平成21年度　卒業研究論文

解候補の探索優先度を考慮した
部分問題の再配分を行う並列A*の検討

指導教員

松尾　啓志教授

津邑　公暁准教授

名古屋工業大学 工学部　情報工学科

平成18年度入学　18115031

大嵜 惇也

平成22年 2月 8日

解候補の探索優先度を考慮した部分問題の再配分を行う並列A*の検討

目 次

1 はじめに

CPU の演算性能の限界により実時間内での解答が困難な大規模演算に対して、並列演算は複数の CPU を協調させて演算性能を高めることで高速に解答を得る研究が行われている。かつては端末間のネットワークの通信速度が端末内部の通信速度に比べて非常に遅いため、台数の増加に見合った性能向上を果たすことは困難であった。また単一 CPU の性能向上が著しい発達もあり、アルゴリズムの研究は逐次処理の最適化が中心であった。この状況が近年の通信速度の向上やマルチコアプロセッサの普及によって変化し、並列アルゴリズムの重要性が増している。

並列アルゴリズムの研究に、探索アルゴリズムの並列化がある。探索アルゴリズムは問題に対して大量の候補を 1 つずつ調べていくことで最良の解を求めるアルゴリズムである。その適応範囲は広く、計算機内部での高速なデータアクセスの様なミクロな問題からゲームの次の手の選択決定などの複雑な問題にまで至っている。しかし、大量な候補の調査のため、解を返すまでの使用メモリ量や計算コストが総じて高く、実時間での解答が困難である場合も多い。そのため状況や目的に応じた様々な逐次探索アルゴリズムが考案されてきた。

探索アルゴリズムには問題の持つ性質を知識として探索の補助に用いることで効率的な探索を行う手法がある。その代表的なアルゴリズムとして A*アルゴリズムが提案されている。A*アルゴリズムは目標までの予測コストを問題の知識から算出して、これを探索の補助として利用する。得た予測からより評価を行う解候補を効率よく選出しながら探索処理を進めるというアルゴリズムである。A*アルゴリズムは知識を用いない探索アルゴリズムよりも各候補の情報の精度が高くなることが多く、探索の性能を向上させやすい。

探索アルゴリズムの効率化は長期に渡って続けられてきたが、依然として解答に必要とする計算コストは高く、複雑な問題を解答する為には高価なスーパーコンピュータを利用した大規模演算を行うことも少なくない。現在の大規模計算機はクラスタをはじめとする大規模並列演算処理で実現されるため、必然的に探索アルゴリズムの並列化も研究された。

探索アルゴリズムを並列化する研究が進むにつれ、処理の一貫性の保証や端末間の通信・計算のオーバーヘッドのトレードオフ、メモリの効率的運用の問題や解候補の分散手法が大きな課題となった。これに対し、これまでに候補から求めたハッシュ値による探索担当端末の決定手法や、メモリのスケーラビリティの確立、非同期通信を使用して処理を独立化させることで通信オーバーヘッドの隠蔽が実現されてきた [7]。

その結果、メモリを効率的に利用しつつ計算コストを抑え、通信オーバーヘッドを軽減した探索が可能になった。

本研究では従来手法とは異なるアプローチでの高性能並列探索の検討を行う。本研究の目標は従来手法では確率的に行っていた解候補の分散を、探索優先度を考慮して明示的に配分することで、探索精度に理論的根拠を持たせつつ高性能な並列探索を行えるようになることである。本論文ではこの手法を実装した場合に、従来手法以上の探索性能を得られるかどうかの検証を行う。その検討として従来手法による探索空間の分割の上で、明示的に候補の探索優先度を考慮して再分散させる並列 A*を用いる。

探索アルゴリズムを適用する問題として 15 パズルの最小手数探索を設定した。15 パズルは古典的なスライドパズルであるが最小手数解の探索は現在でも大きなコストを必要とする問題である。本研究で実装した並列探索アルゴリズムは、各端末が保有する未調査の候補を互いに動的に配分しあう。これにより各端末が保有する未探索候補の探索優先度の均等化を図っている。

本論文では研究の背景となる探索アルゴリズムの対象にした問題について第 2 章で、既存の A*アルゴリズムの並列化に関する研究を第 3 章で説明する。その後、検討手法について第 4 章で説明した後、実験とその結果をそれぞれ第 5 章で述べる。

2　背景・準備

本章では、研究の背景である一般的な探索アルゴリズムについての概要と形式についての説明を行った後、本研究の対象とする A*アルゴリズムの詳細を説明する。また、評価のための例題として用いた 15 パズルの説明と A*アルゴリズムの適用法を説明する。

2.1　探索アルゴリズム

探索アルゴリズムは問題が与えられると、そこから考えられる解の候補を調査してゆき、調べた候補の中で最適な解を返すアルゴリズムである。

データアクセスの為の線形探索や 2 分探索、チェスなどのボードゲームの「手」を探し出す敵対探索など様々な問題が提案され、また問題に応じてそれぞれに適当な逐次アルゴリズムが考案されている。

2.2　形式化

探索問題は、次のような形式で表現される (参考出典：[9])。

- 状態
  例えば将棋の場合は盤面の駒の配置が状態になり、迷路の場合はプレイヤーの現在位置の座標がこれに該当する。与えられた問題の性質によって状態群の全体あるいは一部が解候補となる。特に、最初に問題として与えられる状態を初期状態、目的とする状態を最終状態と呼ぶ。

- 状態遷移 (状態変化、オペレータ)
  例えば将棋の場合はプレイヤーによる駒の移動、迷路の場合はプレイヤーの移動といったように、何らかの動作によってある状態を別の状態に変化させる手段のことを状態遷移または状態変化、オペレータなどと呼ぶ。

- 探索空間 (状態空間)
  ある状態に状態遷移を適応することで問題の対象の状態は変化する。与えられた問題の初期状態から状態遷移を繰り返すことで到達しうる全ての状態の集合のことを探索空間と呼ぶ。問題によって探索空間全体を推測できる場合とそうでない場合がある。

- 拘束条件
  目標を達成する為に守らなければならない条件のことを拘束条件と呼ぶ。これは与えられる問題特有のものであり、状態遷移の方法に制限が加えられたり解答到達までの状態遷移回数やコストに条件がつけられることもある。

2.3 探索空間のグラフ表現

計算機科学におけるグラフは、図 1 の様にいくつかの節点と枝から構成され複数の節点が枝によって結ばれたものである [11]。枝には方向性があるものと無いものがあり前者を有向辺、後者を無向辺といい、有向辺を含むグラフを有向グラフ、無向辺のみで構成されるグラフを無効グラフと呼ぶ。また、グラフ内にループ構造が存在しないグラフを木構造のグラフという。

問題が与えられるとき、問題は探索空間を直接的または間接的に定義している。ある問題を与えられた場合に状態を節点、状態遷移を枝に対応させることで探索空間はグラフに対応させることが出来る。つまり探索は初期状態を出発節点、最終状態を目標節点に対応付けられたグラフの探索に帰着させたグラフ探索アルゴリズムと等価になる。探索の方法を抽象的に記述することによって様々な問題の探索に共通した解法を考えることが出来る。

以降の説明で、問題をグラフに帰着させた説明の為に探索空間内の状態のことを節点と表記する事がある。

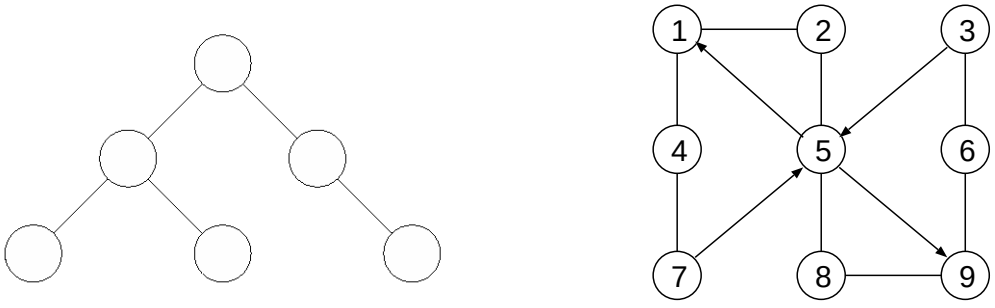

図 1: 木構造の無向グラフと一般的な有向グラフ

2.3.1 基本的な探索の方法

探索アルゴリズムはグラフを、与えられた初期節点から各節点を調査しながら辿って行き、高速条件を満たす目標節点の発見やその経路を発見することを課題とする。問題によってグラフの構造は様々に変化するが、探索アルゴリズムの基本的な処理手順は同じである。

