を用いて実装されている。Jojo は java で実装された、階層構造をもつグリッド環境を前提としたオブジェクトパッシング通信ライブラリで、これを用いて実装されている jPoP も同様に階層的な制御が可能となっている。これにより、jPoP は前節に述べられている要請や問題にも対処できるようになっている。**図 1** に Jojo を下位レイヤとした jPoP の全体レイヤを示す。

jPoP は、Jojo のアプリケーションとして実現されている。Jojo によって通信レイヤや実行環境などが隠蔽されるので、jPoP 自身は実行環境に依存しないポータブルなシステムとなっている。

jPoP は個々の解法に対して、テンプレートとなる

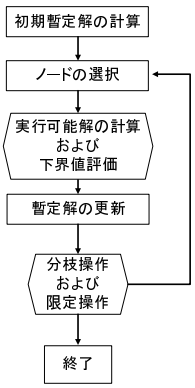


**図 2** 分枝限定法の処理手順

抽象クラスと、この抽象クラスを操作して解法を実行するエンジン部を提供する。ユーザはテンプレートクラスを具体的なクラスで実装することでプログラミングを行う。エンジン部は Jojo の Code クラスのサブクラスとして実現され、分散して並列に動作して解法を実行する。このエンジン部はさまざまな種類のものが実装可能であり、これを取り替えることでさまざまな並列化手法を同じ問題に対して適用できる。

以下では、今回実装の対象とした分枝限定法用のクラス群 jPoP-BB について述べる。

## 4. 分枝限定法用クラス群 jPoP-BB

jPoP のひとつとして、分枝限定法を分散環境で実行するための枠組である jPoP-BB を設計、実装した。これを用いることで、データ構造や分枝操作などが定義されたノードと、実行可能解及び下界値評価が定義された評価環境をクラスとして記述するだけで、様々な分散環境で分枝限定法で並列計算を行うことができる。もちろん、一つのマシンで逐次に実行することも可能である。また、分枝限定法のノードの探索法に関してもユーザが独自の実装を与えることができる。

### 4.1 jPoP-BB の設計

分枝限定法の一般的な処理手順を**図 2** に示す。jPoP-BB においては、ユーザは問題依存領域に含まれる事項、すなわち下界値や暫定解の計算方法、分枝操作、解の探索方法などを記述するだけでよく、図 2 にあるような手順まで記述する必要はない。解きたい部分問題を定義するための要素はノードに記述され、評価環境にはノードを評価するための下界値や暫定解の計算方法などが記述される。ノードは部分問題の生成に伴ってその問題によって変化するデータであり、評価環境は起動時に各マシンが共有すればよいデータである。このように問題依存領域を分けることで、通信量を小さく抑えるように設計してある。

### 4.2 jPoP-BB の API

jPoP-BB では基本的には、その実行全体の制御を行っているのは Driver クラスである。この部分はユーザからは完全に隠蔽されており、ユーザは一切記述を行うことなく、前項で示された項目を定義するだけで分散環境で分枝限定法を用いて組合せ最適化問題を解くことができる。以下でそれぞれの詳細について述べる。

```
   /*static 変数 prop に設定されている
              SilfProperties を用いて自らを初期化*/
public static void initializeProperties();
   /*static 変数 env に設定されている
     Environment を用いて下界値評価を行い結果を返す*/
public double getLowerBound();
  /*static 変数 env に設定されている
     Environment を用いて許容解の計算を行い結果を返す*/
public BBNode getFeasible();
  /*分枝操作で新たなノードを生成し、それを返す*/
public BBNode [] branch();
```

**図 3** BBNode オブジェクト

```
interface Environment extends Clonable Serializable{
    /*初期化*/
  void init(SilfProperties prop);
}
```

**図 4** Environment インターフェース

#### 4.2.1 ノードの定義

ノードをあらわす BBNode オブジェクトは silf.jpop.bb.BBNode クラスのサブクラスとして実装する。このクラスは**図 3** のメソッドを実装する必要がある。また、対象となる問題のデータ構造などもここで定義する。

#### 4.2.2 環境の定義

ノードを評価する環境をあらわすオブジェクトは、**図 4** のような Environment インターフェースを実装しなければならない。ここに記述される部分は、並列実行の場合、起動時に 1 つだけ生成され、そのコピーがリモート側である各マシンに配布されて実行される。

