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[†] 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、本実装は大規模なシステム上でも実行でき、かつ一般的な問題のほとんどに対して実行時間の面から十分な並列効果が得られることがわかった。

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

Shingo Nakagawa ,[†] 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. We could execute the system on the large-scale computing environment, and it could show remarkable speedup for general problems.

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。

上記に挙げたうちの 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、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、最適化問題を解くためにアプリケーションを実装する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トを提供し、プログラムの並列化、安全性などの問題

† 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

をプログラマから隠蔽する機構をもつ。プログラマは問題に依存する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。

我々は、この jPoP が提供するアルゴリズムの一つとして、分枝限定法用のクラス群 jPoP-BB を設計し、そのプロトタイプの実装を行った。本稿では、まず jPoP の概要を述べ、その上で jPoP-BB の設計、プロトタイプ実装、およびその性能評価について述べる。

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際には、

- 並列度のスケーラビリティの獲得が困難

という問題が生じる。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一台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ず実行可能となっている。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[6] の GSI(Grid Security Infrastructure)[7] 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

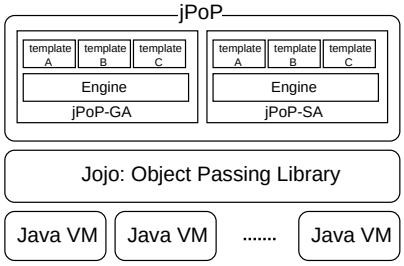

図 1 jPoP レイヤ

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。

2.4 アルゴリズム実装支援

jPoP は、内部に様々な並列組合せ最適化アルゴリズムをもっている。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し、使用するアルゴリズムを指定することで、組合せ最適化アプリケーションを構築できる。しかし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、一つの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しており (jPoP におけるその例は 4 節に示す)、ユーザはこれらを問題に即した形で定義することでプログラミングを行う。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り、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

前節で提案した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[8] 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。**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

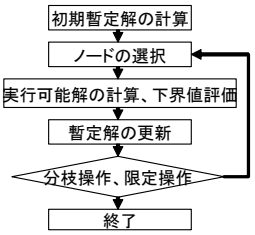

図 2 分枝限定法の処理手順

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。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、使用するエンジンを選択す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 の一つとして、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 に示す手順はエンジン部に実装されているため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。ノードは生成される部分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、ユーザはこの部分に一切記述を行うことなく、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、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
(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

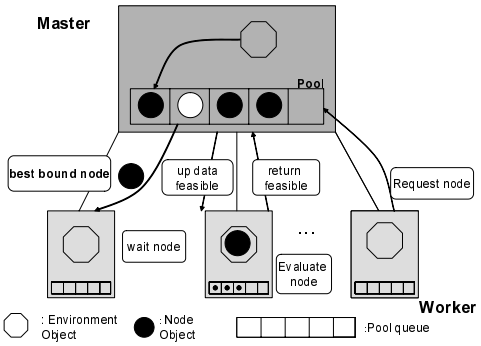

図 5 jPoP-BB プロトタイプの概要

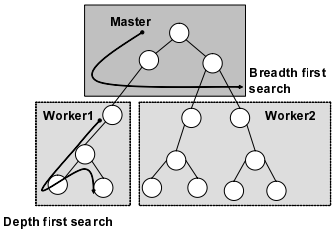

図 6 jPoP-BB プロトタイプにおける探索の手順

(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、その評価結果をマスタに通知して (5) に戻る

(10) また、ワーカが与えられたノードの評価を一定の範囲まで進めても評価が終了しない場合、その時点でワーカのプールが保持するノードの一部をマスタのプールに返す

(11) (5)〜(10) 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。

また、(10) に記す機構は、(4) において生成されるノードの間に計算負荷の大きな偏りがある場合を想定して導入した負荷分散機構である。この手順における、ワーカが評価を進める範囲、およびワーカがマスタに返すノードの数については、ユーザが実行時に指定する。この範囲を小さくした場合、またはこのノードの数を多くした場合、負荷のより均等な分散が期待されるが、通信量や通信回数、メモリ使用量が増えるという問題が生じるため、問題の性質に応じて適切な値を指定しなければならない。

```
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。したがって、ユーザが並列処理構造について記述する必要は一切ない。

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。これは、分枝限定法においては、逐次実行時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、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の影響により、計算量そのものが不規則に変化することに起因する。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。

```
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

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
- 各荷物の価値の最大格差 (=最大値/最小値) 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%