探索アルゴリズムの中に各状態の調査状況を管理しておく為にリストを用いる手法がある。リストとして未評価の節点を管理する為の open リストと、評価済みの節点を管理するための closed リストを用意し、必要に応じて節点情報の追加、削除、変更を行う [9]。

2.3.2 知識を用いない探索アルゴリズム

問題の探索空間に関する知識が無い (使わない) 場合、探索空間の探索には何らかの法則を用いて一定の順序で調べるしかない。その場合に使用されるのが深さ優先探索、幅優先探索といった探索アルゴリズムである。

探索空間が有限であり、空間内に解が存在するならばこれらのアルゴリズムは理論上は必ず解を見つけ出すことが出来る。また、最短経路を発見するという拘束条件が存在している場合にはダイクストラ法 [10] と呼ばれる探索アルゴリズムが汎用的に使用される。

2.3.3 基本的な探索アルゴリズムの問題点と対策

一般的に探索空間は非常に広い。これまでに紹介した探索アルゴリズムでは小さな問題でも解に到達するまでには時間を必要とすると言う問題がある。

例えば 2 分木のグラフを考えると、深さ n の節点の数に対して深さ n+1 の節点の数はちょうど 2 倍になる。このように探索状態は初期状態からの深さが深くなればなるほどに状態の総数を指数的に増加させてゆく。これは問題の複雑性が少し上昇するだけで探索候補が急激に増加し、それらの探索候補を調査して回答に到達するまでに多大な時間を必要とする事を意味する。また同様の原因で、リスト管理の為に使用されるメモリ量が爆発的に増加して使用可能なメモリ領域を埋め尽くすという問題がある。

この問題は、探索を行う節点の数を減らすことで改善することができる。その基本的な方法として「枝刈り」と呼ばれる手法が存在する。「枝刈り」では探索の中で調べたある節点が拘束条件を満たさないものであった場合にその節点以降に続く節点は探

索を行わないというものである。これは評価によって条件を満たさないと分かった節点について、そこから遷移してゆく節点は確実に拘束条件を満たさないことが保証されるためであり、「枝刈り」は探索効率を向上させる有効な手法である。

「枝刈り」以外の方法として問題の知識を利用する方法が考えられる。この方法を使うことで状況に合わせた探索処理を進め、効率よく候補を探索していくことができる。

2.4 A*探索アルゴリズム

問題の知識を利用する代表的な探索アルゴリズムがA*アルゴリズム [1] である。A*では問題の性質を利用しヒューリスティクスと呼ばれる問題固有の値を補助として使うことで探索の効率化を図る。A*アルゴリズムの動作イメージについては図2に示す。

2.4.1 A*アルゴリズムの概要

ある問題に対して、スタートとなる節点Sからゴールとなる節点Gまでの最適な経路が発見できたとする。このとき、経路の中に節点nが存在していると仮定すると、発見した経路の全コストは (1) 式で表現することができる。

$$f(n) = g(n) + h(n) \tag{1}$$

ここで $f(n)$ はnを通りSからGまで到達するときの最小コストであり、$g(n)$ はSからnまでの最小コストであり、$h(n)$ はnからGまでの最小コストである。全ての節点に対してそれぞれh(n)が既知である場合、探索処理を簡単に説明すると次のようになる。

1: openリスト中から $f(n)$ の値が最も小さい状態を取り出し、評価する

2: 状態nの展開によって発見される全ての状態mについて、それぞれ $f(m) = g(m) + h(m)$ を求め、未探索の状態や再探索の必要のある状態をopenリストに、また状態nをclosedリストにそれぞれ格納する。

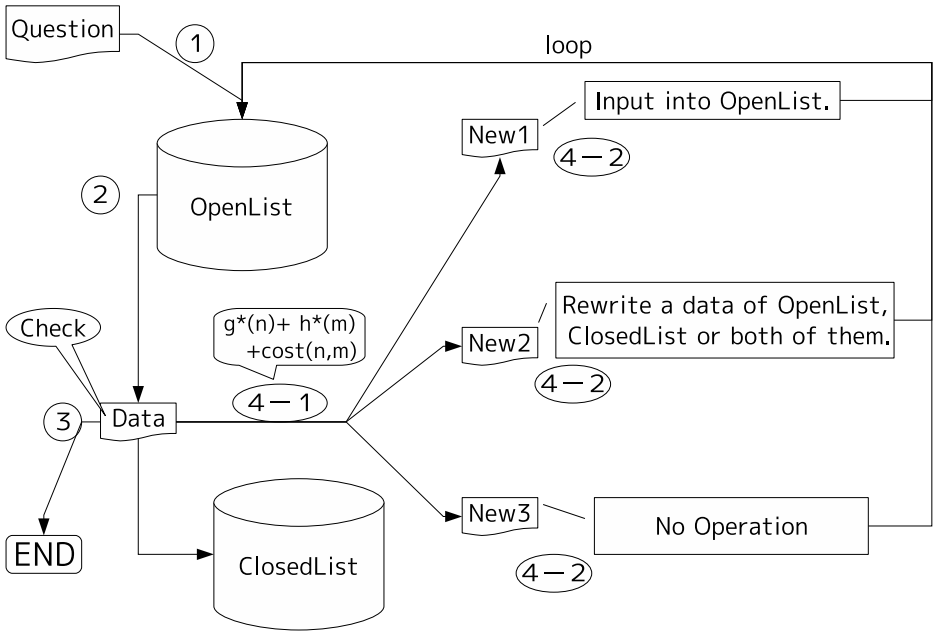

図 2: A*アルゴリズムの動作

実際には $g(n)$ や $h(n)$ の値は未知の状態から探索が始まることになる。そこで、$f(n)$ の替りとして予測値 $f*(n)$ をもちいて (2) 式のように置換する。

$$f*(n) = g*(n) + h*(n) \tag{2}$$

$g*(n)$ についてはダイクストラ法や幅優先探索などを使用することで探索を実際に行いながら $g*(n) = g(n)$ となる値を求めていくことができる。一方、$h*(n)$ は探索が終了して初めて明確になる値であるため確定した値を使うことはできない。

そのため $h*(n)$ には問題の性質から、状態に対して適当な値を計算して割り当てる。そのための関数をヒューリスティック関数といい、ヒューリスティック関数により求められる値のことをヒューリスティクス値と呼ぶ。ヒューリスティック関数の例として、例えば地図上を道路沿いに移動した時の最短経路を求めたい場合には、現在位置とゴール地点の直線距離を用いることができる。

このように探索中に明確にできる $g*(n)$ を適宜その値を最適値に更新しつつ、探索をすすめる探索アルゴリズムを A 探索アルゴリズムという。

A 探索アルゴリズムのうち、特に $h*(n)$ について、(3) 式に示されるように

$$\forall n0 \leq h*(n) \leq h(n) \tag{3}$$

であることが満たされている時、$h*(n)$ は許容的（アドミッシブル）であるといい、この条件を満たしたアルゴリズムを A*アルゴリズムという。A*アルゴリズムでは高速条件を満たす節点が始めて発見された時点で、それが最小コストの解であることが保証される。

$h*(n) = 0$ として探索を行う場合 A*の条件は常に成り立つため汎用的に利用できるがこれはダイクストラ法と等価である。$\forall n, h1*(n) < h2(n) \leq h(n)$ となるヒューリスティック関数が存在していることが分かっている場合、h2(n) を使用した方が計算負荷が少なくて済むことが多いので、許容的なヒューリスティック関数が存在するならば A*アルゴリズムを適応する方が良い。

2.4.2 A*アルゴリズムの動作

A*アルゴリズムの動作のイメージを図 2 に示した通りである。また、図 2 中の動作番号に従って A*アルゴリズムを実装する時の処理の流れの詳細を図 3 に示す。2 つの図の処理手順の番号はそれぞれ対応したものとなっている。A*アルゴリズムの実装も既に紹介した探索アルゴリズム同様 open リスト、closed リストを活用して実装することになる。

2.5 15 パズル

本研究では探索を行う対象として 15 パズルの最短手数探索を対象とした。15 パズルにおける A*アルゴリズムではヒューリスティクス値にマンハッタン距離が利用できる。

本節では 15 パズルに関する説明及びマンハッタン距離について説明を行う。

0: 最終状態（以下 G）と初期状態（以下 S）を作成する。また、未探索の状態を格納しておく為の open リスト、探索済みの状態を格納しておく為の closed リストを用意する。

