リンケージを考慮した遺伝的操作の並列化に関する提案

井上, はづき / INOUE, Haduki

(発行年 / Year)
2012-03-24
(学位授与年月日 / Date of Granted)
2012-03-24
(学位名 / Degree Name)
修士(理学)
(学位授与機関 / Degree Grantor)
法政大学 (Hosei University)

平成2011年度　修士論文

リンケージを考慮した遺伝的操作の並列化に関する提案

Parallelization of Genetic Operations that Takes Building-Block Linkage into Account

法政大学

情報科学研究科　情報科学専攻

佐藤　研究室

学籍番号：10T0004

氏名：井上（イノウエ） はづき（ハヅキ）

指導教授：佐藤　裕二

平成 24 年 2 月 17 日提出

目次

概要 1
1 はじめに 2
2 数独のルールと GA について 4
2.1 数独について 4
2.1.1 数独の概要 4
2.1.2 数独のルール 5
2.1.3 数独の解法 6
2.2 遺伝的アルゴリズム（Genetic Algorithms） 7
2.2.1 遺伝的アルゴリズムとは 7
2.2.2 GA の処理手順 7
3 GA による数独解法の精度向上案 10
3.1 ビルディングブロック問題への対処法 10
3.1.1 初期設定 10
3.1.2 評価計算 12
3.1.3 突然変異 13
3.1.4 交叉 14
3.2 局所探索能力向上の提案 16
4 遺伝子座を意識した並列化による性能向上案 18
4.1 遺伝的操作の並列化 18
4.1.1 評価計算の並列化 18
4.1.2 突然変異の並列化 19
4.1.3 交叉の並列化 19
5 実験 21
5.1 実験方法 21
5.2 精度実験 21
5.2.1 パラメータ 21
5.2.2 実験結果 21
5.3 性能実験 24
5.3.1 パラメータと実験環境 24
5.3.2 実験結果 24
5.4 考察 27
6 まとめ 29
参照文献 30

リンケージを考慮した遺伝的操作の並列化に関する提案

Parallelization of Genetic Operations that Takes Building-Block Linkage into Account

井上　はづき
Inoue Hazuki
法政大学情報科学研究科情報科学専攻
Email: haduki.inoue.ec@cis.hosei.ac.jp

概要

Previously, we proposed genetic operations that consider effective building blocks for using genetic algorithms (GA) to solve Sudoku puzzles, and proposed also a stronger local search function. In this paper, we propose performance enhancement by parallelization of genetic operations takes building-block linkage into account. In other words, parallelization of sub-blocks (regions) of Sudoku puzzles. Valid parallelization of genetic operations becomes a strong method of speed-up in conjunction with parallelization of individuals, since parallelization of genetic operations is not competing with parallelization of individuals. We show using multi-core processors and OpenMp that parallelization of genetic operations have positive effect of speed-up.

1　はじめに

本稿では遺伝的アルゴリズム(GA)の確率的探索手法としての有用性を実証するために、数独を GA に適用し、プログラムの精度と性能の向上を試みる。

現在の数独は、1979 年にアメリカ人建築家が匿名で考案したペンシルパズルの一つである。ペンシルパズルとは図で示された問題に対して、ペンで答えを徐々に書き込んでいくことによって解答するタイプのパズルのことである。数独は 2005 年にイギリスでブームになったのをきっかけに、日本を含め世界中で流行している。一般的な数独は 9×9 のマスで構成されている。数独では解答開始時からいくつかの数字が与えられており、初期配置と呼ばれるそれらの数字のない空白のマスすべてに、1～9 の数字を制約条件に従って埋めていく、制約充足問題に属するパズルである。制約条件とは行・列・リージョンと呼ばれる 3×3 の小ブロックすべてにおいて 1~9 の数字が重複しないことである。より難解な数独として、16×16 や 25×25 といったサイズの大きなものも存在する[1]。

一方、確率的探索手法の一つである GA は、組合せ最適化問題や NP 困難な問題などの様々な問題に適用可能であり、数独が属する制約充足問題にも強いとされている。GA とは 1960 年代に John Holland によって提案されたメタヒューリスティックアルゴリズムの一つで、交叉や突然変異などの生物の進化の過程を模倣した操作を行い、近似解を探索するアルゴリズムである。GA では容易に解が求まらないような大規模かつ複雑な問題に対し、比較的容易に最適解あるいは近似解を得られると考えられている。

そのため、数独に GA を適用した研究例はいくつか存在する。GA 単独での解法を試みた従来例では、初期配置数の多い簡単な数独は GA で解けることが証明されている。しかし、最適解に至るまでのステップ数（世代数）が莫大になっており、難解な問題ではほとんどが有効世代内に解を得られていない[2-4]。原因はいくつか考えられる。まず、GA のメインオペレーションである交叉が、数独の場合では有効な解の一部（ビルディングブロック）を破壊しやすいことが考えられる。交叉とは二つの染色体の一部分を交換する操作のことである。数独の場合行・列・リージョンそれぞれに対して制約条件を満たさなければならないため、適応度が同程度の個体同士で交叉を行っても、それぞれが有するビルディングブロックが一方は行方向で見た場合、もう一方が列方向で見た場合であると、結果として双方のビルディングブロックを破壊してしまうだけになる場合があるためである。また、他にも GA の局所探索（ローカルサーチ）能力が高くないこと、広大な解の探索範囲に局所解が多数存在し、局所解からの脱出が困難であること、などが考えられる。それらの問題に対して、従来研究では GA と Grammatical Evolution[3]や Cultural Algorithm[4]などの別のアルゴリズムを組み合わせることで、アルゴリズムの改善を図っている。

上記問題の対処として我々は、遺伝子座のリンケージを考慮した遺伝的操作を提案することにより、プログラムの精度向上を試みた。その結果、従来例では求解の難しかった問題においても、実行回数の半数で解を導き出すことに成功した[5,6]。

本稿では前述した精度向上手法に加えて、並列化による性能向上を行う。

遺伝的アルゴリズム(GA)を含む進化計算では、大規模な問題へ適応する場合に生じる膨大な計算量に対する解決方法が課題となっている。進化計算を高速化する手段の一つとして、並列化の研究は 1990 年頃から行われてきた[7-11]。研究が始まった当初は超並列計算機上での実装方法の研究が主流であったが、近年ではマルチコアプロセッサや GPU を用いた並列化の研究も盛んになってきている[12-15]。現在は進化計算向けのベンチマークテストを用いたものが中心であるが、マルチコアプロセッサや GPU が一般の PC にも普及され始め、比較的安価に利用できるようになったことが、研究が盛んになってきている要因の一つと考えられる。マルチコア型の高性能マイクロプロセッサ型の計算機では、2,4,8 と言った演算装置を有し、マルチスレッドプログラミングにより、並列型の進化計算の実現が比較的容易に実現できる[13]。一般に GA の並列化を行う場合、マスタースレーブ方式などの個体レベルでの並列化を行うことが多い[16]。GA におけるマスタースレーブ方式とは、マスターで個体群を生成し、個体を各スレーブへ割り当て、遺伝的操作や評価計算をスレーブにて行う方式のことである。これは構造が単純で並列化の効果が得やすいため、並列化の基本と言える。

