MapReduce による列挙木探索手法の提案と評価

中川　遼
大阪大学基礎工学部情報科学科ソフトウェア科学コース

概要　列挙木探索は，木で表せるような状態空間の中にある「解」を探すことを目的とした探索手法である．しかしながら，状態空間探索においては状態空間は入力に対して爆発的に増加する場合が多く，探索には膨大な時間的・空間的資源が必要となる．本報告ではこの探索を，大規模なデータを分散処理することの出来る MapReduce を用いて行うことを考える．MapReduce は，BigData の重要性が注目を浴びている現在，代表的な分散処理フレームワークとして注目されている．しかし MapReduce を素朴に状態空間の探索に適応すると，MapReduce は並列分散実行中はデータのリアルタイムでの受け渡しが困難なため，マシン間での限定操作が困難である．本報告ではこのように MapReduce には不向きだと考えられる状態空間の探索を，できるだけ手軽で少しでも効率的に MapReduce で行う手法を提案し，具体的な問題に対して提案手法のシステムを実装した上で提案手法の評価を行う．

1　はじめに

状態空間の探索 [?] は，ある問題において“ 条件を満たす様々な状態 (State) の中から最適な解を探索する手法 ”であり，組み合わせ最適化問題や人工知能などの分野で幅広く用いられている．しかし多くの場合，問題から生成される様々な状態の数は入力に対して爆発的に増加する [?] ため，全ての状態を探索することは膨大な時間的・空間的資源を必要とする．したがって，状態空間を効率的に探索することは重要な課題である．

本報告では状態空間の探索の中でも最も一般的である，状態をツリー状に列挙した列挙木の探索問題について議論する．

列挙木の探索は，一般的に木の大きさ（状態数）が大きくなるとその分探索にも時間がかかるので，莫大な数の状態を素早く探索するため，多数のマシンを用いた並列分散処理が考えられる．本報告で利用する分散処理フレームワークである MapReduce [?] は，通信量の増加や通信されるデータの大規模化が進む近年において，大規模なデータの分散処理のモデルに幅広く利用されている．

MapReduce はユーザーが Map フェーズと Reduce フェーズの処理内容を記述するだけで簡単に分散処理を行うことができ，記述内容次第で様々な種類の問題に適応可能である．また，多数のマシンで構成されているクラスタ上で動作するように実装されており，大規模データを効率良く分散処理するために Map フェーズと Reduce フェーズに記述された処理を行う．Map フェーズではマスターノードによって分割された入力データそれぞれに対し処理を行い，Reduce フェーズでは Map フェーズの出力を集約する．基本的な流れを（図 1）に示す．

また，MapReduce では，マスターノードが入力データを分割して割り当てる Map タスクはワーカーノードの数に比べて多い方がよい．そして，マスターノードによって分割されたそれぞれのタスクは均等な処理時間で行われるものである方がよい [?].

例えば，ある組み合わせ最適化問題を仮定する．組み合わせの制約を一つ一つ追加していくことによる状態の遷移を，問題の初期状態やすべての制約が追加された状態以外の，まだ追加されていない制約のある状態も含めた状態の存在範囲を状態空間とする．そうすると状態空間のそれぞれの状態と制約の追加による遷移で論理的な木を生成することができる．それが列挙木の生成であり，その列挙木の中に存在する特定の最適解を探すことが列挙木の探索である．

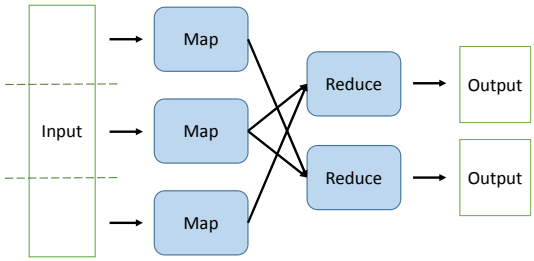

図 1: MapReduce における基本的な動作の流れ

MapReduce を列挙木の探索問題に適応するには,（図 2）のように探索問題の最初に根から生成されるいくつかの子問題（子である状態が持つ問題）を，それぞれ Map フェーズの入力タスク（Map タスク）として割り当てる手法が素朴である．