#### 4.2.3 プールの定義

ユーザが指定した探索法でノードの選択を行う Pool オブジェクトは Pool クラスのサブクラスとして実装される。jPoP-BB はデフォルトで、一般的な探索法である 1) 深さ優先探索、2) 幅優先探索を Pool サブクラスで提供するので、必ずしもこの部分をユーザが実装する必要はないが、ユーザが独自の探索法を定義することも可能である。

### 4.3 プロトタイプ実装における並列エンジン

前項までに述べた設計方針等に基づき、エンジン部を実装して jPoP-BB のプロトタイプを作成した (以下、jPoP-BB プロトタイプと呼ぶ)。このプロトタイプでは、jPoP-BB はマスタ・ワーカ方式 (**図 5**) で実装されており、以下のように実行される。

(1) マスタが探索のルートとなるノードを作成

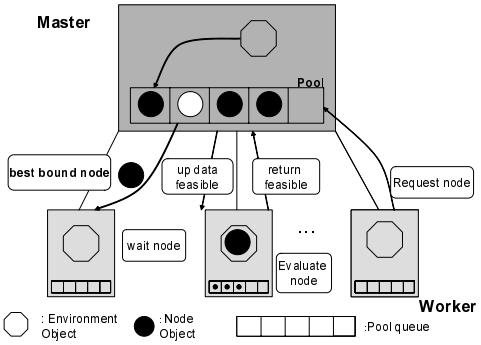


**図 5** jPoP-BB プロトタイプの概要

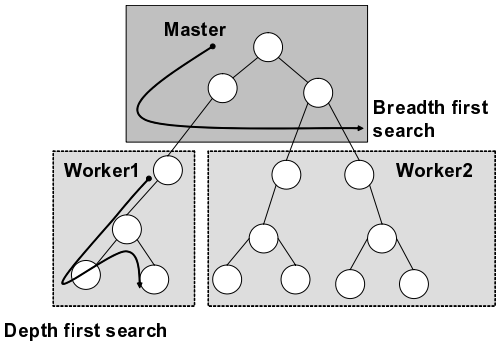


**図 6** jPoP-BB プロトタイプにおける探索の手順

(2) マスタはルートノードを自身が保持するプールに入れる
(3) マスタはプールからノードを取り出して評価し、分枝操作が行われた場合は生成されたノードをプールに戻す
(4) (3) を幅優先探索で繰り返し、プールが保持するノードが一定数になるまで行う
(5) ワーカは、自身が計算を行うノードを保持してなければ、マスタに対してノードを送信するように要求を出す
(6) ワーカからの要求によりマスタはプールから下界値優先 (もっとも下界値のよいノードを優先させる) でノードを取り出しワーカに与える
(7) ワーカは受け取ったノードを深さ優先探索で評価する
(8) ワーカは探索中に暫定解や下界値の更新があった場合はマスタに通知し、マスタはそれを全ワーカに通知する
(9) ワーカは与えられたノードの評価が終了したら、暫定解をマスタに通知して (5) に戻る
(10) (5)〜(9) を繰り返し、マスタのプール内と、全てのワーカにノードがなくなった時点で計算が終了し、マスタがもつ暫定解が最適解となる

ここで述べた探索の手順を**図 6** に示す。(4) におけるマスタでの展開ノード数は、ワーカの数より多く設定される。これは、各ワーカが評価するノードの数を動的に変えることで、計算負荷の均等な分散を実現するためである。

```
import silf.jpop.bb.*;
import silf.util.*;
import java.util.*;

public class KnapNode extends BBNode{
  public double array[];
  public int branch[];
  public int feasible;
        :
  KnapEnvironment env = new KnapEnvironment;
        :
  public double getLowerBound(){
    return env.calcLower(this);
  }

  public BBNode getFeasible(){
    return env.calcFeasible(this);
  }
        :
  public BBNode [] branch(){
        :
    KnapNode child1 = (KnapNode)this.clone();
    KnapNode child2 = (KnapNode)this.clone();
    child1.branch[i] = 1;
    child2.branch[i] = 0;
        :
    return new BBNode []{(BBNode)a, (BBNode)b};
  }
}
```