本稿では前述した個体並列化と組合せて利用できる並列化手法を提案する。その手法とは、リンケージを考慮した遺伝的操作の並列化である。また、本稿では提案手法をマルチコアプロセッサ上で OpenMp を用いて実現し、検討を行う[18]。

2　数独のルールと GA について

2.1 数独について

2.1.1　数独の概要

数独の基となったものに、18 世紀にスイスの数学者 Leonhard Euler が考案した、ラテン方陣と呼ばれるものがある。ラテン方陣とは、*n* 個の要素を *n*×*n* の行列に、各要素が各行および各列に 1 回だけ現れるように並べたものである。Fig.1 にラテン方陣の例を示す。

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Fig. 1 ラテン方陣

現在の数独は、アメリカ人建築家 Howard Garns が 9×9 のラテン方陣に 3×3 のブロックという制限を加え、パズルとしたものである。これは Number Place の名前で 1979 年にニューヨークの出版社デル・マガジン社から初めて出版された。

日本には、ニコリの『月刊ニコリスト』1984 年 4 月号で、「数字は独身に限る」の題で初めて紹介された。命名者はニコリ社長の鍜治真起である。1988 年 4 月 1 日、ニコリから単行本『ペンシルパズル本 6・数独 1』が刊行された際、略称として「数独」という名称が登場した。以後数年間は「数字は独身に限る」の方が引き続き正式名称だったが、1992 年 3 月 1 日発行の『パズル通信ニコリ 37 号 はる分』より、「数独」が正式名称となった。

世界的な流行は、1997 年に 59 歳のニュージーランド人 Wayne Gould が日本の書店で数独の本を手にとったことに始まる。Gould は 6 年後数独をコンピュータで自動生成するプログラムを作る事に成功しイギリスの新聞・タイムズに売り込み、2004 年 11 月 12 日から Su Doku の名で連載を開始した。2005 年 4 月から 5 月にかけてブームに火が付き、インデペンデント、ガーディアン、ザ・サン、デイリー・ミラーなどイギリスの主要日刊紙に軒並み掲載されるという状況になった。2005 年 7 月 1 日にはテレビ局スカイ・ワンが、数独をテーマにした初のテレビ番組を放映。イギリスの人気は他国にも飛び火し、パズルとしては 1980 年頃のルービック・キューブ以来の大流行と言われた[1]。

2.1.2　数独のルール

一般的な数独の問題は 9×9 のマスから構成されており、1～9 の数字がいくつかのマスに予め与えられている。出題の段階で与えられた数字を初期配置と呼び、その数字の数や配置によって難易度が変化する。基本的には初期配置の数字が少ないほど、解となり得る組み合わせが増えるため、難易度は高くなる。現在、解が一意に決まる最小の初期配置数は 17 であると言われている[18]。

数独は初期配置以外の空白となっている全てのマスを、1～9 の数字で埋めることが解答となる。その際、縦・横の各列、Fig.2 において太枠で囲われた 3×3 の小ブロック（リージョン）の全てにおいて、1～9 の数字が重複していないことが条件となる。

Fig.3 において、数独の左図に問題例、右図に解答例を示す。

また、数独には多くのバリエーションが存在している。16×16、25×25 など大きさを変更したものや、対角線においても数字を重複させない、色分けされた同色の 9 マスの中でも数字を重複させない、などの条件を付加したものなどがある[1]。

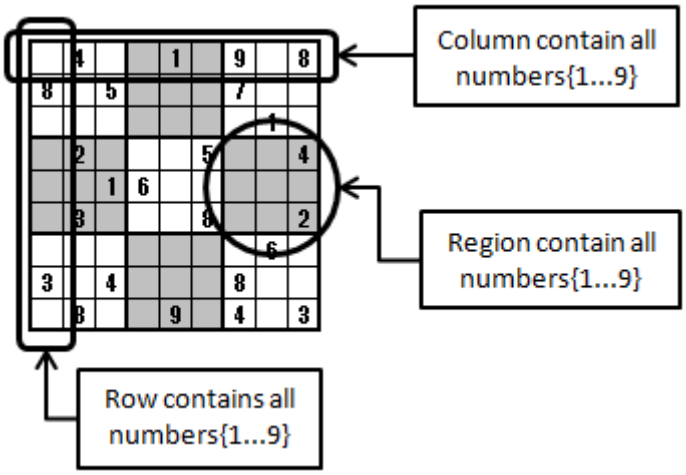

Fig. 2 数独のルール説明

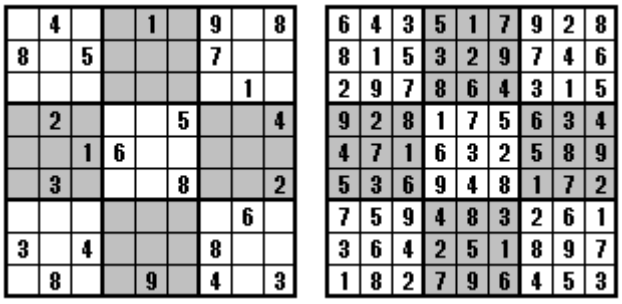

Fig. 3　数独の問題と解答例（初期配置数:24）

2.1.3　数独の解法

人間が数独を解く際の方法を以下で説明する。

初級の問題においては、基本的な二つの方式によって求解が可能である。一つには、ある数字に注目してその数字が入るマスを決定する方法である。もう一つは、一つのマスに注目して、そのマスを含む行・列・リージョンに含まれる数字を調べることで、入る数字を特定する方法である。Fig.4 の左図では数字に注目した解法、右図ではマスに注目した解法の説明を行う。

まず初めに、数字に注目した解法だが、図では{4}に注目している。図中で 3 つある全ての 4 に対して、それぞれの行・列・リージョンにはこれ以上 4 が入ることはない。Fig.4 の左図において、4 が入ることのないマスについてはグレーで色分けしている。すなわち、4 が入るのは白色の空マスということになる。そこで、左図の丸で囲ったリージョンを見ると、白色の空マスが一つしかない。よって、そのマスに 4 が入るということが分かる。