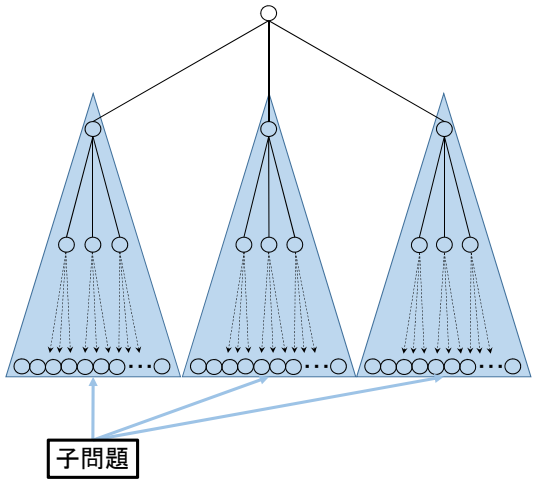

図 2: MapReduce における列挙木探索の素朴なタスク割り当て例

しかしながら，列挙木の探索においては，この手法をそのまま適応するのはあまり効率的ではない．なぜ

なら状態空間の性質上，入力に対してその先に莫大な状態空間が広がる場合が多いため，最初に生成される子問題の数は全体の状態数に対して少なく，それぞれの子問題の木の大きさに非常に大きな差が生まれる可能性がある．したがって Map を行う多数のマシンに割り当てるにはタスクの数が少なく，その上それぞれの Map フェーズの実行時間に大きな差ができる [?]．また，MapReduce の性質上，Map タスクの最中に他の Map タスクや Reduce タスク内で得られた情報をやり取りすることが難しい．

今日，BigData の重要性が注目を浴び，効率的にデータを管理・処理するフレームワークが必要とされている．その中で Hadoop を始めとする BigData 用システムが幅広く使われ，代表的な分散処理フレームワークとして MapReduce が注目されている [?]．列挙木探索を分枝限定法を用いて行うフレームワークはいくつか開発されているが，汎用性の高い MapReduce が多くの企業で導入されている中，列挙木の探索において他の専用のフレームワークを使用するのは不便である．本報告では，本来 MapReduce での計算に不向きだと考えられる列挙木の探索問題を，主にノードへの Map タスクの割り当て方と，Map タスクでの計算内容を操作することにより，総探索時間の短縮を図る．また，提案手法では逐次実行時ほど正確にではないが，限定操作を行うことも可能にする．本稿では提案手法の概要を説明し，提案手法を用いて実装した特定の問題（0-1 ナップサック問題）を解くシステムを Hadoop [?] 上で動作させた実験結果を元に提案手法を評価する．

2 関連研究

2.1 MapReduce

MapReduce は大規模なデータを分散処理するためのプログラミングモデルである．1 台のマスターノードと複数台のワーカーノードで処理を行う．マスターノードは分散処理の指示・管轄を行い，ワーカーノードは実際に処理を行う．マスターノードはユーザーから受け取ったジョブをタスクと呼ばれる小さい単位の処理に分割し，複数のワーカーノードにタスクの処理を要求する．その後，マスターノードはユーザーからのジョブが全て完了されるまで，ジョブやタスクの処理の進捗・タスクのワーカーノードへの割り当て・ワーカーノードの死活などの状況を監視し，管理する．基本的な処理の流れは以下である．

1. マスターノードは入力データを分割しファイルシステムに保持して，複数のワーカーノードに割り当ての指示を出す．
2. マスターノードから割り当てられたデータをファイルシステムから受け取ったワーカーノードは，Map クラスに記述された処理を行い，何らかの中間（Key,Value）ペアを生成し，出力する．（Map フェーズ）
3. Map フェーズから出力された中間（Key,Value）ペアを Key の内容でソートする．このとき同一の Key はまとめられる．（Shuffle フェーズ）
4. Shuffle フェーズによってまとめられた（Key,Value）ペアに対し，Reduce クラスに記述された処理を行い，結果を最終出力として出力する．（Reduce フェーズ）

MapReduce の特徴としては処理の継続性や拡張性，データの局所性などがある．

処理の継続性 MapReduce では，マスターノードでジョブとタスクの進捗を管理している．また，Map 処理や Reduce 処理で扱うデータは基本的には他の Map 処理や Reduce 処理とは関連がなく独立している．したがって，あるタスクがワーカーノードの故障によって中断された場合に，他のワーカーノードに同じタスクを再度割り当てることにより処理を継続することが可能である．

処理の拡張性 Map 処理や Reduce 処理は，処理するデータを分割することにより複数のワーカーノードで並列処理が可能である．したがって，データノードの台数を増やすことで処理性能の向上を図ることができる．

データの局所性 Map 処理はできる限り処理をしているワーカーノードと同じノード上で起動しているファイルシステムのデータを利用しようとする．これによりデータの通信を抑え，ネットワーク帯域の節約や通信コストを抑えることに繋がる．

基本的な MapReduce の実装でユーザーが記述するのは Map フェーズと Reduce フェーズの処理内容だけである．

2.2 Hadoop

本研究では，Hadoop により開発・公開されている Hadoop MapReduce を使用する．

Hadoop は Apache Software Foundation が開発・公開している，大規模データの分散処理や管理を行うためのソフトウェア基盤である．オープンソースソフトウェアとして公開されており，誰でも自由に利用することが可能である [?]．Hadoop は多くの要素で構成されるが，共通の基盤システムである“ Hadoop Common ”をベースに，前述の Hadoop MapReduce や HDFS, HBase などが主に開発されている．