**図 7** BBNode クラスの定義例

### 4.4 定 義 例

本稿の数値実験で用いた 0-1 ナップサック問題を最適化するためのノードクラス (**図 7**) と環境クラス (**図 8**) の定義を示す。ノードクラスの getLowerBound メソッドで下界値を求め、getFeasible メソッドで実行可能解を求めているが、実際には環境クラスのそれぞれのメソッドで評価される。並列実行ではこの環境クラスが各ノードでアップロードされ、Driver クラスを経由して環境クラス内のメソッドで実行される。

## 5. jPoP-BB プロトタイプの性能評価

一般に、分枝限定法の並列化による実行時間の改善の評価は困難である。なぜなら、分枝限定法においては、同じ逐次実行でも探索手法によって実行時間・メモリ使用量が大きく変わり、さらに並列実行時には計算の分散の手法や、プロセッサ間の暫定値や下界値の情報交換のタイミングなどのユーザが関知できない要因も影響するため、計算の量そのものが不規則に変化するからである。このため、実行時間改善の評価にあたっては、さまざまな性質の問題を作り、それぞれ数回実行して統計的に議論する必要がある。

本稿では、さまざまな条件下で問題をランダムに作成し、それらを逐次実行、さらに並列実行することで、問題の性質、および jPoP-BB プロトタイプの並列効果を測定する。

なお、数値実験は、0-1 ナップサック問題を対象として行った。この問題は、荷物の数、各荷物の重量と価値、ナップサックの大きさで定義される。実験を行うにあたり設定したパラメータは以下の通りである。

- 荷物の数が 100 個、150 個

```
import silf.jpop.bb.*;
import silf.util.*;
import java.util.*;

public class KnapEnvironment implements Environment{
  final double capacity;
  final int number;
  final double [] value;
  final double [] weight;

  public void init(SilfProperties prop){
        :
  }

  public double calcLower(KnapNode node){
    double g;
    double tmp;
        :
    for(int i=0; i < node.array.length; i++){
      switch(node.array[i]);
      case 1;
        g = g + (double)value[i];
        break;
      case -1;
        if((tmp+ weight[i]>capacity){
          g=g+((capacity-tmp)/weight[i])*value[i];
          node.array[i]=(capacity - tmp)/weight[i];
        }else{
          tmp +=tmp+weight[i];
          g +=value[i];
        }
      case 0;
        break;
    }
    return g;
  }

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

    return node;
  }
}
```

**図 8** Environment クラスの定義例

- 各荷物の価値の最大格差が 1.1 倍、1.5 倍、2 倍
- 各荷物の重量の最大格差は 2.5 倍に固定
- ナップサックの容量が荷物の総重量の 50%、80%

実際には、各荷物の価値・重量は、与えられた格差の範囲内で乱数を用いてランダムに生成している。本実験では、上記の 4 パラメータを組み合わせて得られる 12 種類の問題に対し、乱数を 10 種類設定して、合計 120 問を生成した。

実行に用いたクラスタは、東京工業大学のグリッドである titech-grid を構成するクラスタの 1 つで、マスタが CPU1GHz、メモリ 512MB、ワーカが CPU1.4GHz × 2、メモリ 512MB である、58 台 (115CPU) から成るクラスタである。マスタ・ワーカ間の通信レイテンシは 0.075 秒で、100base/T で接続されている。

### 5.1 逐次実行における性質

分枝限定法を逐次実行する際に実行時間等に大きな影響を与えるのは、問題自体の性質と、探索方法である。そこで、生成した 120 問全てを逐次実行で解き、パラメータの変化による実行時間の変化を調べ、逐次実行における性質を議論する。

まず、探索方法は深さ優先に固定して、ナップサックの大きさだけが異なる 2 つの問題の実行時間の違いを調べたところ、 **図 9** と**図 10** に示すように、荷物 100 個の場合、150 個の場合とも、全体の傾向を分析することは不可能であった。問題によっては、ナップサックの大きさを総重量の 50%から 80%にすることで実行時間が大きく増加する問題、逆に大きく減少するものもあった。次に、各荷物の価値の最大格差だけが異なる 3 つの問題の実行時間の違いを調べたところ、**図 11** に示すように、最大格差を 1.1 倍とした

