グリッド環境に適した並列組み合わせ最適化システム jPoP における分子限定法の実装

秋山智宏[†] 中田秀基[††,†]
松岡聡[†,†††] 関口智嗣[††]

多次元パラメータ関数の最適値を求める組み合わ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。これらの解法は自明な並列度が大きく、粒度の調整も比較的容易なためグリッド上での実行に適している。しかし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。われわれ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。jPoP は各解法に対してテンプレートとなるクラスを提供しプログラミングを支援する。また、動的なプログラムのアップロードによってグリッド上での実行を支援する。現在、最適化アルゴリズムのひとつである遺伝的アルゴリズム用クラス群 jPoP-GA が実装されているが、本稿ではこれに加えて、分枝限定法のテンプレートクラスの設計と実装について述べる。

Parallel Combinational Optimization System for the Grid: jPoP

Tomohiro Akiyama ,[†] Hidemoto Nakada ,[††,†]
Satoshi Matsuoka [†,†††] and Satoshi Sekiguchi[††]

For combinatorial optimization problems, which compute the optimal value of a multi-dimensional parameter function, serveral methods are known to be effective, such as Branch-and-Bound methods, Genetic Algorithm, etc. Since these methods can be massively parallelized and the granularities of computation tasks are easily controllable, they are considered to be suitable for executing on the Grid. However, 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. To support the development, the jPoP provides a template class for each algorithm. And to reduce the cost of the setup, it automatically stages the user programs to the Grid environment. In this paper, we foucus on the Branch-and-Bound method and describe design and implementation in detail.

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、組み合わせ最適化問題[1] に注目している。組み合わ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 Grid 技術の有効性を確認してきた[3),4)]。

しかし、一般的に Grid 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)Grid 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、組み合わせ最適化アプリケーションでは、4) 組み合わ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 Grid 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、Grid 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。

† 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

この問題を解決するために、我々は並列組み合わせ最適化システム jPoP[5] を提案している。jPoP は、代表的な数種の並列組み合わせ最適化アルゴリズムのテンプレートを提供し、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、グリッド上で組み合わ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ひとつである分枝限定法用クラス群 jPoP-BB の設計と実装について述べる。

2. 組み合わせ最適化問題

以下では組み合わせ最適化問題の概要について述べる。組み合わせ最適化問題は数理計画問題の一つの分野であり、「与えられた制約条件のもとで、最大の評価が得られるような組み合わせを求める」という問題である。

有名な問題としては巡回セールスマン問題 (TSP) や 2 次割当問題 (QAP)、ナップザック問題などがある[1]。組み合わせ最適化問題は、スケジューリング、設計問題や生産計画などの実社会において広い応用範囲を持っているが、実社会で求められる大規模な問題を解くには膨大な計算量が必要である。しかし、組み合わせ最適化問題の多くが独立性の高い部分問題に分割しやすいという性質を持つため、並列化によって高速化及び適用問題の大規模化が期待できる。多くのアルゴリズムの中で我々は 3 つの最適化アルゴリズムに注目してる。以下ではそのアルゴリズムと並列化手法について概要を述べる。

遺伝的アルゴリズム (GA)　GA は生物の進化プロセスから着想された多点探索に基づく探索アルゴリズムであり、自然淘汰・交差・突然変異等の特徴的な操作を用いて新しい個体 (探索点) を生成し、実用解あるいは最適解を高速に発見する手法である。並列化手法として、一つの母集団を複数の集団に分割し、複数の PE(Processing Element) に割り当てる分散 GA[6] がある。

分枝限定法 (Branch and Bound)　分枝限定法は問題を複数の部分問題に分割し (分枝操作)、部分問題の中に最適解が見つかるかもしくは最適解が存在しないことを判定し、それ以降の解の探索を取り止める (限定操作) ことで解の探索効率を挙げる手法である。