2.2.1 HDFS

HDFS（Hadoop Distributed File System）は，Hadoop で提供している分散ファイルシステムであり，大きな

ファイルを複数の計算機にまたがって格納することが出来る．HDFS はデータの複製を複数のホストに格納することにより信頼性を確保している．

2.3 Solving Hard Ploblems with Lots of Computers

Solving Hard Ploblems with Lots of Computers は，組み合わせ最適化問題をクラスタなどを用いて並列化し解くことに対する課題を探ることを目的とした研究と，得られた課題を改善できるような動的フレームワークを提案している．この研究では次のように述べられている．「分岐限定法は探索木の力まかせ探索に基づく方法であり，この方法では洗練された限定技術により最適解を含まない部分問題の切り捨てを行う．しかしながら，以降で述べる理由により分岐限定法は MapReduce にうまくマッチしない.」とあり，その理由部分が「分岐限定法は，部分木の計算にどれくらいの時間 (仕事量がいかほどか) がかかるか実行してみるまでわからないため，MapReduce などのフレームワークで実装されている静的な分割では効率はあまり良くならない.」である．これらの事実を踏まえて，本報告では MapReduce のフレームワークをそのままに少しでも効率的に分枝限定法を行う手法を提案する．

3 研究背景・諸定義

3.1 状態と状態空間

組み合わせ最適化問題等において，解を探す各段階における対象の有様や制約を状態（State）と呼び，全ての状態の集合からなる論理的な仮想の空間を状態空間 (State Space) と呼ぶ．状態空間の例を（図 3）に示す．

3.1.1 初期状態

状態空間の状態のうち，入力直後の一番初めの状態を初期状態と呼ぶ．

3.1.2 許容解

状態空間の状態のうち，すべての制約を決定し，もうそれ以上有様の変わらない状態を許容解と呼ぶ．許容解は最終的な最適解になる可能性がある．

3.1.3 目的状態

許容解のうち，求める最適解である状態を目的状態と呼ぶ．

3.1.4 中間状態

状態空間の状態のうち，初期状態でも許容解でもない状態を中間状態と呼ぶ．

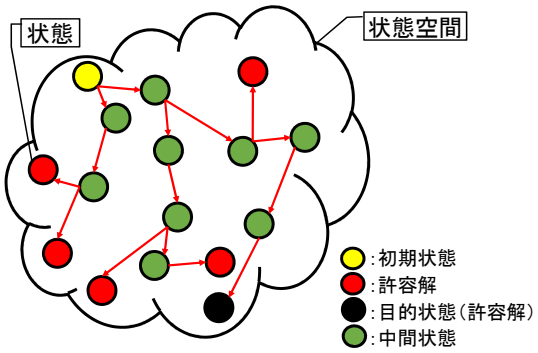

図 3: 状態空間の例

3.2 列挙木

状態空間を表した木のことを列挙木と呼ぶ．状態空間を木で表すには，問題の初期状態を根として表し，ある状態での条件の追加や分岐から他の状態が生成される様子が木における親から子が生成される様子で表される．すなわち，目的状態は列挙木においては葉になる．解も含めて，木における葉が許容解になる．初期状態や各中間状態から生成される次の状態をそのノードの子として記述し，子の問題を子問題と呼ぶ．