1: S を open リストに追加する。この時 S の保有する情報として $g*(S) = 0, f*(S) = h*(S)$ とする。探索が行われる前なので、closed リストは空にしておく。

2: open リストに格納されている状態のうち、最小の $f*(n)$ を持つ状態 n を取り出す。open リストが空である場合、解答には到達できないと判断し、終了する。

3: $n = G$ であるならば探索は終了、解答を表示する。そうでない場合は n を closed リストに追加する。

4: 状態 n を展開し、新たに作成された全ての状態 m について、

4-1: "$f'(m) = g*(n) + h*(m) + cost(n, m)$" を計算する。ここで $cost(n, m)$ とは、状態 n から状態 m での移動する為のコストである。

4-2: m の状態に応じて評価結果に応じたリスト操作を行う。

* m が open リストにも closed リストにも格納されていない場合、$f*(m) = f'(m)$ とする m を open リストに格納する。
* m が open リストに既に存在しており、かつ $f'(m) < f*(m)$ である場合、$f*(m) = f'(m)$ に置き換える。また内部データとして m の親を n に置き換える。
* m が closed リストに既に存在していおり、かつ $f'(m) < f*(m)$ であるならば、$f*(m) = f'(m)$ とする m を open リストに移動させる。また内部データとして m の親を n に置き換える。

5: 2∼4 を解答が見つかるまで繰り返す。

6: 経路探索が終了したのち G から親を順番にたどることで S から G までの最短経路を得ることができる。

図 3: A*アルゴリズムの動作

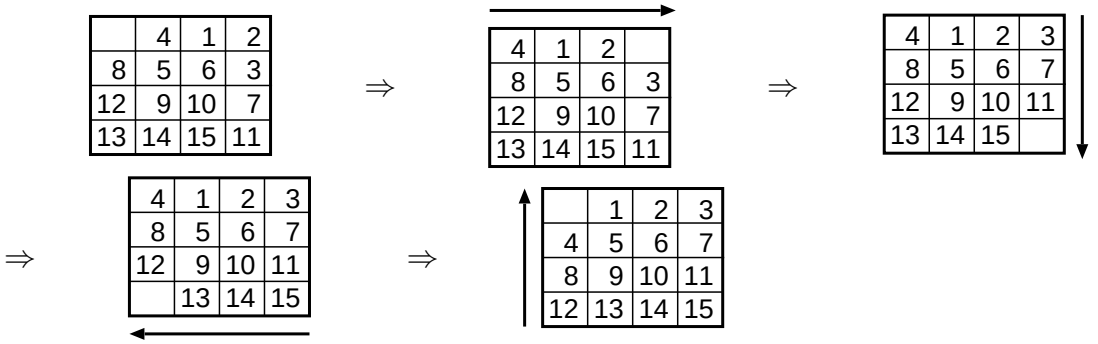

図 4: 例題と解答手順

2.5.1 15 パズル概要

15 パズルはスライドパズルの一種である。

15 パズルは 4 × 4 の正方形のマスで構成される盤面上に 1 ～ 15 までの数字のマスと 1 つの空白マスが配置されている。このうち 15 枚の数字マスのうち、空白部分と隣接するマスをスライドさせることによって盤面を操作し、マスの数が整然となった状態を作ることを目的としている。これまでの研究で 15 パズルの最長手数は 80 手であることが証明されている [3]。本研究ではこの１５パズルの最小手数解の発見を課題に設定し、A*アルゴリズムを用いた探索処理を行う。

15 パズルは全ての盤面に対して、その盤面が「到達できない盤面」が存在している。例えば目標盤面のうち 1 と 2 のマスが入れ替わった盤面は、どれだけスライドを行っても目標盤面に到達することは無い。このような到達できない盤面が到達できる盤面と同数だけ存在している。

図 4 に例題と解答手順を示す。この問題では、空白のマスが盤面の外周に接しているマスを時計回りに 1 周する様にスライドさせることで最終盤面に到達する。

2.5.2 マンハッタン距離

15 パズルに A*アルゴリズムを使用する際のヒューリスティクス関数としてマンハッタン距離とよばれる概念を利用できる。

マンハッタン距離とは幾何学上の距離概念の一つであり、2 点間の距離を各座標の差（絶対値の総和）とする。形式的には 2 点間の距離を直交する座標軸に沿って測定することで n 次元に於けるマンハッタン距離が定義される。マンハッタン距離は、目

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

図 5: 15 パズルの盤面と xy 平面座標への対応

標となる状態があらかじめ分かっている問題を対象とした場合については 15 パズルに限らず汎用的に使用できる。

例えば 15 パズルでは左右方向を x 軸、上下方向を y 軸とする xy 平面上と考えることで各マスに $0 \leq x < 4, 0 \leq y < 4$ の範囲に各マスが配置されているととらえることができる。これにより 2 つの点 P1(x1,y1) と P2(x2,y2) 間のマンハッタン距離は (4) 式で計算できる。

$$| x1 - x2 | + | y1 - y2 | \tag{4}$$

15 パズルにマンハッタン距離を適用することを考える。

15 パズルでは空白を除く各マスはそれぞれに固有のゴール地点となるマスが予め決まっている。これを利用して 15 パズルのヒューリスティクス関数を、「全ての数字マスの、現在位置と目標位置のマンハッタン距離の総和」と定めることでその盤面に対するヒューリスティクス値を算出できる。

各マスで求めたマンハッタン距離はそのマスが「最低でも何回スライドさせる必要があるか」と一致している。実際には途中で別のマスとの兼ね合いによってあるマスにとっては余分なスライドを必要とすることがある。これにより算出したヒューリスティクス値よりも実際には手数が多くなることは珍しくない。一方でヒューリスティクス値よりも少ない手数で最終状態に到達することはありえない。

このことが 15 パズルにおいてこのヒューリスティック関数を適応することは h*(n) として許容的（アドミッシブル）であることを示している。

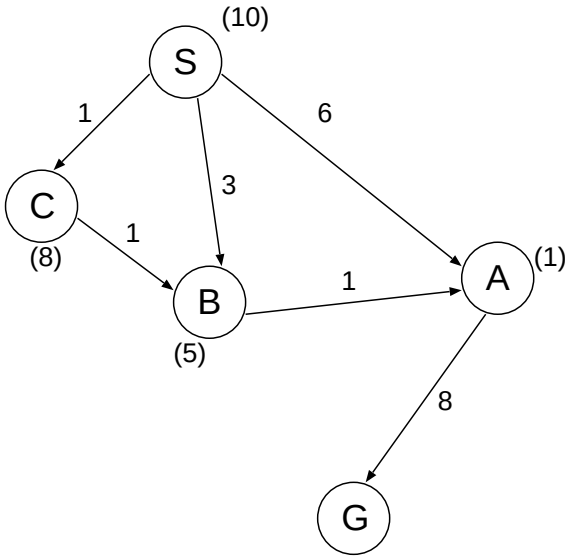

図 6: 再探索が頻発するグラフ

3　従来研究

本章では A*アルゴリズムの並列化に関する従来研究について説明を行う。

3.1　逐次探索の改良

A*をそのまま適応した場合、探索の効率があまり改善されない場合が存在する。図 6 に示したグラフのようなグラフを探索を行うと、節点の再探索が何度も必要となり結果として探索が A*が効率的に実行されない。

図 6 に対する探索の挙動について、通常の A*の動作は表 1 左に示す通りである。通常の A*の動作表を見ると状態の探索の順番は、”$S \to A \to B \to A \to C \to B \to A \to G$” であり、節点 A、B にそれぞれ複数回の探索処理が行われている。

一方、表 1 右に示されているのが、探索優先度を変更した改良 A*の動作表である。

こちらでは探索の順番は ”$S \to C \to B \to A \to G$” となり各状態はそれぞれ 1 回ずつ探索を行うだけで正しく最適解に到達することが確認できる。

表 1: 通常の A*の探索動作（左）と改良 A*の探索動作（右）

S	A	B	C	G
10				
10^+	7	8	9	
10^+	7^+	8	9	14
10^+	5	8^+	9	14
10^+	5^+	8^+	9	12
10^+	5^+	7	9^+	12
10^+	4	7^+	9^+	12
10^+	4^+	7^+	9^+	11
10^+	4^+	7^+	9^+	11