なお、各荷物の価値・重量は、上記の格差の範囲内でランダムに設定している。本稿では、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 問を生成して実験対象とした。

実行に用いたクラスタは、CPU が Athlon MP 1900+(1.6GHz)、メモリ 768MB である、256 台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** に示すように、全体の傾向を

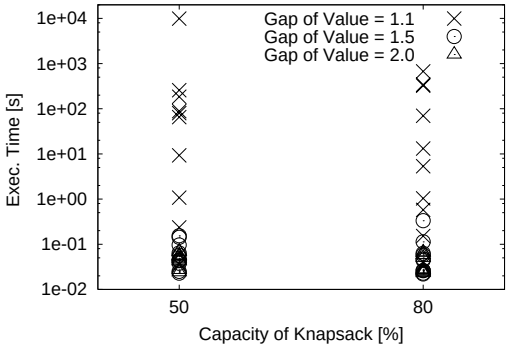

図 9 ナップサックの大きさの違いによる実行時間の変化 (荷物 100 個)

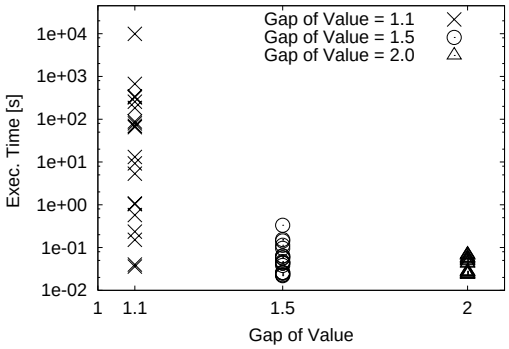

図 10 荷物の価値の最大格差の違いによる実行時間の変化 (荷物 100 個)

分析することは不可能であった。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がある一方、逆に大きく減少する問題もあった。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 10** に示すように、最大格差を 1.1 倍とした問題だけに、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 (最大でも 2.75 倍) ため、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く、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 2 倍の問題の方が実行時間が短くなっているが、その逆の挙動を示した問題もあった。これは単位あたりの価値にある程度のばらつきがあ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 11** 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

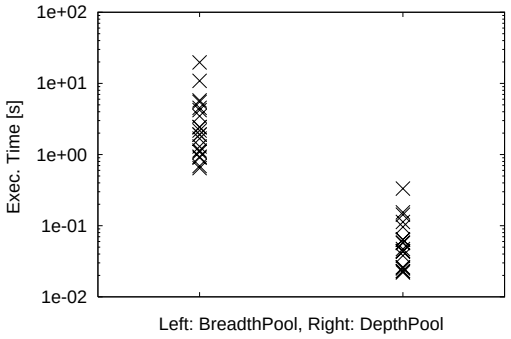

図 11 幅優先探索 (左) と深さ優先探索 (右) による実行時間

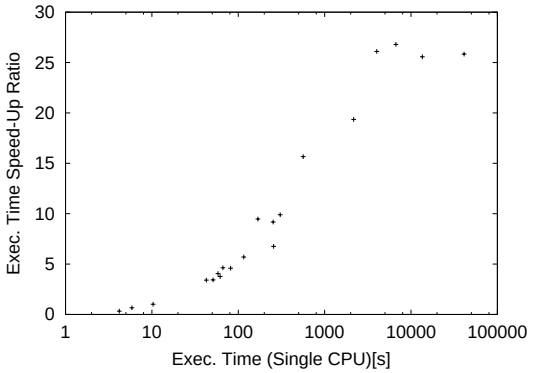

図 12 SingleCPU とワーカ 32 台使用時の実行時間の関係

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[9]。☆

よって以下では、深さ優先探索での逐次実行における実行時間を基準とし、これを並列実行によって短縮することを考える。

5.2 逐次実行と並列実行の比較

前項で解いた 120 の問題のうち逐次での実行時間が 4 秒から 12 時間までの範囲にある 20 問を抽出し、並列実行による実行時間短縮効果を調べた。

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 5 回ずつ行い、その実行時間の平均をとって逐次実行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する時間や終了処理の時間は含まない。