例として，各状態において次に生成される状態が高々二つであるような状態空間の列挙木は（図 4）のような二分木になる．

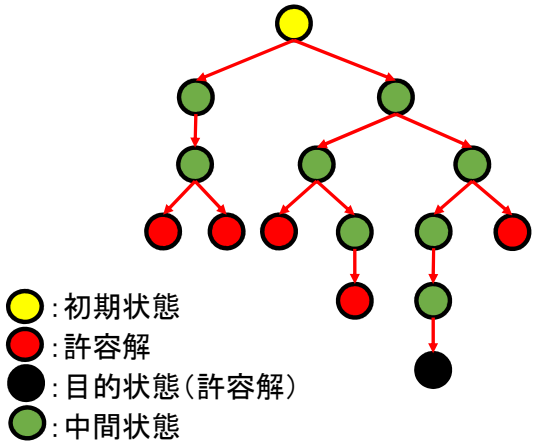

図 4: 列挙木の構築例

3.3 ステップ

状態空間探索の列挙木において，ある中間状態が 1 つ先の状態（子）を探索することを 1 ステップと定義する．（図 5）に例を示す．

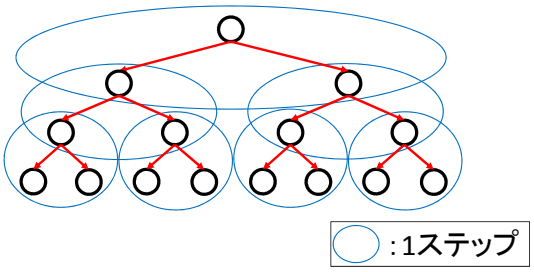

図 5: 1 ステップの定義

3.4 ポテンシャル

各状態が持つ，最適解になり得る期待値のこと．ナップサック問題のような最大化問題のある状態においては，その状態から進んでいくと得られる可能性のある値の上界のこと．

3.5 分枝限定法

分枝限定法（branch and bound）は，組み合わせ最適化問題の最適解を求める汎用アルゴリズムである．分枝操作と限定操作から構成される．以下では，$f(x)$ の最大値を求める最適化問題を例に説明する．$x \in S$ とする．

3.5.1 分枝操作

分枝操作は，解く問題を場合分けにより部分問題に分割することである．つまり，S を $S_1 \cup S_2 \cup S_3 ... = S$ となるような複数の集合 $S_1, S_2, , ,$ に分割する．S_i における $f(x)$ の最大値を v_i とすると，S における $f(x)$ の最大値は $max\{v_1, v_2, ...\}$ である．なお，$S_i \cap S_j \neq \phi$ でもよい．

3.5.2 限定操作

限定操作は，S_i の最大値の上界や下界を計算する手続きである．求めた上界と下界を用いて，以下に述べる枝刈りを行うことが可能になる．

3.5.3 枝刈り

例えば S_i の最大値の上界が S_j の下界より小さければ，S_i には $f(x)$ を最大にする x は存在しないので，それ以上探索をしないようにする．それが枝刈りである．枝刈りにより総探索数は大幅に減少する場合も多い．

3.6 対象問題

本報告では，提案手法を評価するために具体的な問題を解くシステムを実装する．対象とする問題に“ 0-1 ナップサック問題 ”を設定する．

3.6.1 0-1 ナップサック問題

0-1 ナップサック問題とは，分枝限定法が有効な代表的な組み合わせ最適化問題であり，NP 困難であることが知られている．

「入る重さの容量（capacity）が C のナップサックと n 個の種類の荷物（item）があり，荷物には各々重さ（weight）と価値（value）がある．ナップサックの容量 C を超えない範囲で荷物をナップサックに詰めていき，ナップサックに入れた荷物の価値の和を最大にするにはどの荷物を入れればよいか.」という問いに対して最大の価値の和とそのときの荷物の中身を求めればよい．一つ一つの荷物については，ナップサックに入れるか入れないかの二択である．n 個の荷物を $x_0, x_1, x_2, ..., x_{n-1}$ とし，荷物 X_i がナップサックに入っているときを $X_i = 1$，入っていないときを $X_i = 0$ と表記すると，0-1 ナップサック問題の解の探索は (図 6) のような列挙木で表すことができる．(図の簡略のため $n = 3$ としている)

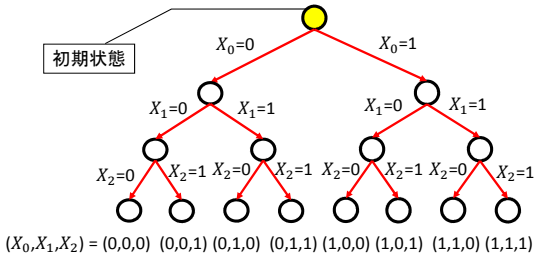

図 6: 0-1 ナップサック問題の列挙木

3.6.2 問題設定

提案システムへの入力として荷物の数 n が，10 個,15 個,20 個,25 個,30 個,35 個,40 個,50 個,100 個の 9 通りの入力サイズを用意する．対象とするのは二分木なので，木の深さは n であり，探索数の上界は $O(2^n)$（枝刈りが起こらずに全探索した場合）である．

4 提案手法

本報告では，MapReduce による列挙木探索の手法を提案する．提案手法の目的は，MapReduce のフレームワークをそのまま利用した上で，本来 MapReduce に不向きな状態空間の列挙木探索を効率的に行うことである．

4.1 概要

提案手法の基本的な流れを記述する．

1. 入力が初期状態，つまり入力用の中間データを保持していなければ，使用するワーカーノード全てに最初から仕事が渡るようにマスターノードは状態数がワーカーの数になるまで探索する．そしてその探索結果の状態を Map タスクとして各ワーカーノードに割り当てる．入力用の中間データがあれば，それをポテンシャルの高い順にワーカーノードに割り当てる．
2. Map タスクでは，ワーカーは受け取った状態から，ユーザーに指定されたステップ数だけ探索を行う．複数状態を保持しているときは，各状態の中で一番ポテンシャルの高い状態を探索する．指定されたステップ数の探索が終わると，辿り着いた全ての状態を Reduce タスクに受け渡す．
3. Reduce タスクでは受け取った状態がまだ探索中の中間状態か，許容解（最終的な解とは限らない）かどうかを確かめる．解ならば現在の最適解（以後，暫定解と呼ぶ）と比較してより良い方を保持し，中間状態ならば次のように限定操作を行う．