次に、マスに注目した解法だが、Fig.4 の右図では丸で囲われたマスに注目している。そのマスを含む行・列・リージョンをグレーで色付けしている。グレーで色付けされた行には{1, 3, 4, 5, 6, 8}、列には{3, 8, 9}、リージョンには{3, 4, 7, 8, 9}の数字が含まれていることがわかる。そこで、行・列・リージョン全ての要素の論理和をとると、{1, 3, 4, 5, 6, 7, 8, 9}となる。したがって、このマスには要素の和に含まれていない{2}が入ることが特定される。

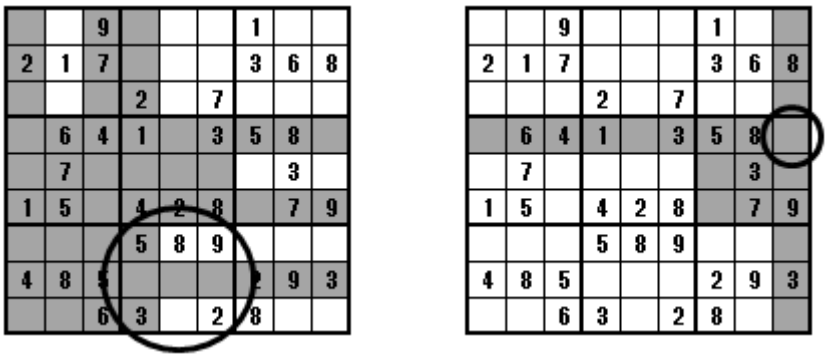

Fig. 4 数独の解法例

また、難解な問題においては、初期配置の段階から上記の解法が使えないこともある。その場合、上記の方法の応用で数字の入る場所を特定していく方法がある。それらの手法を用いても求解に至らない場合、取り得る数字の候補が少ないマスにおいて、候補の中から適当な数字を当てはめ、それを仮に正解として解いていき、矛盾が出なければ仮においた数字が確定となり、解答に至る方法など、さまざまなヒューリスティックな解法が存在する。

2.2 遺伝的アルゴリズム（Genetic Algorithms）

2.2.1　遺伝的アルゴリズムとは

GA とは 1960 年代に John Holland によって提案された、生物の進化を模倣したアルゴリズムのことで、最適化問題の解の探索手法として用いられる他、人工生命や複雑適応系の適応メカニズムとして用いられている。

遺伝子とは生物の情報を書き込まれた設計図であり、それは生物の長い歴史の中で自然によって徐々に編集されてきたものである。遺伝子という情報の流れに着目するならば、進化とは遺伝子の組合せの可能性の中で、その組合せを環境などの刺激を受け最適化するプロセスに他ならない。このことを現実問題の最適化手法として利用したのが、「遺伝的アルゴリズム(Genetic Algorithms)」である[19]。

GA では適用したい問題を記号で表現した遺伝子へと落とし込み、その遺伝子に対して選択や、交叉、突然変異と呼ばれる遺伝的操作を繰り返すことによって、環境への適応度が高い遺伝子の組合せを探索していく。遺伝子の適用方法を考えることにより、組合せ最適化問題や NP 困難な問題など様々な問題へ適用することが可能である。

2.2.2　GA の処理手順

GA では通常、個体は染色体を一つだけ持っていると想定する。染色体は遺伝子で表され、遺伝子が複数並んだ遺伝子配列として構成されている。遺伝子は記号で表されるが、具体的には{0, 1}や実数値などで表現される。各遺伝子の位置を遺伝子座といい、その位置にある遺伝子はある特有の性質を指定する。例えば人間の遺伝子を考えるとして、1 番目の遺伝子は髪の色、2 番目の遺伝子は目の色を表すといった形である。また、遺伝子座が取り得る遺伝子候補のことを対立遺伝子という。例えば遺伝子が髪の色を表しており、遺伝子候補、つまり髪の色の候補が黒色か茶色という二つのうちどちらかを取るならば、黒色と茶色が対立遺伝子となる。

以上のような染色体から遺伝子情報を解読して表現することによって、個体が誕生する。個体の集合のことを個体群といい、GA では個体群に対して選択・交叉・突然変異などの操作を繰り返していくことが最適解の探索を行う。また、個体群に行う一連の操作をまとめて一世代と表し、操作する世代（個体）を親世代（個体）、操作の結果生まれた世代を子世代（個体）と呼ぶ。また、各個体は環境への適合度や他個体との競争によって優劣がつけられる。その評価値を適応度といい、評価値の算出式を評価関数という。

例として、最適化問題を解く場合、終了条件を設定し、Fig.5 に示されるようにまず個体の集合である個体群を生成した後、評価、選択、交叉、突然変異の手順を繰り返すことによって、適応度の高い染色体を探索する。一般的には最適解が探索された時や、一定世代数が経過した時などが終了条件となる。

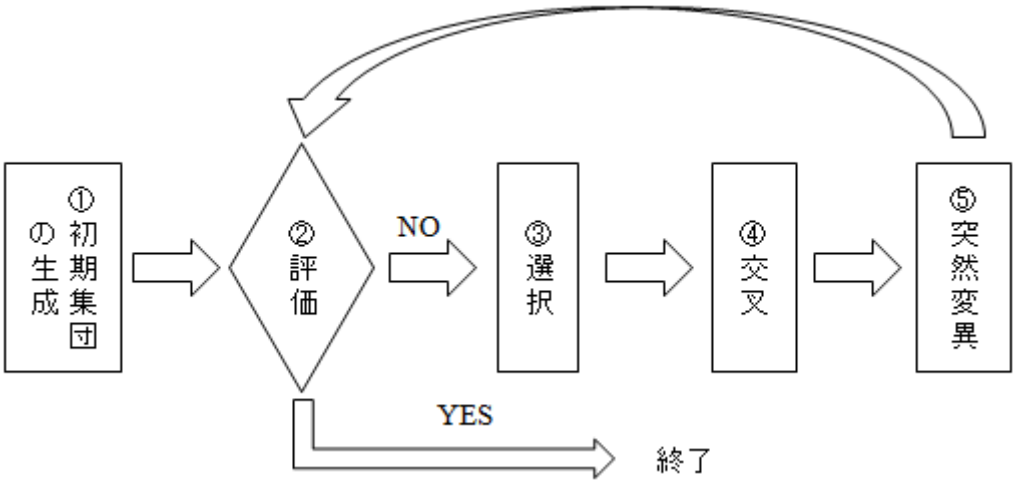

Fig. 5 GA の操作手順

Fig.5 に示される各操作手順について、下記で詳しく説明する。

（1）初期集団の生成

初期集団の生成では、設定した個体数分の個体（染色体）を生成する。この時、個体の遺伝子は遺伝子候補の中からランダムに値を決定する。

（2）評価

評価では、個体の遺伝子から環境における各個体の適応度を求め、各個体の評価値を計算する。評価関数と呼ばれる管巣を用いて個体毎に求める。一般的に、より優秀な個体の適応度が高くなるように、評価関数を設計する。