並列分枝限定法[7] では分割した部分問題を複数 PE に割当て、各々で探索と分枝操作、限定操作を行う。

焼き鈍し法 (SA)　焼き鈍し法は確率的に解候補を改善していく手法である。この手法は、暫定解近傍を調べてより評価値の良い側へ暫定解を変化させる操作を繰り返すことで良質な解を求める。さらに、解の評価値が悪くなる方への移行を確率的に認め、温度という状態遷移確率を制御する変数を高い方から低い方へ徐々に変化させることで最適解を求める。これを複数の異なる温度を別々の PE で並列に行い、隣り合う温度間で定期的に解交換を行うのが温度並列焼き鈍し法[8](レプリカ交換法) である。

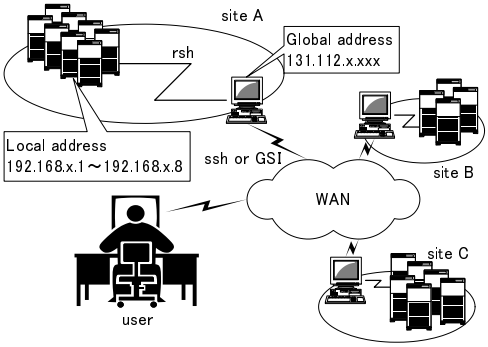

図 1　jPoP の実行環境

3. jPoP の設計

jPoP は実行環境としてクラスタオブクラスタのようなグリッド環境を想定している (**図 1**)。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。このようなクラスタの各ノードの IP アドレスはセキュリティやアドレス空間枯渇の問題から、ローカルアドレスとなり、NAT 等を用いて外部と通信する場合が多い。それぞれのクラスタには、グローバルなアドレスを持ち外部と通信が出来るノード (マスタ・ノード) が少なくとも一つあるとし、各クラスタの内部のノードはこれを通して外部と通信を行う。

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、組み合わ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み合わ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。☆

☆ 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

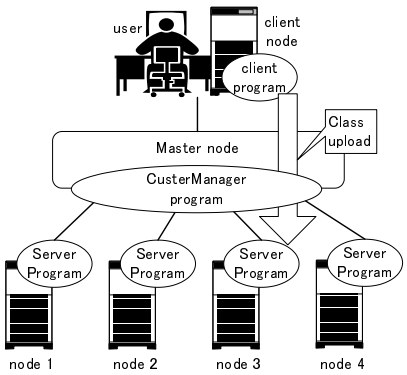

図 2 jPoP のシステム構成

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。

3.1 プラットフォーム独立性

jPoP は実装に Java を用いることで、さまざまなプラットフォーム上での実行を可能にする。クラスタの各ノード上には Server、マスタノード上には ClusterManager と呼ばれる Java プログラムが存在する。ユーザは、やはり Java で書かれたクライアントプログラムを自分の記述したクラスを引数として起動する。クライアントは ClusterManager を介して Server と通信し、ユーザが記述したクラスを各ノードに自動的にアップロードする (**図 2**)。

3.2 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、異なるサイトのマスタ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。これにより、通信路の暗号化やサイト間のセキュリティポリシーの違いといった問題を解決できる。

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。jPoP の場合は各サイトに認証されているユーザのコードのみを実行可能とするため、悪意のあるユーザコードが実行される危険性はないと仮定できる。しかし、プログラムのバグから被害を与える可能性もあり注意が必要である。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。

3.3 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ラ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。

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

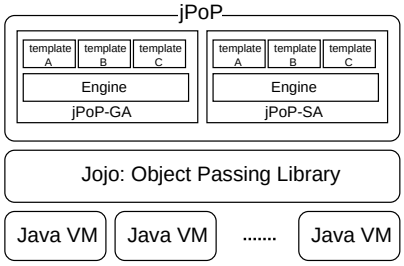

図 3 jPoP レイヤー

3.4 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なか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。

4. 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 3 に Jojo を下位レイヤとした jPoP の全体レイヤを示す。以下で Jojo の概要について述べる