   中間状態のポテンシャルが暫定解以下のとき
   その中間状態を探索し続けても暫定解より良い解は得られないので，その中間状態を探索対象から削除する．

   中間状態の現在のポテンシャルが暫定より大きいとき
   その中間状態を次の MapReduce の入力とする．
4. 全ての Map,Reduce タスクの終了後，次の MapReduce への入力があればもう一度 MapReduce を実行する（1. へ戻る）．

   なければ暫定解を最終的な最適解として出力する．

以上の提案手法の流れを（図 7）に示す．

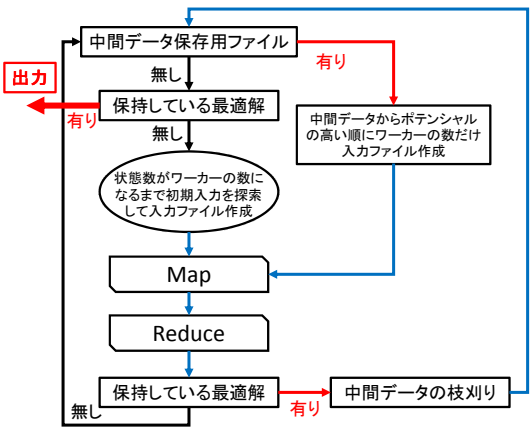

図 7: 提案手法の流れ

4.2 有用性

本報告の主な利点を説明する．状態空間の探索を分枝限定法で高速で行う研究はいくつかあるが，新たなフレームワークを用意しなければならないか，もしくはユーザーに高いプログラミング能力を求められるものばかりである．本報告は，状態空間の列挙木探索が MapReduce フレームワークの処理の手順そのままに，Mapper と Reducer を定義するだけで，高い可能性を持った状態から探索する Best-First Search を実現することができる．また，割り当てる Map タスクの数と 1 回の実行での探索ステップ数をユーザーが指定できるので，実行時のクラスタ環境や取り組む問題の規模に合わせて指定するだけで提案手法を問題により効率良く適応することが可能である．

4.3 提案手法の設定問題への適応

本報告では，探索の高速化のため，n 個の入力荷物は単位重さあたりの価値の大きさ（以降，比率と呼ぶ）が高い順に並んでいるものとする．n 個のデータのソートは $O(nlogn)$ の時間複雑度で可能であり，これは総探索時間に比べて極めて小さく，全体計算量に影響を及ばさないため一般生を失わない．

探索はナップサックに何も入っていない状態を初期状態とし，荷物 X_0 から順に入れるか入れないかで分岐しながら次の状態に進んでいく．初めてある中間状態が n 個目の荷物について入れるか入れないかの探索を終えると，そのナップサックに入っている荷物の価値の和をそのときの最適解として保持する．2 回目以降はそのとき保持している最適解と得られた解を比較し，大きい方を最適解として保持する．一度の MapReduce で進むステップ数はユーザーが指定し，最終的な解が得られるまで複数回 MapReduce を実行する．MapReduce の実行が終わる度に中間状態に対して限定操作を行う．つまり，指定したステップが終わった段階で最適解を保持していれば，その最適解以下のポテンシャルをを持つ中間状態は削除し，その先の探索を行わない．

5 実装システム

5.1 実行環境

本システムの実行環境は以下の表の通りである．

CPU	Intel Core i3 (2cores/4threads)
RAM	2GB(16 台)/4GB(20 台)
HDD	500GB
OS	CentOS 5.8 (Linux Kernel 2.6.x)
Framework	Hadoop 1.2.1 stable

マスターノードを 1 台，ワーカーノードを 35 台の計 36 台で実験を行った．なお，マスターノードには RAM が 4GB のマシンを使用している．

5.2 引数と入力データ

実装システムはマスターノードで実行する．実行時の引数は以下の二つである．

第一引数 使用するワーカーノードの数（以下，nMaps）

第二引数 探索で進むステップ数（以下，nSteps）

ステップ数の大小で探索終了までの実行回数が変わる．探索終了まで手動で繰り返し実行する必要がある．また，問題の入力データはファイルに入れてシステムに受け渡す．入力ファイルの中身は，1 行目にナップサックの容量と荷物の数，2 行目以降に各行 1 つずつ荷物の重さと価値を効率の良い順に記述する．