評価関数によっては探索が進むと個体同士の適応度の差が少なくなり、淘汰圧が弱くなる場合がある。これを防ぐために、スケーリングと呼ばれる手法を用いることがある。これは適応度に定数を掛ける、あるいは適応度を何乗かすることによって、個体間の適応度の差が強調されるようにする手法のことである。

（3）選択

自然界において適応度の高い個体が子供を多く残すように、GA でも適応度の高い個体を選択することによって、適応度の低い個体を淘汰する必要がある。適応度の低い個体は淘汰されるが、適応度の高い個体は複製されるため、個体の総数は一定である。一般的に、評価関数を用いて各々の個体の評価値を決定した後、それを基に選択交配を行う。

選択のアルゴリズムはいくつか存在するが、ルーレット方式が最も単純で一般的であると言われている[19]。ルーレット方式とは、適応度に比例した割合で個体を選択する方法である。適応度の高い個体ほど選択される確率が高くなるが、適応度の低い個体であっても、選択される可能性は残る。また、選択にはルーレット方式の他にも、適応度の順位によって確率を決定するランク方式、個体群の中から決められた数の個体を取り出し、その中か

らもっとも適応度の高い個体を選択するトーナメント方式などがある。いずれの方式を用いるかは、適用する問題により異なる。

(4) 交叉

交叉とはGAの代表的な遺伝的操作のひとつであり、二つの染色体の一部分を交換し、新しい個体を生成する操作のことである。これは生物が交配によって子孫を残すことを模倣した遺伝的操作で，個体の遺伝子をランダムな交叉点で区切り、双方の染色体の一部ずつを入れ替え子孫の染色体を作る操作であり、「一点交叉」「多点交叉」「一様交叉」などの種類がある[19]。

一点交叉は染色体上に一ヶ所だけ分離点を決めて交叉を行うものである。つまり、染色体を決められた分離点で分けた前半部分が親個体A、後半部分が親個体Bである AB という子個体と、前半部分が親個体B、後半部分が親個体Aの BA という子個体が生成される。多点交叉は複数の分離点で交叉するもので、一般に二点で交叉する「二点交叉」がよく使われている。また、一様交叉は任意数の交差点によって交叉させる方法である。どの方法でも分離の場所はランダムに決められる。

前述の選択により増えた適合度の高い個体郡に対し、交叉は進化を進める為にそれぞれの個体が持つ染色体の一部を組み替えることにより、最適な解を得る可能性が高まる。染色体の交叉は選択交配を行う個体対が決定された後に行う。

(5) 突然変異

交叉と同様に、突然変異もGAの代表的な遺伝的操作の一つである。突然変異とはランダムに選ばれた遺伝子座の遺伝子を他の対立遺伝子に変えることである。突然変異も生物の進化に見られる現象を模倣した操作であり、例えば先祖代々黒色の毛を持っていた獣の集団に、突如として白色の毛を持つ獣が現れることなどが、突然変異と呼ばれる。

GAにおける突然変異の例としては、0と1で表される染色体において、ある遺伝子に突然変異が起こると0ならば1、1ならば0に反転することになる。{0, 1, 1, 0, 1, 1}という染色体の全ての遺伝子に突然変異が起こったと仮定すると、染色体は{1, 0, 0, 1, 0, 0}となる。ただし、突然変異が起こるか否かは突然変異率にしたがう。

交叉によって生まれる個体は親個体の一部分のコピーである為に、交叉のみでは個体の親に依存する限られた範囲でしか子個体が生成されず、十分な進化を得ることが出来ない事がある。突然変異を行う事で、個体集団の多様性を生み出す可能性が上がり、局所解に陥ることを防ぐ効果がある.

3　GA による数独解法の精度向上案

GA を数独に適用した従来例を見ると、最適解が求まるまでにかかる世代数が莫大になっている [2]。これは問題が難解になり、初期配置数が少なくなるほど顕著に表れている。数独において初期配置数が少なくなるとは、解答者が埋めなければならない空白マスが増え、解の探索空間が広がることを表わしている。

解の探索空間が広い問題に対して、確率的探索手法である GA は一般に有効であるとされている。一方、今回の数独の問題のように探索空間が広くなるにも関わらず、初期探索空間に依存的な局所解が多数存在しているような問題に関しては、GA が有効に機能しているとは言えない。

GA が有効に機能しない原因として、いくつかの可能性が考えられる。もっとも有力な原因の一つとして考えられるのが、数独のメインオペレーションの一つである交叉が有効に機能していない可能性である。これは GA を数独に適用した場合、交叉により有効な解の一部（ビルディングブロック）を破壊してしまっているためであると考える。数独は行や列、リージョン毎に 1〜9 の数字を揃えていくパズルであるが、その枠を無視して解の一部分を交換する操作である交叉を行うことによって、一行または一列という範囲で部分的に正解が得られていた場合、その有効な解の並びを破壊してしまうことになるためである。よって、提案手法ではビルディングブロックを破壊しないための工夫として、リージョンに着目した手法を提案した。

また、GA は一般にグローバルサーチの性能は高いが、ローカルサーチの性能は高くないとされている。従来手法において、最適解とその近辺の評価値にあまり差が見られないのは、個体群が最適解の近くまで来ながら、なかなか最適解へとたどり着けず、いたずらに世代数を増やしてしまっている可能性が考えられる。よって、提案手法ではローカルサーチの機能を加えることで、さらなる数独解法の精度向上を試みた。

以上の問題点を改善するための工夫を、以下で説明する。

3.1 ビルディングブロック問題への対処法

3.1.1　初期設定

数独では解答の開始時からいくつかの数字が初期配置として与えられている。したがって初期値を決める時には、初期配置以外の空白のマスを 1〜9 の数字でランダムに埋めることになる。また、数独には「全てのリージョン内で数字が重複してはいけない」という制約条件がある。提案手法では、初期値を決定する際にこの条件を初めから満たすようにする。上記の条件を初めから満たし、各操作をリージョン単位で行い、この条件を満たし続けることによって、リージョン毎のビルディングブロックを崩され難くすると共に、「全ての行および列内で数字が重複してはいけない」と言う条件のみを考慮すれば良くなるため、探索及び評価の処理を単純化することが出来る。