各節点 n についての f*(n) を記述
数字の右側の”+”は探索済みを示す

S	A	B	C	G	f_m
0+10					0
$0+10^+$	6+1	3+5	1+8		10
$0+10^+$	6+1	2+5	$1+8^+$		10
$0+10^+$	3+1	$2+5^+$	$1+8^+$		10
$0+10^+$	$3+1^+$	$2+5^+$	$1+8^+$	11+0	10
$0+10^+$	$3+1^+$	$2+5^+$	$1+8^+$	$11+0^+$	11

各節点 n についての g(n)+h*(n) を記述
各節点が最適化後に探索されている
最右の”f_m”はその時点での予測コスト

A*アルゴリズムを適応したことで節点の再探索が頻発し、探索の効率が落ちてしまうことがある。そこで探索順を変更した改良 A*[9] を使用する。A*アルゴリズムが同じ節点を再評価しないためには節点 n_i とその子節点 n_j に (5) 式に示す条件が成立していれば良い。

$$h(n_i) \leq h(n_j) + Cost(n_i, n_j) \tag{5}$$

これは単調性の制約条件と呼ばれるもので、表 2 に示した条件を満たすように取り出して評価を行う候補を変更することで同じ節点を複数回調査することがなくなる。これを適応するとき、A*が open リストから取り出す候補の処理順が修正後の動作に変化する。

表 2: 探索優先度の変更

通常の A*	f*(n)=g*(n)+h*(n) で求められた f*(n) の最も小さな状態 n
改良 A*	f*(n) のうち、更にその中でも g*(n) の値が最も小さい状態 n

このように探索処理が単調性の制約を満たすようにすることでグラフの探索効率が上昇する。これは逐次処理での A*探索の効率化の研究であるが、並列探索アルゴリズムにも効果がある。並列探索では逐次処理とは異なり、複数の端末で一つの問題の探索処理を進めるため、各端末が探索処理の現状を正確に把握することが困難になる。これは逐次探索では発生し得ない現象であり、再探索の可能性を増大させる原因となる。

並列探索処理において再探索が発生にくくするために、逐次探索の挙動を考慮することは重要である。

3.2 PRA*(Parallel Retracing A*)

PRA*(Parallel Retracting A*,Evett et al.1995)[4] はコネクションマシン CM-2 に搭載された並列 A*である。Retracting A*と呼ばれる、省メモリに特化した並列 A*を実装したものである。コネクションマシン CM-2 とは、１ビットの単純なプロセッサを最大 65,536 台がハイパーキューブ型に接続された SIMD 演算方式の超並列コンピュータである。

PRA*では並列化の為に状態をハッシュ関数をかける。ハッシュ関数は、並列化した端末のうちのどれか 1 つのアドレスを返す。各端末は得られるハッシュ値から探索を行う担当となるプロセッサを一意に決定する。探索空間全体を全端末に一様に分散させるハッシュ関数を用いて確率的にロードバランシング実現している。各プロセッサは自身のローカルメモリの情報のみを使用して受け取った状態の探索の必要性を判定する。これは他の端末への確認処理を行わないことで通信オーバーヘッドを削減し、処理の高速化を図ったものである。

一方で PRA*は CM-2 に搭載するように設計された為、各探索処理の結果を配分する為に同期が必要である。CM-2 は SIMD 演算のマシンであるため各端末で実行される全ての処理は同時に行われる。そのためプロセッサ間での多大な同期、通信オーバーヘッドを必要とした。更に、現在と比較して各プロセッサがローカルに保持しているメモリ量が少なく、メモリがデータで圧迫された時に Retraction 処理を実行して不要なデータを削除することでメモリ空間の確保を行うという探索以外の処理オーバーヘッドの必要があった。以上の様に PRA*は課題を多く残した並列探索アルゴリズムである。

3.3 TDS(Transposition-Driven-Scheduling)(P-IDA*)

TDS(Transposition-Driven-Scheduling,Romein et al. 1999)[5] は分散メモリ環境下での処理のスケジューリング手法であり、評価のために並列反復深化 A*（Parallel IDA*,P-IDA*）を実装した。

PRA*同様、TDS でも状態をハッシュ関数にかけることで節点の探索を担当させるプロセッサを決定させている。一方で PRA*と異なり、TDS ではハッシュ値としてシグネチャと呼ばれるデータを使用している。シグネチャは巨大な数であり、そのビット列の一部には、PRA*同様に状態の探索担当となるプロセッサ番号が指定されているが、更にトランスポジションテーブル（closed リストに相当するテーブル）のインデックス値を含んでいる。TDS の要点はシグネチャによって各プロセッサが保管する義務を負うデータが互いに素になるように分割配置されているということである。

TDS ではハッシュ関数として Zoblist のハッシュ関数 [2] を使用している。シグネチャの導入により TDS は並列演算を行う端末の台数が増えるにつれて TDS 並列計算機全体で使用できるメモリ量が台数に比例して増加するというメモリのスケーラビリティを実現した。これは台数が増えれば増えるほど、メモリが圧迫されるまでに使用できるメモリ量が増加するため、重複状態の発見や枝切りがより効果的になることを意味している。

また PRA*と異なりプロセッサ間の全ての処理を非同期に実行することで、各プロセッサは独自に探索処理を進めることが可能であり、更にこれによりプロセッサ間での通信オーバーヘッドを探索処理により隠蔽も図ることが出来る。各プロセッサの処理能力の差などの理由でプロセッサが保有する仕事量に偏りが発生した場合には WS(Work Steal) 機構により処理の負荷分散を行っている。WS は探索処理が進み open リストに状態がなくなった端末に探索処理の終わっていない端末の仕事の一部を肩代わりさせる機構である。

TDS に発生しうる問題点として、論文 [5] 中で探索処理ごとに生成されるデータの送信処理オーバーヘッドについて言及している。一般的に通信処理のコストはプロセッサ内部の演算に比べて非常に遅いため、状態の展開により作成された新たな状態情報をシグネチャにしたがって別端末に送信する処理が非常に多くなる。そのため並列探索では通信処理オーバーヘッドが探索処理のボトルネックになる。しかしこの問題について、同論文内では探索担当プロセッサへのデータをまとめて送信することで、送信オーバーヘッドは軽減されるため大きな問題で無いと論じている。

3.4 HDA*(Hash Distributed A*)

HDA*(Hash Distributed A*,Kishimoto at el,2009)[7] は PRA*、および TDS の利点を用いて共有メモリ、分散メモリのどちらの環境でも効果のある並列 A*アルゴリズムである。

状態の分散戦略は PRA*と同様にハッシュ関数を利用した分散手法を使用し、通信処理の部分には TDS での非同期通信手法を採用しており、また closed リストおよび open リストはプロセッサ毎に互いに素な状態で保有させている。一方で PRA*の様に Retraction 機構のような複数プロセッサの協調を必要とする処理を行わないため、探索速度および使用メモリ量の双方に台数分のスケーラビリティを実現させている。また、探索処理以外の計算オーバーヘッドを最小にするため TDS 同様ハッシュ関数には Zoblist のハッシュ関数を使用、プランニングには LFPA ヒューリスティクスによって機能を高めた逐次 FastDownward を使用する（Helmert,2007）[6] ことで性能向上を図っている。ただし FastDownward+LFPA は最適な形で並列化されておらず今後の課題としている。

HDA*における特徴の一つとして共有環境での並列化に関する性能向上手法が挙げられる。通常の共有メモリ環境下ではプログラムのマルチスレッド化により並列化を行うが、スレッド間でデータの送信を行うためには他の端末がメモリを使用できないように”lock”が必要となる。これは送信を実行するスレッド側で、メモリアクセス権限を獲得するまで処理を停止し待ち状態に移行しなければならないことを意味し分散メモリ環境化における同期に相当する。そのため共有メモリ環境下での並列化において性能向上に対する最大のボトルネックとなっていた。HDA*では各処理単位が自身の管理するリストが明確に分割されているため共有メモリ環境化であっても分散メモリ環境と同様に MPI でのノンブロッキングの Send,Recv 命令を使用させることで自身の担当メモリ空間を独自に管理することを可能としている。

4　検討案

本研究では従来手法とは別アプローチからの並列探索の検討として、解候補を一定のタイミングで再配分しあうことで open リスト探索優先度の平均化を図る。また、探索の効率を挙げるために 15 パズルの省メモリ化を目的とする手法を取り入れた。本章ではそれら効率化の手法と並列化手法の高性能化の検討案について説明を行う。