5.3 Map 処理

Map タスクを行うワーカーノードは，入力の中間状態に対して，まだナップサックに入れるか入れないか決まっていない荷物 X_i のうち i が最少のものについて入るか入らないかを確かめる．ナップサックに荷物 X_i が入るときは入れるパターンと（$X_i = 1$）入れないパターン（$X_i = 0$）の状態を，入らないときは入れないパターンの状態のみを生成する．生成したのが中間状態なら保持，許容解なら Reduce に出力する．ここまでが 1 ステップの動作である．

ユーザーが指定したステップ数が 2 以上のときは指定されたステップ数まで引き続き，前述の動作を自身が生成した中間状態の中で一番ポテンシャルが高い中間状態を取り出し，その状態に対して行う．自身が生成した中間状態がない場合は，ステップが残っていても処理を終了する．

指定されたステップ数まで処理を終えると，保持している中間状態をすべて Reduce に送り，処理を終了する．

5.4 Reduce 処理

受け取った状態に対して，入れるかどうかの判定済みの荷物の数で中間状態か許容解かを判定する．

中間状態ならば，受け取った状態をすべて中間データを保持するためのファイルに書き込む．

許容解ならば，暫定解として最適解を保持するファイルに書き込む．このとき，すでに何らかの最適解を保持しているならば，新たな許容解と比較してナップサックに入っている荷物の価値の和の大きい方を最適解として保持し直す．

Reduce の終了時に保持している最適解を，暫定解を保持するファイルに書き込む．

5.5 マスターノードの処理

マスターノードが Map タスクを生成する動作は 1 回目の実行と 2 回目以降の実行で大きく異なる．1 回目は初期状態から，状態数が nMaps になるまで逐次探索を行い，それによって得られた中間状態を Map タスクの入力とする．2 回目以降は中間データを保持しているファイルからポテンシャルの高い順に中間状態を nMap 個取り出し，得られた中間状態を Map タスクの入力とする．

Map/Reduce タスクの実行後は，暫定解がファイルに保持されていれば，中間データを保持しているファイルの中の中間状態で暫定解よりポテンシャルの低いものは削除する．このとき中間データを保持するファイルが空になれば，そのとき保持している最適解が最終的な解となるので標準端末に出力してシステムを終了する．

6 実験

実装したシステムの評価を行うために他システムと提案システムとの比較実験を行った．

6.1 評価軸

総探索時間（time） 各 Map タスク内での探索時間の合計．

総探索数（count） 各 Map タスク内での探索回数の合計．新しい状態に辿り着く度にインクリメントする．

6.2 比較対象

本報告の提案手法は探索中の限定操作とアルゴリズムの並列実行を実現している．したがって本実験での比較対象は以下の 4 通りである．

Parallel-BnB Hadoop 上で動作し，並列実行を行う．提案手法のようなステップの指定はなく，最初に割り当てられた Map を行うノードが許容解まで探索を行うため，1 回の実行で解が出力される．また，MapReduce では Map タスク実行中のワーカーノードのデータの受け渡しは不可能なので，各々の Map タスク内での限定操作は行っているが，あくまでローカルでの限定操作のみとなっている．

Parallel-DFS Parallel-BnB から各 Map タスクのローカルな限定操作を除いたものである．Hadoop 上で動作し，並列実行ではあるが限定操作は一切行っていない．

Single-BnB 限定操作ありの逐次実行アルゴリズムである．並列処理は一切行っていない．

Single-DFS 限定操作なしの逐次実行アルゴリズムである．

6.3 実験結果

以下の記載する実験結果のグラフでは，提案システムを OurSystem と表記している．

6.3.1 Parallel-BnB

Parallel-BnB システムと提案システムの比較グラフは（図 8）である．

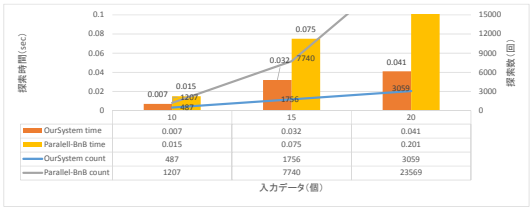

図 8: Parallel-BnB との比較結果

グラフを見ると，どちらのシステムも入力データ数の増加とともに time,count ともに増加しているが，増加の割合が圧倒的に Parallel-BnB の方が大きい．入力データが 20 個の時点で提案システムは time,count ともに Parallel-BnB より明らかに小さく，このまま入力を増加させるとこの差はさらに広がっていくと考えられる．