「全てのリージョン内で数字が重複してはいけない」という条件を満たすため、初期値として空白に入れることの出来る数字は、1〜9 の中でも、リージョン内の初期配置以外の数字となる。また、同一リージョン内では、一度ランダムに選択した数字は使い捨て、同一の数字がリージョン内に表れないようにする必要がある。

初期設定の例を、Fig.6 に示す。Fig.6 左図では、問題例と各リージョンにおいて空白マスに入る可能性のある数字を示しており、Fig.6 右図ではランダムに初期値を設定した結果を示している。

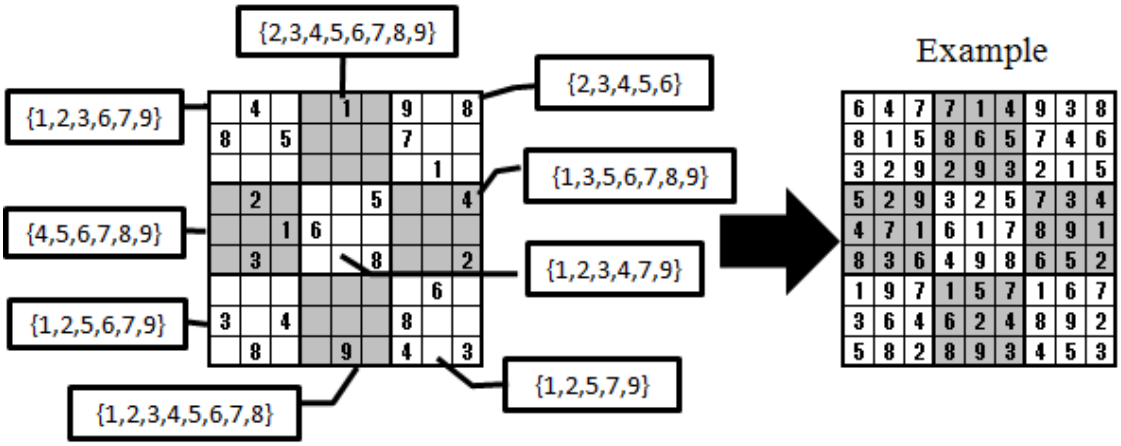

Fig. 6 初期設定の説明

初期値を設定する具体例として、Fig.6 左図の問題例における左上のリージョンに注目する。このリージョンには初期配置として{4, 5, 8}の数字があらかじめ置かれている。したがって、このリージョンの空白マスに入る数字は{4, 5, 8}以外の数字である{1, 2, 3, 6, 7, 9}になることが分かる。提案手法では、それら 6 つの数字の中から、ランダムに選択した数字を左上の空白マスから順に埋めていく。その際、一度使った数字を二度以上使わないようにしなければならない。

初期値を決めるプログラムの詳しい挙動は以下のようになる。初めに左上のマスに{1, 2, 3, 6, 7, 9}の中からランダムに選んだ数字である{6}を入れる。右へシフトすると初期配置の{4}があるので、そのマスは飛ばす。さらに右へシフトし、初期配置と最初に使った 6 以外の数字である{1, 2, 3, 7, 9}の中から、ランダムに選んだ{7}を入れる。次に下の行へシフトする。二行の一列目には初期配置の{8}があるので、そのマスは何もせずに飛ばし、その隣のマスへ移る。残っている数字は{1, 2, 3, 9}の 4 つである。この中からランダムに選択した{1}を空白マスへ入れる。次のマスには初期配置{5}があるので、そのマスも飛ばし、三行目に移る。三行目には初期配置がひとつも存在しないので、一列目から順に{2, 3, 9}の中から数字をランダムに選択していくことで、結果として Fig.6 右図に示す染色体が生成される。

また、上記手法を全てのリージョンで繰り返すことによって、Fig.6 右図が示す通り、各リージョンで 1〜9 の数字が重複することがなく、かつ、初期配置がそのまま継承された個体を生成することが出来る。

3.1.2 評価計算

評価関数は、「一行中で数字が重複しない」「一列中で数字が重複しない」という二つの条件を基に、各行列の要素数をそれぞれの評価値とし、行列の評価値を全て加算した値を個体の評価値とする(1)。ただし、9×9 マスの数独における「各マスには 1~9 の数字しか入らない」という制約条件から、各列の要素は 1〜9 に限る(2)(3)。

$$f(x)=\sum_{i=1}^{9} g_i(x)+\sum_{j=1}^{9} g_j(x) \tag{1}$$

$$g_i(x)=\left|x_i\right| \tag{2}$$

$$g_j(x)=\left|x_j\right| \tag{3}$$

Fig.7 では行列毎の評価計算の仕方を説明している。図中の数独パズル右側に記した数字がその行の評価値であり、パズル下方に記した数字が列の評価値である。

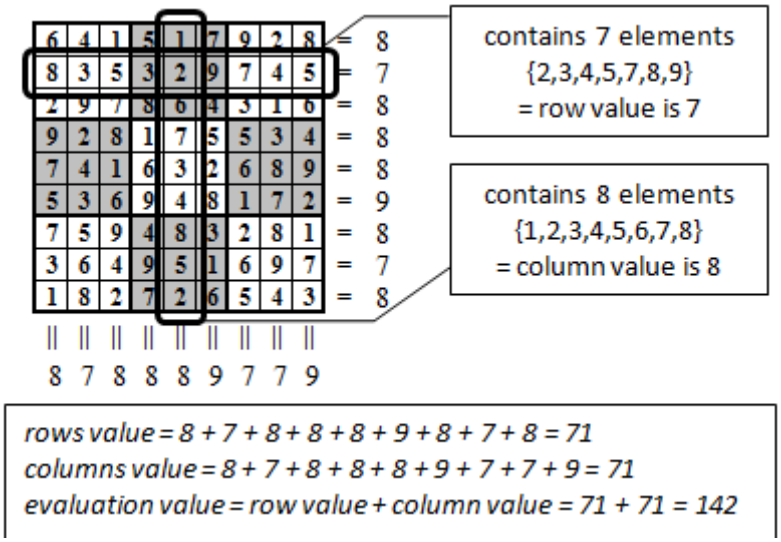

Fig. 7 評価計算の説明図

評価計算の具体的な計算例を以下で示す。

Fig.7 において四角で囲まれている行には、{ 2, 3, 4, 5, 7, 8, 9 }の 7 つの数字が含まれている。すなわち、要素数は 7 である。行の要素数がその行の評価値となるため、四角で囲まれた行の評価値は 7 となる。同じように、四角で囲まれた列を見てみると、列に含まれる要素は{1, 2, 3, 4, 5, 6, 7, 8}であり、要素数は 8 である。したがってこの列の評価値は 8 となる。このように算出した全ての行列の評価値を合算したものが、個体の評価値となる。Fig.7 の例では、行の評価値の合計が 71、列の評価値の合計も 71 であるため、個体の評価値は 142 である。