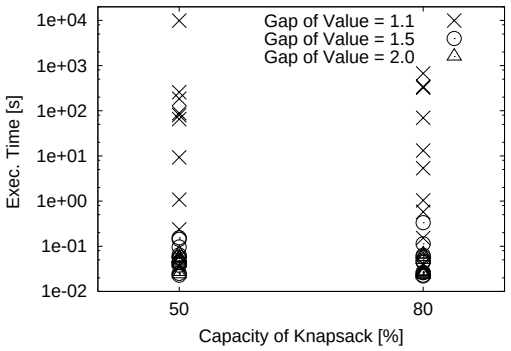


**図 9** ナップサックの大きさの違いによる実行時間の変化 (荷物 100 個)

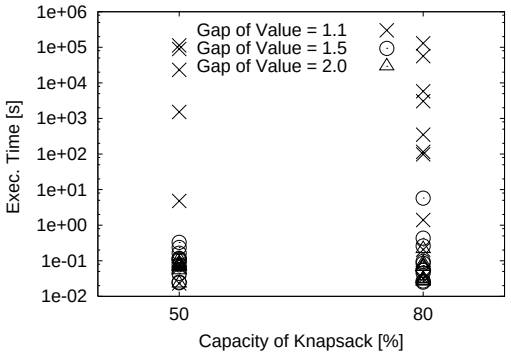


**図 10** ナップサックの大きさの違いによる実行時間の変化 (荷物 150 個)

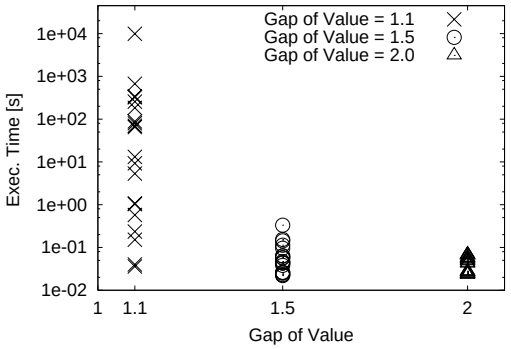


**図 11** 荷物の価値の最大格差の違いによる実行時間の変化 (荷物 100 個)

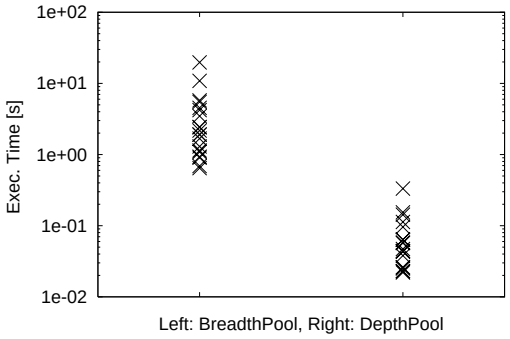


**図 12** 幅優先探索 (左) と深さ優先探索 (右) による実行時間

問題だけに、極端に時間がかかるケースが多く見られた。これは、1.1 倍の問題の場合、各荷物の単位あたりの価値に大きな差がないため (その格差は最大でも $2.5 \times 1.1 = 2.75$ 倍にとどまる)、計算の途中で暫定解を得ても、それより悪い下界値を出してしまう子問題が多いために限定操作が困難であることに起因すると考えられる。1.5 倍の問題と 2 倍の問題との間では、概して実行時間が減少しているが、問題によっては 1.5 倍から 2 倍にすることで逆に実行時間が増加するケースもある。これは単位あたりの価値の範囲が十分広ければ限定操作が十分有効に行えることを示している。

また、ここまでの議論は全て深さ優先探索においてのみ行ったが、幅優先探索での実行時間を 120 問のうち 20 問 (荷物 100 個、価値の最大格差 1.5 倍のもの) について調べた結果、 **図 12** のようになった。これより、深さ優先探索の方が明らかに幅優先探索より有利であることがいえる。実際、分枝限定法において幅優先探索は計算時間、メモリ使用量の両面で深さ優先探索に劣る場合が多い[13]。早期の段階で限定操作が可能な場合であれば必ずしもこの限りではないが、0-1 ナップサック問題ではそのようなケースは極めて稀であるため、このような結果になったと考えられる。

### 5.2 逐次実行と並列実行の比較

まず、負荷が均等に分散できた場合のシステムオーバーヘッドを見るため、特別な問題を作って並列での実験を行った。この問題は、40 個の荷物全てが同一の重量・価値をもつ問題である。この問題は他のノードの暫定解による限定操作が起こりえないため、特定のノードへの負荷の集中の度合は大きくないと考え、実験の対象とした。