6.3.2 Parallel-DFS

Parallel-DFS システムと提案システムの比較グラフは（図 9）である．

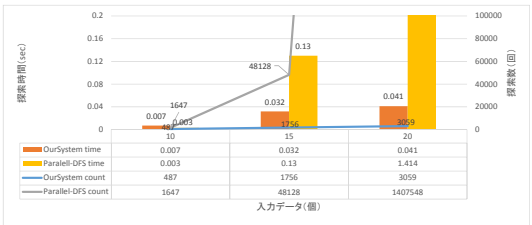

図 9: Parallel-DFS との比較結果

グラフを見ると，どちらのシステムも入力データ数の増加とともに time,count ともに増加している．入力データが 10 個のときは，両システムともに time はとても小さくあまり差はないが，count は分枝限定法を実装している分提案システムが Parallel-DFS の 4 分の 1 程度になっている．入力データが 15 個になると time も提案システムが Parallel-DFS の 3 分の 1 程度になり，count の差はさらに広がっている．入力データ 20 個でも提案システムと Parallel-DFS の time,count の差はさらに大きく広がっているので，このまま入力を増加させても差は広がり続けると考えられる．

6.3.3 Single-BnB

Single-BnB システムと提案システムの比較グラフは（図 10）である．本グラフは差の見やすさの都合上,（図 8）などとは違い，縦軸 time の単位を (msec) にして log スケールで表示した．

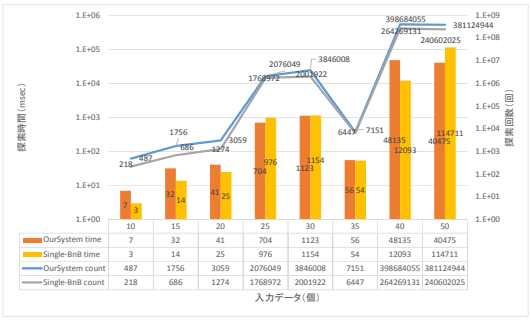

図 10: Single-BnB との比較結果

グラフの下のデータテーブルを見ると，入力データが 10〜30 個まではデータの増加とともに両システムにおいて time,count ともに増加傾向にあるが，入力が 35 のところで両システムともに 30 の場合に比べ count,time ともに大幅に減少している．40 から 50 にかけても少しではあるが同様の現象が見られるが，これらは両システムともに起こっていることから，入力データに原因があると考えられる．基本的にナップサック問題は入力データが増えると探索する状態数も増えるが，入力データを効率順にしている本実験では稀に探索初期の段階でナップサックの容量が満たされる最適解が発見されるようなケースが起こり得る．したがって，35 と 50 はこのような入力データであったことが予想される．

Single-BnB は逐次的に限定操作を行いながら探索するので，count は必ずその入力データに対して最小となる．提案システムは並列処理中のマシン間での限定操作を行えないため，count は必ず Single-BnB の方が小さくなる．しかし，状態空間探索は入力が増えていくと探索する状態数が爆発的に増えるため，逐次実行では入力データ数がある程度大きくなると実行時間もとても増えるはずである．したがって time に着目して比較を行う．入力データサイズが小さいと，逐次で行っても短い時間で終わるため，並列処理の利点を生かせないので，入力データが 20 個までは time はわずかに Single-BnB の方が小さい．これは探索数の差が原因だと考えられる．しかし，入力データが 25 個か

らは提案システムの方が time が小さくなり，その差は 40 のときに一気に広がっている．これは入力データが 30〜40 程度が，並列処理により 1 探索あたりの探索速度は速いが探索数は多くなる提案手法と逐次実行で 1 つずつ無駄なく探索を行う Single-BnB とが総探索時間の点でおよそ釣り合うデータ数であることを表している．したがって，このまま入力を増加させても time の差は広がり続けると考えられる．

6.3.4 Single-DFS

Single-DFS システムと提案システムの比較グラフは（図 11）である．

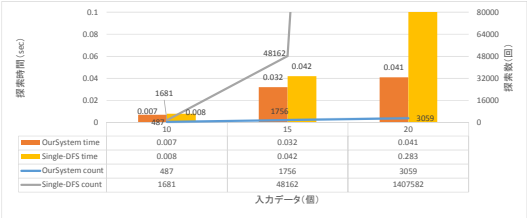

図 11: Single-DFS との比較結果

Single-DFS は初期状態から逐次的に探索を行い，限定操作をしないため，入力データに対する最大の探索数となる．グラフを見ると，入力データが 10 個のときは入力が小さいために両システムの time にはほとんど差がないが，count はすでに提案システムの方が 4 分の 1 で探索している．入力データが 15 個の時点で既に count の差は爆発的に広がっており，time も提案システムの方が小さい．入力データ 20 個でも time,count の差はさらに大きく広がっているので，このまま入力を増加させても差は広がり続けると考えられる．

6.4 考察

4 つの比較グラフを見ると，入力データ数が少ない場合は，提案システムと他のシステムとの間に大差はない．しかしながら，入力データ数が多くなるにつれて大幅に提案システムの探索時間は他のシステムに比べ短くなっている．また，他の並列分散システム（Parallel-BnB,Parallel-DFS）ではメモリ容量を超えて実行できなかった $n = 100$ のときの探索も可能であった点から，提案システムはそれらのシステムに比べて実行時のメモリ使用量が少ないこともわかる．これは処理を複数回の MapReduce に分けて行うため，1 回の実行時に探索する状態数が少ないことで，必要とするメモリが少なくなると考えられる．