3.1.3 突然変異

突然変異では突然変異率にしたがって、リージョン内で二つの数字をランダムに選択し、その二つの数字の位置を交換するという操作を、全てのリージョンに対して行う。突然変異率とは突然変異が起こる確率のことで、提案手法において突然変異率は、個体毎ではなくリージョン毎に対して掛かるものとする。

突然変異をリージョン内で行い、かつ二つの数字の数字の位置を交換するという操作にすることにより、リージョン内で数字が重複してしまい、初期設定で満たした「全てのリージョン内で数字が重複してはいけない」という条件が崩れることを避ける。また、前提である数独の問題を崩さないために、選択されるランダム数字の中に、初期配置が入ることは避ける。

Fig.8 では突然変異の様子を示している。図では個体全体ではなく、一つのリージョンを抜き出した図になっており、左図が操作前、右図が操作後のリージョンを表わしている。また、灰色で示された数字は初期配置である。

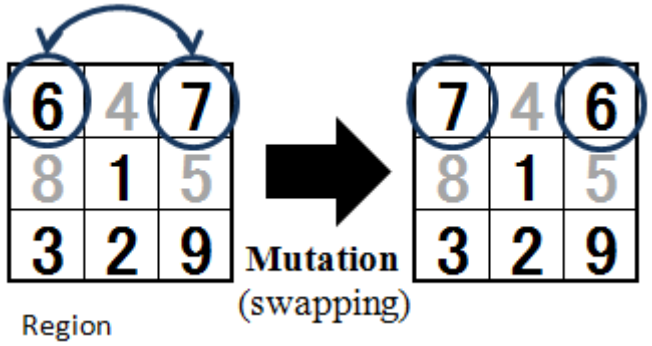

Fig. 8 突然変異の説明

Fig.8 の例では、リージョン内に 3 つの初期配置{4, 5, 8}が存在している。そのため、二つの数字をランダムに選択する際には、上記の初期配置以外の数字を除く{1, 2, 3, 6, 7, 9}の中から選択する必要がある。図中ではリージョン内で(1,1)(1,3)の位置にある 6 と 7 の二つの数

字を選択している。したがって突然変異を行うと、左図では一行目が左から{6, 4, 7}と並んでいた数字が、操作後である右図では{7, 4, 6}と並び変えられている。

上記のような交換を一度終えると、次のリージョンに移る。そして突然変異率にしたがって突然変異を行うか否かを決め、突然変異を行う場合は二つの数字の交換を行う。この操作を全てのリージョンに対して行う。

3.1.4 交叉

交叉とは染色体の一部を別の染色体と交換する操作である。提案手法では Fig.9 で示すように、二つの親個体から二つの子個体を生成している。Fig.9 では Parent1 の個体を白色、Parent2 の個体を灰色で表わしている。

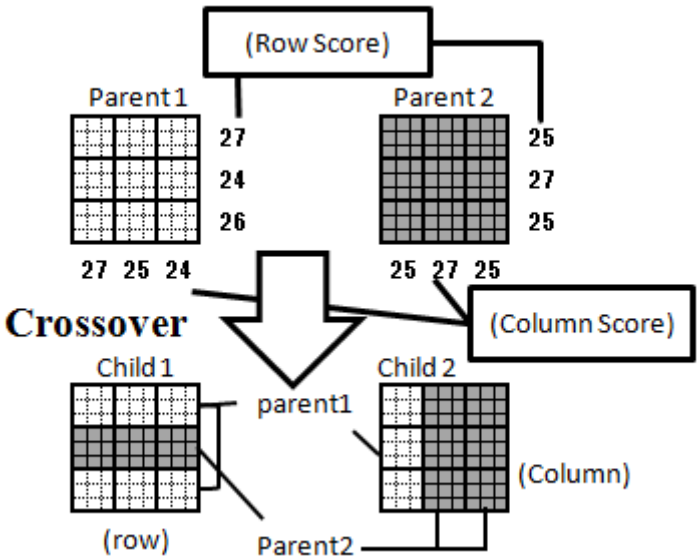

Fig. 9 交叉の説明図

生成される子個体はそれぞれ、行方向の評価値を三分割して比較し分割された部分ごとにより高い評価値の親個体の部分解を継承した個体と、同じように列方向の評価値を三分割して比較・継承した個体の二つとなる。行列の比較・継承は個体を 3 分割、つまり 3 行（列）毎に操作を行うことにより、リージョンが崩れ、初期設定で満たした「全てのリージョン内で数字が重複してはいけない」という条件が破られることを防ぎ、また、リージョンにおけるビルディングブロックを守る。また、この交叉では親個体の比較を行毎で行った結果を Child1 へ、列毎で行った結果を Child2 へと、2 パターンで比較・継承を行っているため、行におけるビルディングブロックの破壊は Child1 が防ぎ、列におけるビルディングブロックの破壊は Child2 が防ぐことが可能である。

数独の問題は「行・列・リージョン内で数字が重複しない」ことが制約条件であるため、前述したようにリージョンのビルディングブロック、行のビルディングブロック、列のビルディングブロックを破壊しない操作であれば、数独において考えられる全てのビルディングブロックの破壊を防ぐことが出来ると考える。

交叉の詳細な具体例を、Fig.10 に図示する。Fig.10 では二つの親個体 Parent1、Parent2 から Child1 へ、行方向で見た場合の比較・継承を行う手順を示している。また、図中では Parent1 の染色体を青色、Parent2 の場合は赤色で表している。