なお、実験はワーカを 2 台、4 台、8 台、16 台、32 台使用した場合それぞれにおいて 3 回ずつ行い、実行時間の平均をとって逐次での実行時間と比較した。ワーカでの処理の前にマスタが幅優先探索で展開するノードの数はワーカの台数の 4 倍とした。また、実行時間は、実際にナップサック問題を解く時間であり、jPoP が各計算ノードに実行プログラムをアップロードしたり終了したりする時間は含まない。

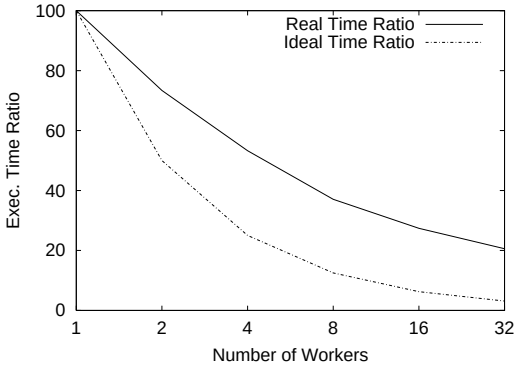


**図 13** ワーカ台数の変化による実行時間の変化
(同じ荷物 40 個を対象にした問題)

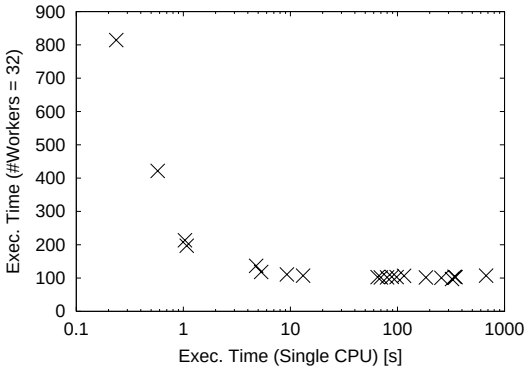


**図 14** SingleCPU とワーカ 32 台使用時の実行時間の関係
注:縦軸は逐次での実行時間を 100 とする。以下同じ。

結果は**図 13** に示すように、スケーラビリティに問題はあるものの、並列処理することによる実行時間の短縮が確認できた。各ワーカが計算を担当したノード数には、ワーカ間で最大格差 20 倍程度の差があり、これがスケーラビリティに問題が生じる理由であると考えられるが、これは問題自体の規模を大きくすれば解決できると考えられる。これより、jPoP-BB プロトタイプは理想的な問題に対しては十分に並列効果があるといえる。

これをふまえ、前項で解いた 120 の問題のうち 20 問について並列処理の実験を行った。前項での各図に見られるように、120 問のうちほとんどの問題は逐次実行でも 0.1 秒前後、あるいはそれ未満で計算を終了しているので、逐次での実行時間が 0.2 秒から 1000 秒までの範囲にある 20 問を抽出し、ワーカの台数を変えながら並列実行して並列処理による実行時間の変化を調べた。

結果の概要は**図 14** の通りである。ワーカ 32 台使用時に逐次実行より短い時間で処理を終えた問題は 20 問中わずか 1 問だけで、しかも逐次での実行時間が短