4.1　従来手法についての考察と検討

従来の並列 A*アルゴリズムは探索空間の分散配置の為に状態から計算したハッシュ値を使用していた。

例えば TDS や HDA*は状態を管理する担当を決定するために使用するハッシュ値の計算のために状態性質に依存しない Zobrist のハッシュ関数を用いている。Zobrist のハッシュ関数は前ハッシュ値を次のハッシュ値の計算に利用できる性質を持ち、十分なビット長があればコリジョンは殆ど発生しなくなる。つまり Zobrist のハッシュ関数により最小限の計算コストでハッシュ値を求めることができることを意味する。得られたハッシュ値は乱数となり、それを一様に分布させる分布関数用いて分散トランスポジションテーブルを構築できる。

一様分布を生成する関数としては、適当な範囲の乱数の商を用いる方法が計算コストが小さく実装が容易な手法として広く使われている。

これにより並列探索においては、探索空間を各端末に確率的に均等に配置することになる。各端末が担当する状態空間の範囲は確率的に同じ大きさになり、本論文で考える探索優先度も自然と均等に配置される。このよいに確率的手法によって探索空間の分割を行うだけで、同時に候補の均等配置についても理にかなった空間の分割が実現できる。

従来の分散手法を踏まえ、並列探索の性能向上を図るために必要な要点について考えてみる。並列探索処理の効率化に必要な要素はメモリのスケーラビリティの確立による端末処理の独立化とそれにより可能になる通信処理の非同期性の実現、処理オーバヘッドの計算量削減、および端末間での負荷のバランシングである。理想的には一つの手法で全ての処理を完成させることであるが、従来手法ではこのうち確率的配分手法により第 1、第 2 の要素を同時に実現可能にし、負荷のバランシングには WS を用いていた。全ての要点を満たすことが出来ないのは、ハッシュ値が問題の性質をハッシュ値に含むことができず、探索空間を予め静的に配分しているためである。

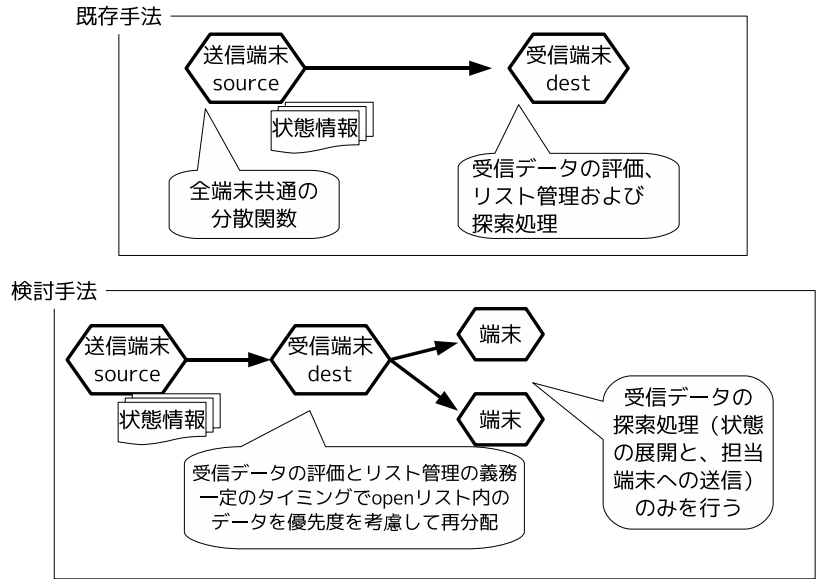

図 7: データ再配分のイメージ

そこで本研究では確率的手法ではなく状態のもつ情報を考慮して状態を動的に配分するアプローチを考える。そしてこのアプローチが確率的配分手法と同等以上の性能が得られるかどうかの検討を行う。

4.2 探索候補の優先度を考慮したデータの再配分

本研究の目標は並列探索の性能向上の為に先に述べた要点を一括して行える手法の提案である。そのため状態を配分する時に探索優先度を考慮する処理について検討する。

4.2.1 データ再配分の実装

単純に展開処理を行って新たに得られた解候補について、それらの探索優先度を用いた動的配分を行うだけでは、メモリのスケーラビリティを破壊してしまうことになる。例えば、既に探索された候補が再び作成された場合に、探索処理を行ったのとは別の端末に送信されてしまうと、探索が行われていない候補として再び探索処理が実行されてしまうという現象が発生する。これは明らかに余分な処理であるため、動的

な候補の分散を行うにしても、候補が不要に複数回の探索処理を実行されてしまう状況を回避する必要がある。しかし問題回避の為には静的な候補の処理担当を必要とする。そこで本研究では従来手法の分散分散手法を利用して、問題回避の為の処理オーバヘッドを最小限に抑え、その上で動的な候補の分配を行わせるよう実装を行った。

つまり本研究で実装した分散手法は、2 つの通信処理を用いてデータを 2 段階に分けて配分処理を行う。1 つ目は従来手法でも用いられているハッシュ値を用いた静的な分散トランスポジションテーブルの確立の為の通信である。各候補は、一意に決定される担当端末に送信され、そこで候補の探索が必要かどうかの判定を受ける。そこで探索が必要だと判断された候補は従来手法どおり一度端末の open リストに登録され、端末内部の探索優先度に従って並べられる。これにより従来手法の利点を確保する。

従来手法ではこの通信によって受信した候補の全てについて受信端末がリスト管理と探索処理の責任を持っていた。しかし、リスト管理さえ確かであるならば、探索処理そのものは別の端末が行っても問題は発生しないはずである。そのため並列探索処理の最適化の為にに 2 つ目の通信として、各端末にで自身の open リストの一部を、一定のタイミング毎に端末全体に再配分させる処理を新たに追加する。通信されるのは特に探索優先度の高い候補である。リストの内容全体を通信しないのは、優先度の低い候補が何度も通信される事を防ぐためである。探索処理によって新たに作成される候補があることも考慮し、再配分の為の通信は処理全体に大きな影響を与えない程度に速度を重視する必要があると考えられるためである。

再配分を行う手法を実装したときに発生するコストとそれを最小限に抑える手法を説明する。再分散により発生するコストは、データ送信処理の増加による通信オーバーヘッドである。ある候補について複数回の通信が行われる可能性が生まれたことからこれは従来手法に比べて確実に増加する事が考えられる。また再分散によって本来送られてくるはずのない状態情報を受信する可能性が発生する。そのためのリスト管理の処理オーバーヘッドが増加することが考えられる。

コストを最小限に抑えるため、通信される候補の付加情報を追加した。再配分される状態は、送信元の端末で既に評価が行われているため、受信者が新たに状態の評価を行う必要はないものである。そこで余分なメッセージの評価を行わないようにする為に通信メッセージを従来のメッセージに加えて「探索要求」のメッセージを新たに用意する。各端末は受信した状態情報に付与されたメッセージ要求を調べ、通常のメッセージならば候補に対する評価を行い探索が必要か選別を行い、探索要求のメッセージであれば候補の評価なしに open リストに状態情報を追加する。以上により各端末が closed リストに使用するメモリ量を従来手法と変化させることなく、また余分な評価

処理を行わない再分散処理を実装した。

次に通信オーバーヘッドの増加について行った対処について説明する。今回の実装では再配分処理の頻度を通常の探索処理に対して少なくすることで再分散の影響を抑えている。再分散の頻度を高くすると最適なデータの配置が可能であるが、通信およびリスト管理オーバーヘッドが大幅に増加することになる。そこで再配分の頻度は通常のメッセージ送信の頻度に対して少なくする代わりに、一度に配分するデータを多くすることで通信オーバーヘッドを抑えたまま処理の最適化を図る。これは従来のハッシュ値に基づくデータの静的配分が探索空間をある程度均等に分割していることを利用している。

4.2.2 検討手法の懸念事項

検討手法にはopenリストの内容を再配置する計算オーバーヘッドと再分散の為の通信オーバーヘッドがある。

並列探索には効率化を妨げる要因として通信オーバーヘッド、計算オーバーヘッド、使用メモリ量の三つが挙げられ、しかもこれらは多くの場合でトレードオフの関係にある事が多い。これまでの並列A*の研究の流れは、これらの並列処理のコストを如何に削減するかというものであった。TDSではデータの分散配置による使用可能メモリ量のスケール化と非同期通信による状態の計算処理を多重化して通信オーバーヘッド隠蔽を実現し、HDA*ではZoblistのハッシュ関数による計算オーバーヘッドの削減と、共有メモリ、分散メモリのどちらの環境でも並列探索を効率的に実行できる汎用性を実現している。

一方、今回の検討案である再配分は従来手法に追加処理を加えている。追加された再配分によって余分な通信，計算オーバーヘッドが発生し、openリストにデータを追加するリスト操作にも処理オーバーヘッドが発生する。そのため現状では検討手法を従来手法に比べて一概に優れているとすることは出来ないことを確認しておく。