4.1 Jojo の概要

Jojo は中小規模のクラスタが分散して存在する環境を対象とし、以下の点を考慮して設計されている。

- グリッドの階層構造を反映したシステム構造
- スレッドを前提とした柔軟で簡潔なプログラミングモデル
- 動的なシステム構成を可能にするとともに、インストールの手間を最小限にする起動方法

4.1.1 システム構造

Jojo は階層的なグリッドをターゲットとし、階層構

```
abstract class Code{
  Node [] neighbors;  /*兄弟ノード*/
  Node [] lowers;     /*子ノード*/
  Node [] upper;      /*親ノード*/
  int rank;           /*兄弟の中のでの順位*/
  public abstract void init(String [] args);
  public abstract void start();
  pubiic abstract Object handle(Message meg);
}
```

図 4 Code クラス

造を意識したシステム構造をとる。Jojo はクライアントを頂点とする任意段数の階層構造を持つシステムを構成し、それぞれのノードで任意のプログラムを実行することができる。個々のノードは自分と同レベルのノード群だけでなく、上位レベルのノード、下位レベルのノード群とも通信することができる。

4.1.2 プログラミングモデル

java を用いた並列通信ライブラリとしては、RMI や HORB などの並列オブジェクトに基づくモデルと、MPI や PVM の流れを汲む SPMD 基盤モデルがあり、Jojo はこの両者の中間ともいうべきプログラミングモデルを提供する。

Jojo では各ノードにひとつのオブジェクトを配置し、そのオブジェクト間でのメッセージパッシング機構を提供する。メッセージパッシング機構としては、オブジェクト単位の送信と受信を行う。送信は明示的に行うが、受信はハンドラを定義することで行う。また、送信に対する返答を受け取る通信パターンに対しては、通信に対する戻り値を返す機構を設けてある。これはブロックを行う同期呼び出しに加えて、Future オブジェクトを使用する機構とコールバックオブジェクトを登録する機構が用意されている。マルチスレッドを前提として、メッセージの受信を別スレッドで行うセマンティクスになっているため、メッセージの受信と処理を重複させるコーディングがごく自然に行うことが可能となっている。

4.1.3 起 動 方 法

Jojo は大域での分散実行を指向しているため、全てのノードが NFS でファイルシステムを共有していることを期待することはできない。しかし、ユーザがコードをすべてのノードにアップロードするのは煩雑である。 Jojo では全てのユーザプログラムが自動的にクライアントからダウンロードして実行される。

さらに Jojo 自体も自動的にダウンロードされて実行される。これによって実行する Jojo のバージョンがノードによって異なる、というような事態を未然に防ぐことができる。

4.1.4 API

ユーザのプログラムには**図 4** の Code と呼ばれるクラスを実装する。各階層用に複数のコードを用意する。 init メソッドが初期化を行い、init メソッドの終了以前に start メソッドや handle メソッドが呼び出されることはない。start メソッドが本体の処理を行い、ハンドラが送信されてきたオブジェクトの処理を行う。ハンドラ (handle メソッド) を実行するスレッドは、オブジェクトを受信する毎に新たに起動されるので、ハンドラの中で長大な処理を行っても他のオブジェクトの受信に影響はでない。

4.2 jPoP の実装

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。現在、遺伝的アルゴリズム用クラス群 jPoP-GA が実装されており、ユーザは遺伝的アルゴリズムのコアとなる個体と評価環境をクラスとして定義するだけで、分散環境上で遺伝的アルゴリズムを用いた並列最適化アプリケーションを実装、実行可能となっている。

以下では、今回新たに実装した分枝限定法用のクラス群 jPoP-BB について述べる。

5. 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が行うことができる。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。