結果の概要を**図 12** に示す。ワーカ 32 台使用時に

☆ 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

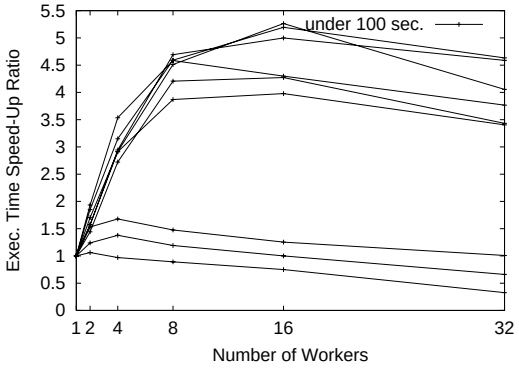

図 13 ワーカ台数の変化による実行時間の変化
(逐次での実行時間が 100 秒未満の 9 問)

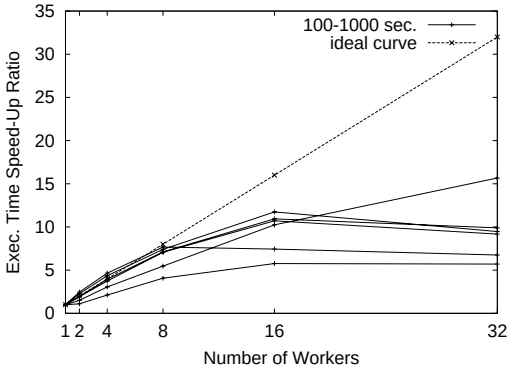

図 14 ワーカ台数の変化による実行時間の変化
(逐次での実行時間が 100 秒以上 1000 秒未満の 6 問)

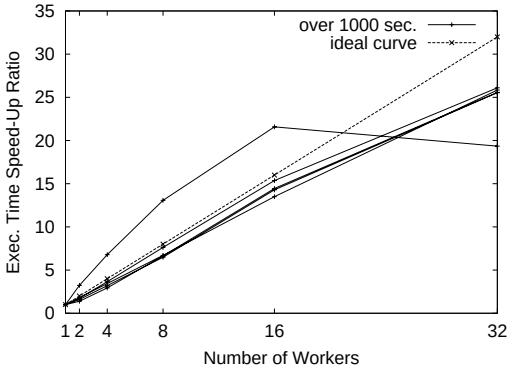

図 15 ワーカ台数の変化による実行時間の変化
(逐次での実行時間が 1000 秒以上の 5 問)

逐次実行時より時間がかかった問題が 20 問中 2 問あるが、逐次実行時の実行時間が長い問題になるほど並列効果が上がり、一定規模以上の問題においてはおよそ 25 倍前後の並列効果が得られている。

詳細を**図 13**〜**図 15** に示す。図 13 に示すような、逐次での実行時間が短い問題においては、並列効果が

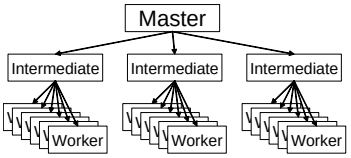

図 16 多階層通信エンジンの通信構造

ほとんどなく、ワーカ台数の増加とともに実行時間が長くなるケースもある。これは、ワーカでの計算時間がほとんど変化しないのに対し、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が伸びたためと考えられるが、これらの問題を並列処理で解く必要性は小さいため、このことは実用上問題にはならないと考える。一方、**図 14**、図 15 に示すような、逐次での実行時間が長い問題においては、概してよい並列効果が見られる。異常加速が起こっている問題も 1 問あるが、これは、分枝方法の違いにより計算量の総和が変化することに起因している。

これらの図より、大規模な問題を解くにあたって並列効果という観点では jPoP は十分に実用的であることがいえる。ただし、図 15 に示す 5 問の中には、メモリ使用量が大きくなりすぎたためにワーカ 2 台、4 台使用時の実験に失敗した問題が 1 問あり、メモリ使用量がさらに大規模な問題を解くにあたっての障害となることが考えられる。

5.3 多階層通信プロトタイプ実装への改良

前項で使用したプロトタイプエンジンには、ワーカ使用台数が 32 台を超えるとマスタに通信負荷がかかり過ぎる、という欠点がある。これを解決する手法として、我々は階層的な通信構造の導入を考えた。これは、従来マスタ・ワーカ間に限られていた通信を、ワーカ同士で、または、ワーカがさらにワーカを呼び出して呼び出した側のワーカをマスタとするなどして行うように拡張する手法で、以下の効果が期待できる。

- 使用ワーカ台数を増加させたときのマスタの負荷を軽減
- 実行環境を現在のクラスタからグリッドに拡張して地理的に離れているノードを用いて計算を行う際の、遠距離間での通信量の抑制