4.3 省メモリ化手法

本研究では状態の再配分以外に、処理の省メモリ化手法として、15パズルの性質を利用した探索手法とハッシュ関数を使用している。本項目ではそれらについて説明する。

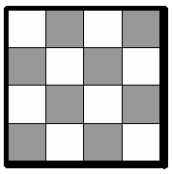

図 8: 盤面の偶奇性

4.3.1 解答手数の偶奇性の利用

15 パズルでは一度の操作では一枚のパネルが空白の位置にスライドされ、そのパネルがあった位置が新たに空白となる。すなわち、空白マスは隣接マスとの交換されている。

この時、図 8 を例に考えると、空白マスは一手ごとに盤面上の黒いマスの部分と白いマスの部分を交互に移動しているといえる。この性質から最終目標となる盤面の空白の位置と、問題として与えられた盤面の空白の位置を比較するだけで問題の解答に必要な手数が奇数手か偶数手のどちらになるかを判別することが出来る。

15 パズルでは、ある盤面に対する展開処理について、盤面の１手先の盤面ではなく２手先の盤面を作成して探索を進めても矛盾が発生しない。なぜならば先の性質からある盤面に対して１手だけ多い（少ない）手数で同じ盤面に到達する方法が存在し得ないからである。

よって、２手先まで展開することで奇数手の情報あるいは偶数手の情報の一方のみが作成されるように探索処理を行うことでリストに格納される情報を実質半数に減少させることが可能になる。また、一度の探索により展開される盤面が増加することで TDS で指摘されている様な頻発する通信処理によるオーバーヘッドの削減を行いやすくなるという利点がある。

4.3.2 階乗進数変換を利用したハッシュ関数

15 パズルの場合は 0 ～15 の数が配置されているが、これを一列に並べることで各数に重複の無い順列が構成できる。この全組み合わせは $\sum_{k=1}^{16} k!$ 通りである。重複の無い順列について、数列を階乗進数で表現される数に変換し、更に基数変換を行うことで状態を一つの数に変換する手法 [8] がある。

この変換によって得られる数は変換前の全組み合わせに対して一切重複することが無く、更に変換後の数の範囲は要素の数 n に対して $[0, \sum_{k=1}^{n} k!)$ となる。この変換手

3	7	1
6		2
8	4	5

図 9: 例図

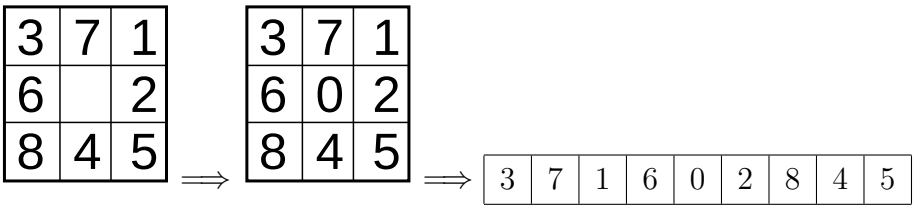

図 10: 盤面から数列を作成

法を使用することで全状態組み合わせに 1 対 1 対応する完全最小ハッシュ関数を構成することが可能である。

このハッシュ関数の最大の利点は、通常のハッシュ関数とは異なり得られるハッシュ値から変換前の数列を復元できることである。すなわち、ハッシュ値そのものが状態を示す情報として扱うことで状態の保存に必要な情報量の削減を実現させることができる。

ハッシュ関数を 15 パズルに適用する方法を説明する。ただし簡単のため図 9 で示した 3×3 で構成される 8 パズルの盤面を例にする。 第 1 に盤面からハッシュ値を求めるハッシュ関数について説明する。ハッシュ値の計算には数列で階乗進数表記の数を作成する。そのための最初の手順として盤面の空白を 0 に見立てて順列を作成する (図 10)。次に、得られた順列について階乗進数の数に変換する。順列を階乗進数とみなす場合、それぞれの桁の基数はその桁以下の数の総順列数となる。例えば 9 桁目の基数はその下の 8 桁の全順列の数である 8!である。同様にして各桁に対して対応する基数を各桁の数に掛け合わせゆくのだが、順列は階乗進数でないためにそのまま基数を掛け合わせてはならない。順列を階乗進数の数に変換する必要がある。その手順は図 11 の通りである。

この手順に従って図 10 に示した順列からハッシュ値を計算する時の動作を表 3 に示す。この変換により最上位の桁から、盤面に 1 対 1 対応する階乗進数表記の数を表す数列を得る。

step1: 値を決定していない最上位の桁を確定させる

step2: 値の確定した桁以下の桁の中で、決定した桁の値以上の値を持つ桁が存在した場合、その桁の値を 1 だけ減ずる

step3: 全ての桁が確定するまで step1,2 の操作を繰り返す

step4: 全ての桁が値を確定させたときその数列が盤面に対する階乗進数となる

図 11: 順列の変換手順

表 3: 例図に対して順列の変換を行った時の動作

									説明
8!	7!	6!	5!	4!	3!	2!	1!	0!	基数
3	7	1	6	0	2	8	4	5	盤面のマスを並べた順列
[3]	6	1	5	0	2	7	3	4	← 下位桁の 3 以上の数を-1
3	[6]	1	5	0	2	6	3	4	← 下位桁の 6 以上の数を-1
3	6	[1]	4	0	1	5	2	3	← 下位桁の 1 以上の数を-1
									(省略)
3	6	1	4	0	0	2	[0]	0	← 下位桁の 0 以上の数を-1
3	6	1	4	0	0	2	0	[0]	← 最終桁を決定
3	6	1	4	0	0	2	0	0	得られる階乗進数

最後に、図 12 に示すように算出した数列に対して基数を掛け合わせ、足し合わせることで数列に対するハッシュ値を算出する。

次に、得られたハッシュ値から元の盤面情報への逆変換の手順を説明する。説明にはハッシュ関数の例で求められたハッシュ値を用いて説明していく。まずハッシュ値から進数変換を行って階乗進数の数に変換する。ハッシュ値に対して上位桁から基数の商とその余りを再帰的に計算することで階乗進数に変換できる。これは方法は通常の基数変換と同様の処理である。基数変換処理によって得られた階乗進数に対して変換前の順列への逆変換処理を行う。逆関数では値を決定しない数が主に変化する。逆

8!	7!	6!	5!	4!	3!	2!	1!	0!
3	6	1	4	0	0	2	0	0

$$\Rightarrow\ 8!\times 3+7!\times 6+6!\times 1+5!\times 4+2!\times 2$$
$$=\ 152404$$

図 12: 階乗進数変換によるハッシュ値の計算

step0: 各桁に全桁数だけのフラグを配列等で用意する

step1: 桁の確定していない最上位桁を決定する

step2: 残りの桁の確定していない数について**下位桁から上位桁の順に**以下の操作を行う。

- 2-1: 対象の桁の上位桁を新たに確定した桁まで調べる。
- 2-2: 調べた桁の中で自身の値以下の桁が存在した場合その数を記憶する。
- 2-3: 記憶していた桁のうち、その桁に対応するフラグを調べフラグの立っていない桁が存在したならば、フラグを立て、その数を数える
- 2-4: 数えられた値だけ自身の値を増加させる

step3: 全ての桁が確定するまで step1,2 を繰り返す

step4: 全ての桁が確定したときの数列が、元の盤面を一列に並べた順列になっている

図 13: 逆変換の手順

変換の手順を図 13 示す。

この手順を先ほど得た階乗進数に適応した場合の動作例を表 4 に示す。なお、動作例では 1 の位の桁に着目して説明を行う。各桁の確定時に新たに step2 でフラグが立てられた位の数には””で、既にフラグが立っている位の桁には” でそれぞれ印をつけた。

実際には第 1 位の確定まで処理は続けられているが、この例では第 6 位が決定した時点で本来の順列に逆変換できており、以降の桁の確定で数が変化することは無い。

以上に示した手法によって 15 パズルでは全ての盤面に対して固有のハッシュ値を算出することができ、また逆関数によりハッシュ値を盤面情報に戻すことができる。これはリスト管理に用いるキー値として使用されるハッシュ値が盤面の情報を含んでい