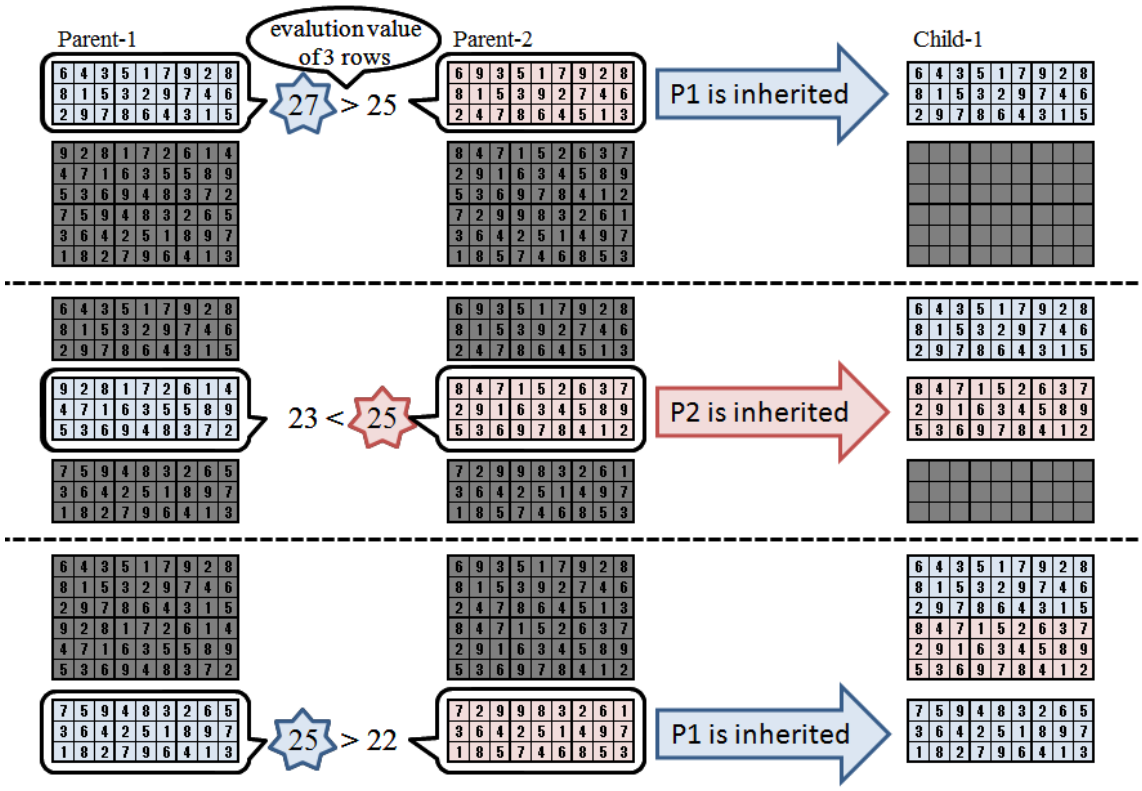

Fig. 10 交叉の詳細説明図

Fig.10 ではまず、上から三行分の部分解を比較している。ここで Parent1 の部分解の評価値は 27、Parent2 の評価値は 25 となっており、Parent1 の評価値の方が高い。したがって、Child1 の上から三行部分に継承されるのは Parent1 の部分解となる。同じように中段の部分解では Parent1 の評価値が 23 で Parent2 の評価値が 25 であるため、Parent2 の部分解が継承され、下段の部分解では Parent1 の評価値が 25 で Parent2 の評価値が 22 であるため、Parent1 の部分解が継承されている。したがって、行方向から部分解を比較した Child1 は、上から Parent1, Parent2, Parent1 の部分解を継承されたこととなる。

Fig.10 では示されていないが、列方向での比較・継承も同様にして行うことが出来る。Fig.9 の例では左から評価値が高い親個体が Parent1, Parent2 , Parent2 となっているため、子個体も左から Parent1, Parent2 , Parent2 の部分解を継承している。

また、本手法において、Parent1 と Parent2 の評価値 25 対 25 など同値になった場合、Parent1 の染色体が継承される。同値になった場合にランダムではなく恒常的に Parent1 を選択する理由は、個体の順序は世代が新しくなる毎に、選択においてランダムに変更されているため、Parent1 となる個体が変わらず交叉が上手く機能しないという事態にはならず、単に処理が増えてしまうだけとなってしまうためである。

3.2 局所探索能力向上の提案

一般に GA はグローバルサーチに比べ、ローカルサーチの性能が高くないと考えられている。局所における探索能力が低いと、個体群が最適解の付近まで近付いていても、最適解に到達するまでにより多くの世代数が必要となる。そのため、提案手法では突然変異の際に生成する子供の数を、設定した個体数の整数倍の子供候補を生成する。それにより、生成される子供のバリエーションを増やし、局所における探索能力を向上させる。

本手法では個体数分以上に生成した子供候補の中から次世代へ残す子供を選択する際は、単純に評価値の高いものから順に選択することとする。

Fig.11 では、広大な探索空間の中から、最適解周辺の極狭い空間を抜き出して、子供の生成時のイメージを示している。左図が一つの親個体から一つの子個体を生成する際のイメージ図、右図が一つの親個体から三つの子個体を生成する際のイメージ図である。黒点が親個体、白点が生成される子個体、赤点が最適解を示している。

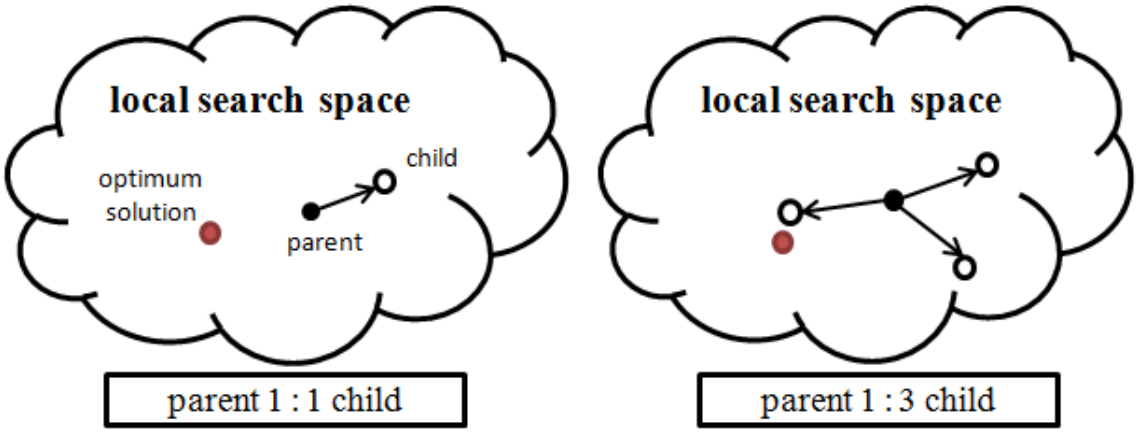

Fig. 11 ローカルサーチのイメージ図

Fig.11 左図が示すように、親個体一つに対し子個体も一つの場合、親個体の近くに最適解があるにも関わらず、最適解から離れた方向に子個体が生成されてしまう場合がある。そのため、最適解に辿り着くまでに大きく回り道をしてしまうことがあり、また、最悪の場合は最適解からかけ離れた方向へ個体群が進化することもある。一方、Fig.11 右図で示されるように、一つの親個体に対して子個体が複数生成される場合、生成される子供の方向が分散されるため、親個体よりも最適解に近い個体が生成され易くなる。したがって、親個体の整数倍生成された子供候補の中から、評価値の高い個体を選ぶことにより、ローカルサーチの性能が上がり、より少ない世代数で最適解を探索することが出来るようになる。