以上より，提案システムは 4 つの比較対象システムよりも大きな入力に対応でき，入力が大きくなればなるほど相対的に探索が速くなることがわかる．

7 今後の課題

本報告の提案手法は，MapReduce を複数回実行する性質上，MapReduce の起動時や終了時などにワーカーの起動, 終了やタスクの読み込みなど探索時間以外の冗長な時間が生まれる．本研究における実験では,1 回の実行につき 10〜15 秒の冗長な時間が発生した．そのため，探索時間や探索回数が小さくても全体の実行時間では（図 12）のように 1 回の MapReduce の実行で探索する場合よりも長くなってしまう場合が多い．

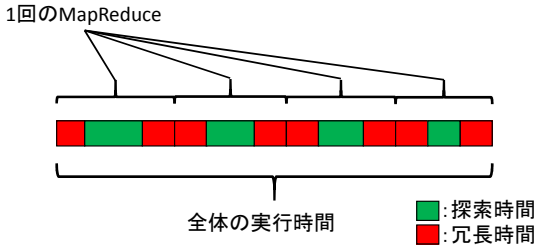

図 12: 提案システムでの全体実行時間例

短い探索時間を生かすために，MapReduce の繰り返し実行をスムーズに行うように改良する必要がある．この繰り返し実行を高速化するフレームワークとして，Apache Spark [?] が登場した．Apache Spark の動作の流れは MapReduce と基本的に同じだが，繰り返し利用するデータを MapReduce はストレージで保存していたのに対して Apache Spark は自身のメモリに保存することで出し入れの高速化を図っている．したがって，提案手法を Apache Spark フレームワークで実装すれば全体の実行時間も短縮できる可能性があるため，現在取り組んでいる．総探索時間の削減が実験により確認できた提案手法に対して，Apache Spark の仕様で全体の実行時間が削減されるのであれば，Apache Spark での提案手法の実装は計算量・時間の両面で有用であると言える．

また，提案システムの汎用化も大きな課題である．本報告での手法では 1 回目の実行の際の初期状態を nMaps 個の状態になるまでの探索や,2 回目以降の入力となる中間データの扱いを問題の内容に合わせてメインクラスに記述しなければならない．これらの部分を汎用化することが出来れば，MapReduce を通して状態空間の探索をより手軽に効率良く行うことが出来るようになるなど期待できる．

8 おわりに

本報告では状態空間におけるポテンシャルを利用し，MapReduce での限定操作の実現による列挙木探索手法の提案と評価と行った．一度の MapReduce で許容解まで探索せずに中間状態を保持することにより，得られたその時の最適解との比較で限定操作を行ってい

る．それにより，少なくとも今回取り上げた 0-1 ナップサック問題については，MapReduce で素朴に探索したときよりも大幅に短い探索時間と少ない探索回数を可能であることを実験により示した．また，現在の提案手法は，探索対象がよほど大きな状態空間でない限り MapReduce を繰り返し実行することによる冗長時間が探索時間に比べて長くなるという現在の一番の課題を踏まえ，今後の展望を述べた．

参考文献

[1] Jeffrey Dean and Sanjay Ghemawat,”MapReduce: Simplified Data Processing on Large Clusters”, USENIX Association OSDI ’04: 6th Symposium on Operating Systems Design and Implementation,2004

[2] Internet Engineering Task Force(Feb. 14 2013 14:40 UTC), ”Combinatorial_explosion”, http://en.wikipedia.org/wiki/Combinatorial_explosion

[3] Sandy Ryza,”Solving Hard Problems with Lots of Computers”, Brown University Department of Computer Science,2012

[4] Tom White,”Hadoop”,O’REILLY,2011

[5] Mihai Budiu and Daniel Delling and Renato F. Wernek,”DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines”,Microsoft Research Silicon Valley Mountain View, CA, USA,2012

[6] Richard Neapolitan and Kumarss Naimipour,”Foundations of algorithms(4th Edition)”,JONES AND BARTLETT PUBLISHERS,2011

[7] Internet Engineering Task Force(Feb. 14 2014 14:45 UTC), ”hadoop PoweredBy”, http://wiki.apache.org/hadoop/PoweredBy

[8] Internet Engineering Task Force(Feb. 14 2014 14:45 UTC), ”hadoop”, http://hadoop.apache.org/

[9] Internet Engineering Task Force(Feb. 14 2014 14:50 UTC), ”BigTable”, http://ja.wikipedia.org/wiki/BigTable

[10] Internet Engineering Task Force(Sep. 12 2014 11:30 UTC), ”Apache Spark”, https://spark.apache.org/