表 4: 逆関数の動作

									説明
3	6	1	4	0	0	2	0	0	階乗進数
[3]	6	1	4	0	0	2	0	0	第 9 位確定、１位に step2 を行う
[3]	6	1	4	”0”	”0”	2	”0”	0+3	フラグをつけた桁だけ値を増加
[3]	7	1	6	”0”	”1”	5	”2”	3	残りの桁に step2 の実行後
3	[7]	1	6	’0’	’1’	5	’2’	3	次の上位桁 (第 8 位) の確定
”3”	[7]	”1”	6	’0’	’1’	5	’2’	3+2	１桁目に step2
”3”	[7]	”1”	6	’0’	’2’	6	’3’	5	残りの桁に step2
’3’	7	’[1]’	6	’0’	’2’	7	’4’	5	第 7 位の桁の確定後
’3’	7	’1’	[6]	’0’	’2’	8	’4’	5	第 6 位の桁の確定後
’3’	7	’1’	6	’0’	’2’	8	’4’	5	本来の盤面から得た順列

る事を意味するため 1 つの状態あたりの情報量を削減することが出来るようになる。また、ハッシュ値の衝突が発生しなくなるためリスト管理のための処理の記述が容易になる。

4.4 検討案を追加した並列 A*

本研究で使用した並列 A*アルゴリズムは基本的には HDA*と同様の実装を行った。すなわち、PRA*で使用されたハッシュ値ベースの探索空間分割と、TDS で使用された非同期通信による処理の独立化である。本研究でも既存研究でのそれらの手法を踏襲し、並列探索の性能を向上させている。一方で、HDA*とは異なり各端末が使用するメモリ量を抑えるためハッシュ関数として検討案中で述べた階乗進数変換および並列 A*に適した形での探索処理とリスト操作の処理順に変更している。実装した並列 A*の探索部分についての処理フローを図 14 に示す。

予測コスト x の状態群の探索が終了して解が発見できなかった場合、逐次 A*では次の予測コスト x+1 の探索を始めることになるが、並列 A*の場合どれかの端末内部で予測コスト x の状態群が無くなったとしても他の端末では予測コスト x の候補が調べ尽くされていない事がある。逐次 A*同様の動作を保証したい場合には全端末で予想コ

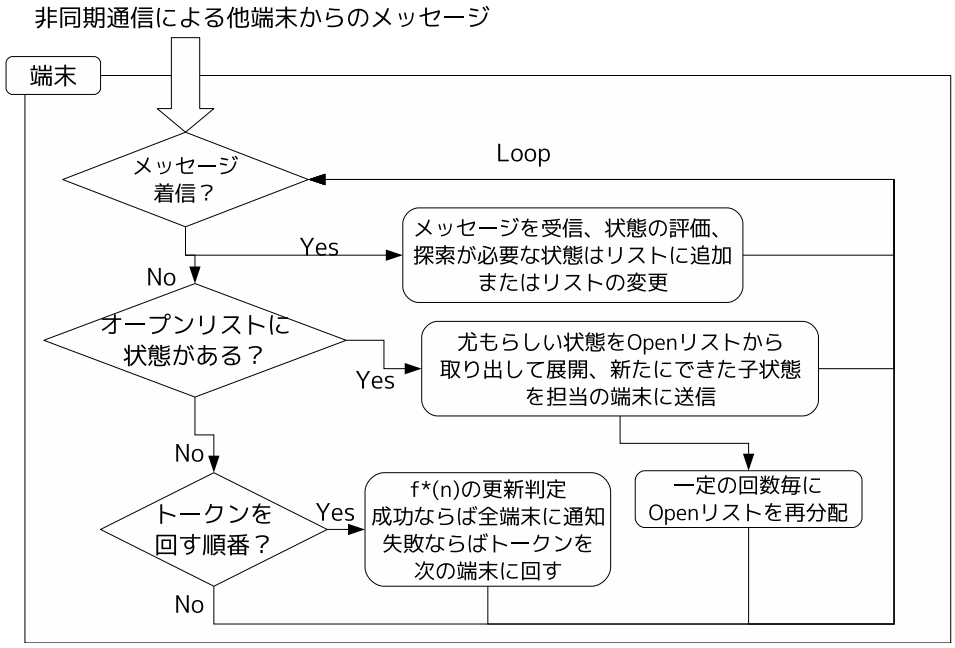

図 14: 並列 A*の各端末での処理

スト x の候補が調べ尽くされたことを判定する必要がある。

各端末が同期をとって探索を行っているならばその判定は容易である。しかし、各端末が独立している本検討手法では別の手法を用いなければならない。この手法は特に分散停止検出と呼ばれている。本研究で作成した並列探索では、並列探索を行っている端末を仮想的にリング状に配置されているととらえてトークンメッセージをリレーさせていくことでこの判定を実現した。

他の手法としてはエネルギーという概念を新たに追加し、データの送信に端末自身の持つエネルギーの一部を付随させるという手法がある。この方法では停止を判定する端末が一つ存在し、仕事が無くなった場合端末は保有していたエネルギーを管理者に送信する。管理者端末に全エネルギーが集まったとき、全ての端末で処理が停止していると判断できる。

あるいはこれら分散停止判定を行わず、各自で予測コストを繰り上げて探索を続けていく事も不可能ではない。受信した状態情報の中に保有していた情報よりも良い状態が存在した場合、リストの更新や状態の再探索が実行されるからである。ただしその場合には予測コストを独自の判断で引き上げるため、十分な判定基準が必要である。

5 評価

本論文では検討案であるデータの再配分を行う並列探索と取り入れない従来の並列探索における探索処理結果の比較評価を行う。

実験に使用した環境は表 5 の通りである。

表 5: 実験環境

OS	CentOS 5.4
名称	Intel-Core-i5-750
動作周波数	2.66Ghz(×20)
キャッシュ	L2:256KB×4, L3:8MB
メモリ	4.0GB
LAN	1000BASE-T
スイッチ	Prosafe GS724AT
使用言語	C++
通信ライブラリ	MPICH2 1.2.1
使用端末数	最大 8 台
演算並列数	1,4,8,16(8 台 ×2 プロセス)

例題は、解答盤面から空白をランダム方向に 100 ～ 300 回スライドさせる事で作成した。これは作成された盤面が必ず解に到達できる事を保証する為である。

4 種類の例題に提案手法を適用した。例題の難易度として逐次処理での探索回数を基準として使用する。今回実験で使用した問題の難易度は表 6 の通りである。

プログラムは C++で記述し、通信には MPI ライブラリを用いた。リスト操作の為に C++の STL である multimap を使用した。multimap 内部は 2 分木で構成されるため、保有する状態の数が多くなればなるほど 1 つの状態へのアクセスが遅くなってし

表 6: 逐次探索での探索結果

	問題 1	問題 2	問題 3	問題 4
探索回数	205600	719016	1412558	1878272
探索時間 (秒)	47.21	289.87	824.59	1553.18

表 7: open リスト修正回数

並列台数	問題 1	問題 2	問題 3	問題 4
1	3859	17870	25181	43538
4[study]	4452	19120	26108	45464
4[old]	4328	18436	26376	45000
8[study]	4264	19696	26976	46664
8[old]	4136	18952	26680	46456
16[study]	5030	21632	31200	50608
16[old]	4624	20448	27808	48336

まうが、候補を探索候補順に自動的に並べ替える事が可能であり、またハッシュテーブルと異なり余分なテーブル領域を必要としないので使用メモリ量を抑えるために採用した。

4、8 台の並列探索にはそれぞれ 1 端末に 1 プロセスを割り当てているが、16 台並列探索は 8 台の端末にそれぞれ 2 プロセスずつ割り当てて並列探索を行っているため、ネットワーク通信オーバーヘッド等で完全な端末 16 台での並列化とは若干結果が異なっている可能性があることを記しておく。

今回の実験では通信オーバーヘッドの削減の為に通常のメッセージの送信は 50 種の盤面の展開の後に作成された状態をハッシュ値にしたがってそれぞれの送信先にまとめて送信する。また再探索の頻度は、”探索要求”のメッセージ（再配分されたデータ）以外の状態の探索 1000 回につき、自身の open リストに格納されている状態の中で優先度の高い順番に最大 1000 までを再配分する。つまり全ての状態が再配分されていないということであるが、これは状態の展開によって新たに作成された状態情報が再配分された状態よりも高くなることが珍しくなく、それらについて余分な通信をすることなく展開処理を行わせるためである。

実験結果として、検討手法による状態の動的再配分により並列探索の効率が従来手法に比べて向上したかの比較として open リストの総修正回数を表 7 に示す。これは探索終了までに、各候補がそれぞれ open リストに保有していたデータが、他端末からのメッセージによって修正された回数である。他の端末から受信したメッセージの中には既に open リストに登録されている候補が混在している事がある。受信端末は重複する候補について、リスト内部のデータと照合してその探索優先度を比較し、より探索