5.1 分枝限定法の並列化

分枝限定法の一般的な処理手順を**図 5** に示す。jPoP-BB では分枝限定法を並列化するにはマスタ・ワーカ方式を用いる。この場合、マスタはノードの管理を行い、選択したノードを複数のワーカに対して割り振る。ワーカは実行可能解の計算、下界値評価をした後に暫定解の更新を経て、分枝操作や限定操作を行う。分枝操作によって新しくノードが分割された場合はマスタにそれを返し、その時点でノードが割り振られていないワーカに対して再分配することによって並列化される。ここで留意しなければならない点は、対象となる問題によってボトルネックとなる処理が異なるということである。この部分は問題依存領域であり、下界値計算、実行可能解の計算、分枝操作にコストがかかる場合などがある。従って、jPoP-BB ではリモート側、つ

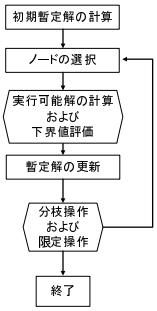

図 5 分枝限定法の処理手順

```
   /*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 6 BBNode オブジェクト

まりワーカで実行する部分をシステム下部のドライバと呼ばれる機構で自由に変更できるよう設計されている。これによって、ユーザが記述する部分ではどの部分が並列実行されるかを全く意識せずに記述することができる。

5.2 jPoP-BB の設計

ユーザが分枝限定法を記述する際には、以下の項目を記述する。

- ノード (BBNode)
- ノードを計算する環境 (Environment)
- 指定した探索法で次ノードの選択を行うプール (Pool)

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、上記に示された項目を定義するだけ分散環境で分枝限定法を用いて組み合わ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。

5.2.1 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 6** 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。

5.2.2 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 7** 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が並列実行の場合、リモート側であるワーカで実行される。

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

図 7 Environment インターフェース

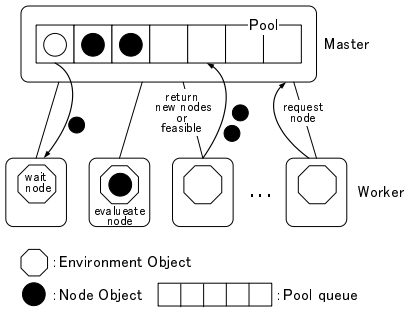

図 8 jPoP-BB の概要

5.2.3 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、3) 最良下界探索を Pool サブクラスを提供するので、必ずしもユーザが実装する必要はないが、独自の探索法を定義することも可能である。

5.3 jPoP-BB の実行

jPoP-BB はマスタ・ワーカ方式 (**図 8**) で実装されており、以下のようにおこなわれる。

(1) マスタが探索のルートとなるノードを作成
(2) マスタはルートノードを自身が保持するプールに与える
(3) 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
(4) ワーカからの要求によりプールからノードが取り出され、ワーカで評価が行われる。
(5) ワーカは実行可能解がある場合はそれをマスターに返し、暫定解の更新が行われる。また、分枝操作が行われた場合は新たに生成されたノードをマスタのプールに与える。
(6) (1)〜(5) を繰り返し、マスターのプール内と、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、マスターがもつ暫定解が最適解となる。

5.4 定 義 例

0-1 ナップザック問題を最適化するためのノードクラス (**図 9**) と環境クラス (**図 10**) 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