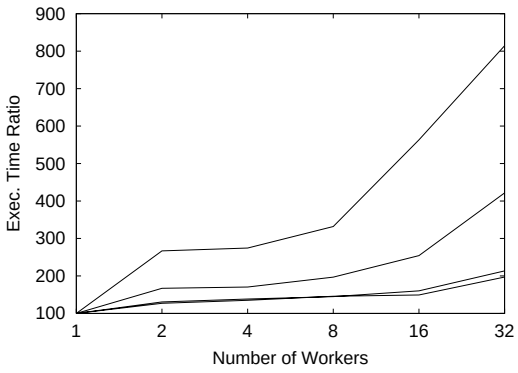


**図 15** ワーカ台数の変化による実行時間の変化
(逐次での実行時間が 4 秒未満の 4 問)

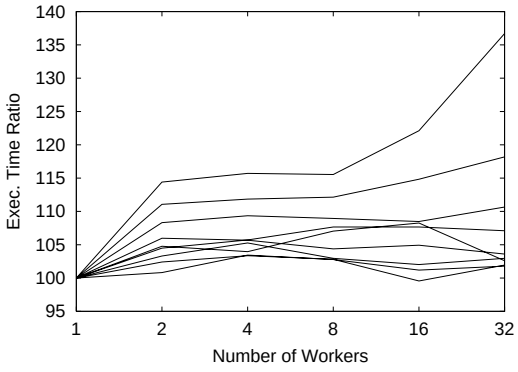


**図 16** ワーカ台数の変化による実行時間の変化
(逐次での実行時間が 4 秒以上 100 秒未満の 9 問)

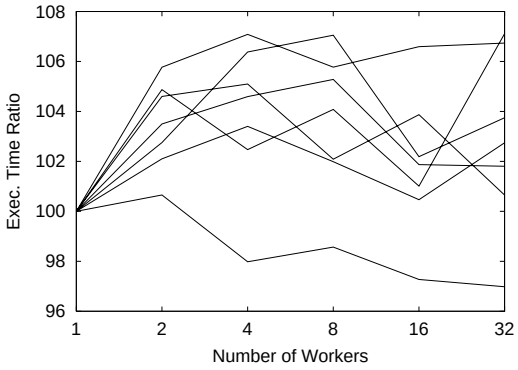


**図 17** ワーカ台数の変化による実行時間の変化
(逐次での実行時間が 100 秒以上の 7 問)

い問題ほど、並列での実行時間が延びる傾向がある。

詳細を**図 15**、**図 16**、**図 17** に示す。ワーカが 2 台〜16 台のときも、実行時間が逐次実行時より減少するケースはほとんどなく、また特に逐次での実行時間が短い問題においては、ワーカ数が増加するほど実行時間が増加する傾向がある。調べたところ、幅優先探索でノードを細かく分けることによるワーカの負荷の分散が、特定のノードにだけ計算が集中するためにうまくいかず、結果として 1 つのワーカだけに計算のほとんどが集中してしまっていることがわかった。このため、ワーカでの計算時間がほとんど変化せず、ワーカ・マスタ間の通信時間、および各ワーカの終了を待つ時間の分だけ、並列処理、特にワーカ台数の多い方が不利になると考えられる。

## 6. 考　　察

前節でも述べたように、今回実装したプロトタイプエンジンは、恣意的に作成した問題では並列時の性能向上を示したものの、一般にはそれが全く見られない。特に 0-1 ナップサック問題の場合、単位あたりの価値の大きな荷物は最適解に含まれやすいという性質があるため、単位あたりの価値の大きな荷物から順に分枝の対象としている現状では、最適解を含みうるノードが限られてしまい、それ以外のノードは限定操作によって早い段階で計算が打ち切られることが多い。この欠点を改善するためには、1) 現行の探索方法を変える、2) 特定のノードに負荷が集中した時点でさらに負荷分散を行う機構を組み込む、というような手法、さらに 3) マスタにおいて展開するノード数の調整が効果的であると考えられる。

1) の手法には、マスタでのノード生成やワーカでのノード計算に下界値優先探索を組み込むことが挙げられる。下界値優先探索は一般に生成ノード数が少なく抑えられ、それゆえ計算時間の短縮に効果的な方法であることが知られている[13]。現在、メモリ使用量に問題があるため実装は完了していないが、この実装ができれば実行時間の飛躍的な向上が期待できる。

2) の手法の実装方法としては、処理が集中しているワーカのプールに保持されているノードを、マスタが取り出して分割して他のワーカに渡す、もしくは他のワーカにノード自体を渡す、といった方法が考えられる。しかし、特定のワーカに処理が集中していることをマスタが感知するのは容易ではあるが、どのノードを渡せば効果的に負荷分散ができるのか、ということは問題自体の性質に依存し、またそれを予想することも困難であり、渡すノードによっては通信コストなどにより逆に実行時間が増加すると想定される。これを解決することは今後の課題として残っている。

3) については、本稿での実験ではワーカ数の 4 倍に展開したが、これを増やすことで負荷の分散を均等に近付けられることも考えられる。安易に増やすと総計算ノードの増加につながりうるため、この点は、問題の性質を見ながら改善する必要がある。