Parallel number	4	8	16
Average time[study]	10.65	5.12	2.69
Max time[study]	10.68	5.18	2.76
Minimum time[study]	10.62	5.10	2.59
Average time[old]	10.54	5.08	2.68
Max time[old]	10.56	5.13	2.73
Minimum time[old]	10.52	5.05	2.58

表 8: 問題 1 の探索結果（秒）

Parallel number	4	8	16
Average time[study]	42.94	20.32	10.45
Max time[study]	43.43	20.40	10.62
Minimum time[study]	42.62	20.24	10.24
Average time[old]	42.75	19.87	10.04
Max time[old]	42.80	19.98	10.29
Minimum time[old]	42.72	19.75	9.76

表 9: 問題 2 の探索結果（秒）

Parallel number	4	8	16
Average time[study]	107.05	43.93	22.14
Max time[study]	107.96	44.48	22.53
Minimum time[study]	106.37	43.57	21.71
Average time[old]	104.64	43.45	20.17
Max time[old]	105.24	43.68	20.60
Minimum time[old]	104.26	43.24	19.81

表 10: 問題 3 の探索結果（秒）

Parallel number	4	8	16
Average time[study]	189.58	73.17	32.29
Max time[study]	190.99	73.76	32.88
Minimum time[study]	188.65	71.95	31.61
Average time[old]	184.17	72.63	31.69
Max time[old]	186.32	73.77	32.43
Minimum time[old]	182.09	70.95	30.56

表 11: 問題 4 の探索結果（秒）

優先度の高い情報を受信したと判断されるときリストのデータの修正を行って自身の処理の最適化を行う。修正されないまま探索処理を実行された場合、その後に展開される全ての状態およびそれらに対する処理はオーバーヘッドとなるため、早い段階で多くの候補についてこの修正が行われることが望ましい。

表 7 に示される結果より、同並列数での探索に着目して修正回数を比較すると、殆どの場合について検討手法の修正回数の方が従来手法を上回っていることを確認できる。並列数が大きくなればなるほど修正回数が大きくなっているのは、探索処理が多重化される割合が増えたことでに余分な処理の発生確率が上昇したためだと考えられる。すなわち検討手法による状態の動的配分によって探索優先度の高い候補が従来よりも早い段階で処理された結果、リストの修正がより多くなったものであり、並列探索の処理効率が上昇したと考えられる。

次に再分散オーバーヘッドの影響について、解到達までに各端末が必要とした並列探索処理の探索実行時間の比較結果を表 8~11 に示す。この表から、検討手法についても従来手法どおり、台数に比例した 1 台あたりの処理の減少から解到達までの処理時間は従来手法どおりの速度向上が見込めることが分かる。一方で、従来手法に比べて各端末での探索処理時間が若干の増大が見られる。この処理時間の悪化の原因は、再分散によって発生した処理オーバーヘッドであると考えられる。その内訳について考察する。通信のオーバーヘッドは非同期通信によって隠蔽されているので、この差は

表 12: 受信データの評価とリスト操作の時間 (秒)

並列台数	問題 1	問題 2	問題 3	問題 4
4[study]	0.94	7.32	32.22	74.73
4[old]	0.80	7.13	28.60	69.28
8[study]	0.32	3.10	9.96	25.14
8[old]	0.29	2.78	9.61	24.53
16[study]	0.19	1.70	5.21	9.10
16[old]	0.17	1.38	3.44	8.47

主にリストに対する処理の増加オーバーヘッドであると予想される。

そこで、各並列探索処理において受信データについてリスト操作の処理に使われた時間を表 12 に示す。この結果から、検討手法において、再分散によって受信データが増加したことでリストに対する処理の増加があったことを確認できる。また、この時間差を先の探索処理全体の悪化の時間差と比較するとほぼ一致している事が分かる。これにより、探索処理時間の悪化は増加したリスト操作の為の時間であると結論できる。

また、並列台数が少ないほどこの差が顕著である理由はリストに使用したデータ構造であると予想される。今回の実装ではリストには multimap と呼ばれる 2 分木のデータ構造を使用していた。つまり、並列台数が少ないほど各端末に割り当てられる探索空間が大きくなり、探索処理が進めば進むほどリストが大きくなるため 1 つのデータにアクセスする時間が大きくなっていく。例えば 4 台並列と 16 台並列では保有データ量はほぼ 4 倍になるため、探索終盤にはリストの木の深さが 2 段階ほど深くなっていると予想される。これは、1 つのデータに対する木のアクセスが 2 ステップ増加することを意味している。リストは頻繁にアクセスされるものであるため 1 つ 1 つのアクセス速度差は微々たるものであったとしても最終的には秒単位で確認できるほどのオーバーヘッドになったものだと思われる。よって、従来手法と同等の性能を引き出す為には、アクセス速度の解消のためにハッシュテーブルを用いるリスト管理機構を実装するなどでオーバヘッドを軽減する必要がある。

6　まとめ

本研究では解候補の分散を動的に行うことで、静的、確率的分散よりも並列探索処理が最適化され、処理が高速化できるかの検討を行う。しかし、単純な候補の動的分散では並列探索全体で見た場合に余分な処理が発生してしまう。そのため従来手法のハッシュ値に基づいたデータ分散によりこれを抑止した上で、問題の知識であるヒューリスティクス値などから探索優先度を考慮した候補の動的分散処理を行う並列探索を実装した。

検討手法と従来手法の比較実験の結果、探索優先度を考慮した明示的処理分散は確率的分散手法より最適な処理を行う事を確認した。一方で、従来では行われなかった状態の送受信とそれらのリスト管理のオーバーヘッドにより探索処理そのものの処理時間は若干の悪化した。これは特に、リスト操作に使用したデータ構造の影響が大きいと考えられる。そのため、今後は再配分による処理オーバーヘッドを削減し、探索空間の動的配分手法による並列探索処理の高速化について、更に検討を進めていくことが今後の課題である。

謝辞

本研究の為に多大なるご尽力をいただき、日頃から熱心なご指導を賜った名古屋工業大学の松尾啓志教授、津邑公暁准教授、齋藤彰一准教授、松井俊浩助教に深く感謝いたします。また、本研究の更に多くの助言、協力をしていただいた松尾・津邑研究室および齋藤研究室の皆様に深く感謝致します。特に、的確な助言と助力をしてくれた伊藤翼氏、加藤雄大氏、川東勇輝氏、熊崎宏樹氏に深く感謝致します。

参考文献

[1] Hart,P.E.; Nillsson,N.J.; Raphael,B.(1968). ”A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. IEEE Transactions on Systems Science and Cybernetics SSS4 (2):pp.100-107.

[2] Albert L. Zobrist.(1969). ”A Hashing Method with Applications for Game Playing”. Tech.Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin.

[3] A.Brungger, A.Marzetta, K.Fukuda, and J.Nievergelt.(1999) ”The Parallel Search Bench ZRAM and its Applications”. Annals of Operations Research, 90:45-63,1999

[4] M.Evett, J.Hendler, A.Mahanti, and D.Nau.(1995) ”PRA*: Massively Parallel Heuristic Search”. Journal of Parallel and Distributed Computing 25(2):133-143.

[5] J.W.Romein, H.E.Bal, J.Schaeffer, and A.Plaat.(2002). ”A Performance Analysis of Transposition-Table-Driven Scheduling in Distributed Search”. IEEE Transactions of Parallel and Distributed Systems 13(5):447-459.

[6] M.Helmert, P.Haslum, and J.Hoffmann.(2007). ”Frexible Abstraction Heuristics for Optimal Sequential Planning”. In Proceedings of ICAPS-07,176-183.

[7] A.Kishimoto, A.Fukunaga, and A.Botea.(2009). ”Scalable, Parallel Best-First Search for Optimal Sequential Planning”. In Proceedings of the 19th International Conference on Automated Planning and Scheduling(ICAPS-2009)

[8] 奥山晴彦　「C 言語による最新アルゴリズム事典」, 技術評論社,(1991),ISBN4-87408-414-1

[9] 白井良明　「人工知能の理論」, コロナ社,(1992),ISBN4-339-02549-6

[10] Dijkstra,E.W. (1959). ”A note on two problems in connexion with graph”. In Numerische mathematik, 1(1959),S.269-271

[11] Biggs.N, Lloyd.E, and Wilson.R. (1986). Graph Theory,1736-1936.Oxford University Press.