```
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 9 BBNode クラスの定義例

```
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 10 Environment クラスの定義例

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。

6. 性能評価

性能評価として、定義例で示した 0-1 ナップザック問題を幅優先探索を用いて解くアプリケーションを**図 11** に示す環境で実行した。TITECH 内のマスタ

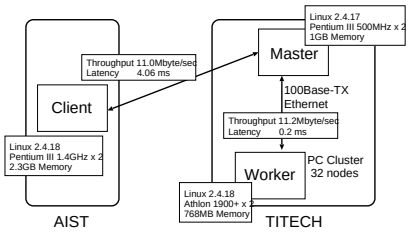

図 11 評価環境

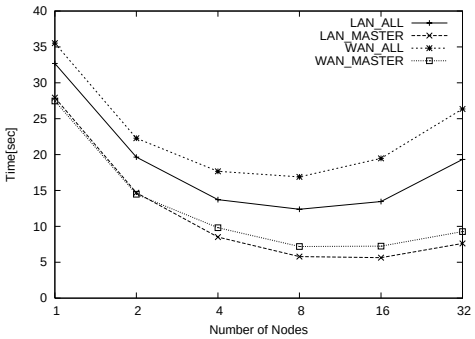

図 12 WAN と LAN の実行結果

ノードでクライアントを起動し、クライアントと同じマシンにマスタを起動し、各クラスタノード上にワーカをアップロードする実験を LAN とし、AIST 上でクライアントを起動し、TITECH 内のグローバルなアドレスを持つマシンにマスタをアップロード、それを介して PC クラスタの各ノードにワーカをアップロードする実験を WAN とする。また、AIST-TITECH 間は ssh、TITECH 内のマスタ・ワーカ間は rsh を用いた通信を行い、PC クラスタを含めた全てのマシンには Sun JDK 1.4.1_01 がインストールされている。

図 12 に LAN 及び WAN 環境での結果を示す。ここで用いた 0-1 ナップザック問題は荷物数が 100 である。LAN_ALL、WAN_ALL はそれぞれの環境でのアプリケーション全体の実行時間であり、LAN_MASTER、WAN_MASTER はそれぞれの環境でのマスタが起動してから終了するまでの時間である。ここでは、マスタの実行時間が実際にナップザック問題を解く時間であり、全実行時間からマスタの起動時間を省いた時間は 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、終了したりするオーバーヘッドとなっている。

LAN と WAN を比較した場合、マスタの実行時間のみを比較した場合はほとんどその差はない。しかし、全体の実行時間を比較した場合、WAN の方がオーバーヘッドが大きくなっている。これは、ワーカをアップロードする際にクライアントからマスタを介して 1 つずつアップロードをするため、WAN のようなクライアントとマスタ間が離れているとそれだけアップロードに時間がかかることが原因であると考えられる。こ

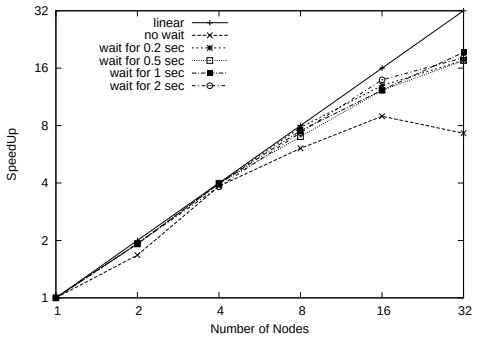

図 13 Wait の違いによる性能向上

れは、マスタ上にワーカのプログラムを一時的にキャッシュすることで回避されると考えられる。また、ワーカの数が多くなるほどオーバーヘッドも増加しているが、これも同様の原因であると考えられる。しかし、これらのオーバーヘッドはマスタの実行時間、つまり最適化問題を解く時間自体が十分長いものであれば無視できる値であると考えられる。

ワーカが増加することによる性能向上は、2〜8 台までは見られるが 16、32 台では逆に性能低下している。この原因としてはワーカ側での計算粒度が小さいことが考えられる。計算粒度が小さい場合、ワーカがマスタからノードを受け取る、評価する、結果を返す、という一連の処理がごく僅かな時間で行われる。これが複数台のワーカでほぼ同時に行われるので、マスタへの負荷が集中してマスタはその処理が終了するまで、新たにワーカにノードを割り振れず、ワーカがアイドル状態になってしまうため、並列性が失われてしまっていると考えられる。

7. 考　　察

前節でも述べたように、jPoP は 2〜8 台では期待された性能向上を示している。しかし、それを越えるような規模になった場合に台数に比例した性能向上は見られない。これは対象となる問題の計算粒度が小さいためであると考えられる。では、jPoP はどの程度の計算粒度の大きさであれば、台数に比例した性能向上を得られだろうか。以下では、対象となる問題の粒度を任意の値に指定し、jPoP で並列化するのに適した計算粒度について議論し、大規模環境においても期待された性能向上を示すための方法について述べる。