## 7. 関 連 研 究

jPoP の関連研究としては、横山らによる汎用の並列

最適化ソルバである PopKern[14] があげられる。PopKern は C と KLIC で実装され、共有メモリマシン上で実行される。相違点として、階層制御を行っていない点、計算機のヘテロ性に対応していない点、実装言語の制約によりポータビリティが低い点などがある。

Condor project[15] の MW[16] はグリッド上でのマスタ・ワーカ形式のアプリケーションを容易に実行するためのソフトウェア・フレームワークであり、Condor をリソース管理に使用している。これを用いて、最適化アプリケーションを実装することは可能だが、ユーザは最適化アルゴリズムを一から実装しなければならないので負担が大きい。

また、階層制御に関しては、夏目らによる Ninf システムを用いたグリッド上での最適化アプリケーション[3] が挙げられる。これは階層的なマスタワーカ方式を採用しており、マスタが負担する通信量を削減することにより性能向上を実現している。

## 8. まとめと今後の課題

本稿では、階層型分散実行環境 Jojo を用いた組合せ最適化問題を容易に解くための最適化システム jPoP のうちの分枝限定法用クラス群 jPoP-BB について、プロトタイプの実装を行いその性能を評価した。その結果、他のノードの暫定解による限定操作が生じない特別な問題に対しては並列効果が得られることを確認したが、一般の問題に対しては負荷の偏りにより並列効果が得られないことが確認された。

jPoP は最終的には、グリッドにおけるより大規模な計算機環境においても効果的に並列処理が行えるシステムに移行していくことを目標としており、本稿で述べたプロトタイプの実装手法はそのための原形であってまだ問題点を多く残している。今後は以下の点を実現させることを目指す。

- 6 節で述べた手法を用いるなどして並列処理による実行時間を短縮する
- スケーラビリティに問題が生じることが予測されるため、階層的なマスタ・ワーカ構造の導入を進める

## 参 考 文 献


1) 長尾智晴. 最適化アルゴリズム. 昭晃堂, 2000.
2) Ninf Project Home Page. http://ninf.apgrid.org.
3) 夏目亘, 合田憲人, 二方克昌. 階層的マスタワーカ方式による BMI 固有値問題の Grid 計算. 情報処理学会研究報告 HPC-2002-91, pp. 73–78, August 2002.
4) A. Takeda, K. Fujisawa, and M. Kojima. Enumeration of All Solution of a Combinational Linear Inequality System Arising from the Polyhedral Homotopy Continuation Method. *Journal of the Operations Reaserch Society of Japan*, Vol. 45, No. 1, pp. 64 – 82, 2002.
5) 秋山智宏, 中田秀基, 松岡聡, 関口智嗣. Grid 環境に適した並列組み合わせ最適化システムの提案. 情報処理学会研究報告 HPC-2002-91, pp. 143–148, August 2002.
6) R. Tanese. Distributed Genetic Algorithms. In *Proc.3rd International Conference on Genetic Algorithms*, pp. 434 – 439, 1989.
7) C. Roucariro. Parallel branch and bound algorithms - an overview. In M.Cosnard et al, editor, *Parallel and Distributed Algorithms*, pp. 153 – 164, North-Holland, 1989.
8) K. Kimura and K. Taki. Time-homogeneous parallel annealing algorithm. Technical Report 673, ICOT, 1991.
9) The Globus Project. http://www.globus.org.
10) Grid Security Infrastructure. http://www.globus.org/Security/.
11) 中田秀基, 松岡聡, 関口智嗣. グリッド環境に適した Java 用階層型実行環境 Jojo の設計と実装. 情報処理学会研究報告 HPC-2002-92, pp. 31–36, Octorber 2002.
12) 品野勇治, 檜垣正浩, 平林隆一. 並列分枝限定法における解の探索規則. 計測自動制御学会論文集, Vol. 32, No. 9, pp. 1379–1387, 1996.
13) 茨木俊秀. 組合せ最適化 分枝限定法を中心として. 産業図書, 1983.
14) 横山大作, 近山隆. 高度な問題領域依存チューニングを許す並列組合せ最適化ライブラリ PopKern. 情報処理学会論文誌:プログラミング, Vol. 41, No. SIG3 (PRO10), pp. 49 – 64, Mar 2001.
15) Condor Project Homepage. http://www.cs.wisc/condor/.
16) J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yorde. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pp. 43 – 50, Pittsuburgh, Pennsylvania, August 2000.