そこで我々は、前者の効果を得るため、ワーカがワーカを呼び出して、呼び出した側のワーカを中継マシンとする、マスタ・中継マシン・ワーカの 3 階層による多階層通信に対応したエンジンを実装した。このエンジンでは、**図 16** に示すように、通信はマスタ・中継マシン間、中継マシン・ワーカ間のみで行われ、ワーカから送られた暫定解などのデータは全て中継マシンを経由してマスタに送られる。

この多階層通信プロトタイプエンジンを用いて、図 15 に示す 5 問を、クラスタ内でワーカ 64 台、80 台、96 台、112 台、128 台を使用して解く実験を行った。(中継マシンはワーカ 16 台ごとに 1 台設けた。)

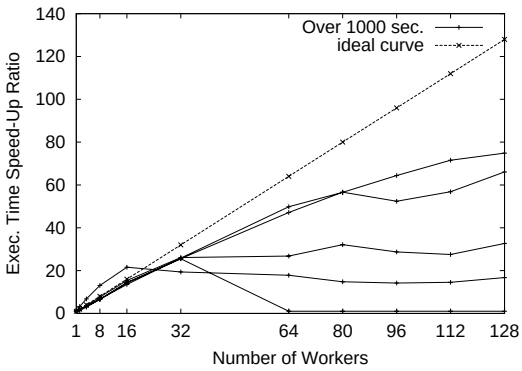

図 17 多階層通信エンジン使用時の実行時間の変化
(ワーカ 32 台以下は従来型エンジンでの実行時間)

その結果を**図 17** に示す。5 問のうち 1 問は何らかの理由でほとんど並列効果が得られず、また 2 問はワーカ 32 台使用時とほぼ同等の並列効果にとどまったが、5 問のうち最も逐次実行時の実行時間が長い 2 問に関しては、ワーカ 128 台使用時で 70 倍前後の並列効果を得ることができた。これらの問題では、ワーカ台数が増えるにしたがって負荷の不均等の度合が増していることが確認されており、これを改善すれば、さらなる並列効果の獲得が見込める。

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、小規模の問題に対してはほとんど並列効果を示さなかったものの、並列処理の必要性の高い、大規模な問題に対しては、小規模な環境では十分な並列効果を得ることができた。4.3 項の実行手順 (10) に記した機構を実装する以前は性能向上比はワーカ 32 台使用時でも高々2 倍にとどまっており、これより、この機構が負荷分散に重要な役割を果たしていると考えられる。これは、0-1 ナップサック問題の性質上、この機構を使わずに単純に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た場合、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に起因すると考えられる。

しかし、現在のエンジンには、1) 大規模な問題を少ないワーカ数で解くことができない、2) 大規模環境での並列効果に限界がある、という、改善の余地を残す点がある。1) については、マスタのプールにワーカから返却されるノードが膨大な数にのぼり、マスタのメモリ使用量が莫大なものになることに起因すると考えられる。具体的な解決手法としては、返却するノードの数自体を減らすことや、返却のタイミングを調整してマスタ内に同時に存在するノードの量を減らすことによるメモリ使用量の抑制が挙げられるが、前者は負

荷の大きな偏り、後者は計算時間の遅延につながるため、問題の性質を見ながら慎重に扱わねばならない。

また、2) について、本稿で対象とした問題のうち最大のものは、使用ワーカ数を増やしても負荷の不均等により実行時間が最短でも 500 秒程度になっている。解決手法としては、5.3 項で述べた多階層通信エンジンを改良して、中間マシンにマスタの役割を持たせる、すなわちワーカからのデータをマスタに送らず中間マシンで止めて処理することで、通信量の削減とさらなる負荷分散を図ることが考えられる。

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[10] 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[11] の MW[12] 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、並列処理の必要性の高い、比較的大規模な問題に対しては、小規模な環境ではほぼ均等な負荷分散が行われ、十分な並列効果が得られることを確認した。また多階層通信エンジンを実装することにより、ワーカを 128 台使用する大規模な環境においても、現実に現われる問題により近い大規模な問題に対しては安定した動作と並列効果を得られることも確認できた。

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残している。今後は、6 節で述べたように、メモリ使用量の面からのシステムの改善を図るとともに、多階層通信エンジンの通信構造を見直して、実行時間全体のスピードアップ、大規模環境でのさらなる並列効果の獲得を目指していく。

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) The globus project. http://www.globus.org.
7) Grid security infrastructure. http://www.globus.org/Security/.
8) 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.
9) 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.
10) 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.
11) Condor project homepage. http://www.cs.wisc/condor/.
12) 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.