計算粒度の大きさを変えるため、性能評価で用いた 0-1 ナップザック問題において、ワーカ側で下界値評価を行った後、こちら側で指定した時間だけ wait するようにした。wait が大きければ、それだけ計算粒度も大きいと仮定できる。LAN 環境において、wait する時間を 0.2〜2.0 [sec] まで変化させて 0-1 ナップザック問題を解いた結果を**図 13** に示す。ここでは問題における荷物数は 30 としている。図の縦軸は 1 台で実行した時と比較したものであり、1〜16 台ではほぼ理想的な性能向上を示しているのがわかる。32 台ではやや性能向上が低下しているがこれは対象となる問題の並列性に限界が生じたためと考えられる。

結果から jPoP においては対象となる問題の計算粒度が一定の大きさでなければ、規模が大きくなった場合に期待される性能向上が得られないことが考えられる。つまり、対象となる問題をノードに分割する際に計算粒度を大きくする必要が生じる。しかし、問題によってはその性質上、計算粒度を調整できない場合もある。この場合は、ワーカに対して複数のノードを一度に送信し、全ての処理が終了するまでマスタに結果を送信しないようにすることで、見かけの計算粒度を大きくすることで対処できる。しかし、この手法によるワーカへのノードの分配法は計算粒度を大きくすることでマスタの負荷を軽減する一方で、暫定解や下界値の伝達が遅れるといった欠点も生じる。暫定解や下界値の伝達が遅れると、その値によるノードの枝刈りが行われるのが遅れるということであり、これによって本来は枝刈りされて探索する必要のないノードを探索してしまうことが生じる可能性が高い。また、プログラマが問題によって、適切な計算粒度を調整するという負担もあり、この手法は現実的ではないと考えられる。

計算粒度を大きくすることによる性能向上には限界があると考えられるため、単一のマスタが全てワーカを管理するという手法を根本的に見直す必要があると考えられる。つまり、マスタを複数に分割し、1 つのマスタが管理するワーカの数を減らすことが提案される。各マスタ間で暫定値や下界値の伝達を行うことで、効率の良い枝刈りが妨げられることはなくなる。また、1 つのマスタにかかる負荷が性能向上を妨げることもなくなると考えられる。また、プログラマが計算粒度を意識する必要もない。この手法は現在の jPoP の実装とは大きく異なっているが、下位レイヤとして階層構造を考慮した通信ライブラリである Jojo を用いているので、同じ階層にある複数のマスタ間の通信の実現などは可能であると考えられる。いずれの手法をとるにしても、上記に挙げたプログラミングモデルをプログラマから隠蔽するということが重要であると考えられる。

8. 関 連 研 究

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[12] 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[13] の MW[14] は Grid 上でのマスター・ワーカー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。

9. まとめと今後の課題

本稿では、階層型分散実行環境 Jojo を用いた組み合わ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 について設計、実装を行った。今後の課題として、焼き鈍し法のアルゴリズムについても上記で述べたアルゴリズム同様、分散環境上で実行するための枠組を設計、実装をする必要がある。

また、それぞれのアルゴリズムについて階層的な制御を行うことを目指す。現在は、単純なマスタ・ワーカ方式による実装のみであるが、分枝限定法はツリー構造の検索ツリーを状況に応じて枝刈りするので、検索ツリーの階層性を制御の階層にマップできる。これを利用することで階層的な制御が期待できると考えられる。

謝　　辞

貴重な助言、御討論をくださった東京大学の横山 大作氏、京都大学の藤沢 克樹氏、徳島大学の小野 功氏、東京工業大学の小島 政和氏ならびに小島研究室の方々、そして Ninf プロジェクトの方々に深く感謝致します。

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) 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.
13) Condor Project Homepage. http://www.cs.wisc/condor/.
14) 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.