突然変異で生成された子供候補の中から子個体を選択する際は、生成された子供候補の評価計算を行い、算出された評価値を基にソートを行う。ソートされた子供候補の中から、評価値の高い順に子個体を選択していき、個体数分の子個体が選択された時点で処理は終了とする。

4　遺伝子座を意識した並列化による性能向上案

GA を含む進化計算の課題の一つとして、大規模な問題へ適応する場合に生じる膨大な計算量の解消がある。本章では上記の課題を解消するために、並列化による高速化を試みる。GA の並列化には様々な手法が存在するが、最も基本的なものの一つとして、個体の並列化がある[15]。個体の並列化とはマスターで生成した個体群を個体毎にスレーブへ割り当て、遺伝的操作や評価計算を行う方式である。個体の並列化は構造が単純で並列化しやすいため、比較的容易に並列化の効果を得やすい。また、GA による数独解法においても、個体の並列化により一定の高速化に成功している[20]。

本稿では個体の並列化と併せて使用出来るような並列化の手法として、遺伝子座のリンケージを考慮した遺伝的操作の並列化を提案、検討を行う。個体の並列化との併用を前提とした手法を提案する理由は以下の二点にある。まず一点目は、提案した手法において高速化の効果が僅かであったとしても、異なる並列化手法と併用出来れば乗算の効果が得られるため、結果としてかなり大きな効果が得られると考えるからである。二点目は、前述したように個体並列化自体が基本的な並列手法のひとつであり、単純な構造であるため他手法と併用しやすいと考えるからである。

また、本研究では並列化の実装に関して、OpenMP を用いている。OpenMP とは並列プログラミングの標準 API である[21,22]。並列化の入門として比較的導入しやすいため、本研究では OpneMP を用いて実装を行っている。

4.1 遺伝的操作の並列化

4.1.1　評価計算の並列化

評価計算では一行一列毎に評価値を算出し、その合計を計算している。なお、各行列の評価値は当該行列の要素数と同じである。実験で用いている数独パズルは 9×9 の構成であるため、行評価値の計算を 9 回、列評価値の計算を 9 回、計 18 回の評価値計算と、その合算を行っている。したがって評価計算を並列処理するために、まずは行列毎である 18 の部分に分割することを考える。18 の行列それぞれで評価計算が行われるが、本手法では Fig.12 で示されるように、一行一列の計算を同一スレッドで行う。Fig.12 では左が 1 番目のスレッド、右が二番目のスレッドでの処理を表わしている。OpenMp では一度並列処理が終了すると生成したスレッドを破棄し、再び並列処理を行うときに再度スレッドを生成する。そのため、各行列の評価計算を別スレッドへ分けることにより、並列化の効果よりもスレッドの生成コストの方が勝ってしまうことを避けるため、一行一列の計算を同一スレッドで行う。

また、各行列の合計値は、各スレッドにおいて、合計評価値への加算を不可分的に実行、つまり、同時参照を行わないように評価値を加算することにより行う。

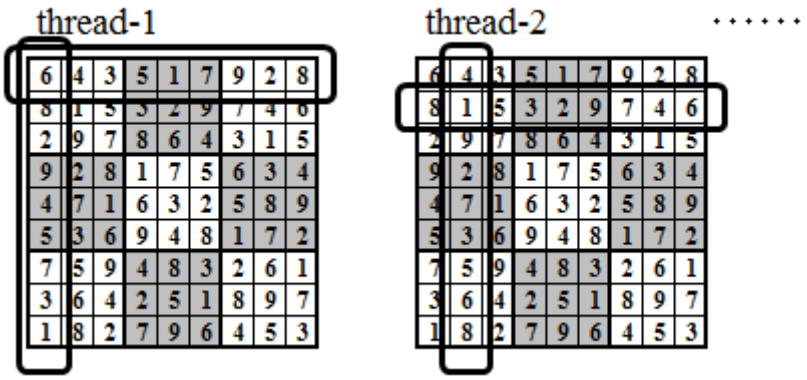

Fig. 12 評価関数の並列化

4.1.2　突然変異の並列化

突然変異では、リージョン単位で初期配置以外の 2 マスをランダムで選択し、それらの数字をスワップするという処理を行っている。そのため、一個体に関して突然変異の処理を行う最大の回数はリージョン数である 9 回になる。したがって、突然変異では個体をリージョン毎に分割し、並列処理を行うこととする。

Fig.13 で示されるように、1 つのリージョン内の操作を 1 スレッドで行うように並列化するため、並列数はリージョン数と同じになり、9×9 の数独パズルでは 9 並列となる。

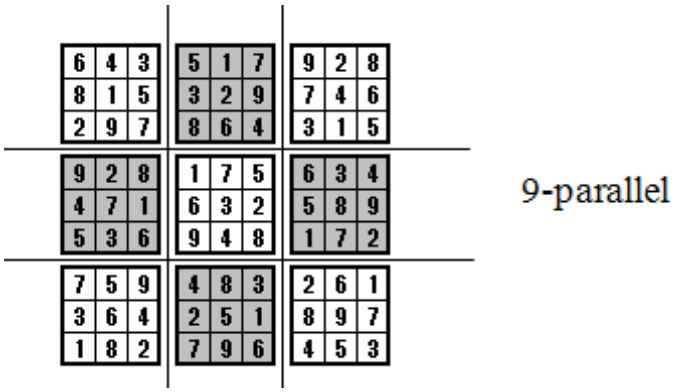

Fig. 13 突然変異の並列化

4.1.3　交叉の並列化

交叉では、3 章で説明したように、二つの親個体から二つの子個体を生成するために、行方向に 3 回、列方向に 3 回の計 6 回、比較・継承を行っている。並列化を考えるにあたってまずは、行列それぞれの比較・継承を別々に行えるように、6 分割にすることを考える。すると Fig.14 で示されるように、一つの個体から行方向と列方向にそれぞれ三分割された、

6 つの部分解が作られる。図中左に描かれているのが基となる個体であり、右側上部には行方向に分割した個体、右側下部には列方向に分割した個体の例を示している。

6 分割された部分解は、Fig.15 で示されるように、行方向と列方向のそれぞれ 1 ブロックを同一スレッドで行うことにより、3 並列で処理する。並列化された一つのブロックでは、行列それぞれの評価値計算・比較・子個体への継承を行っている。

また、行方向の一処理と列方向の一処理を同一スレッドで行う理由は、評価計算の並列化と同じ、各行列の処理を別スレッドへ分けることにより、並列化の効果よりもスレッドの生成コストの方が勝ってしまうことを避けるためである。

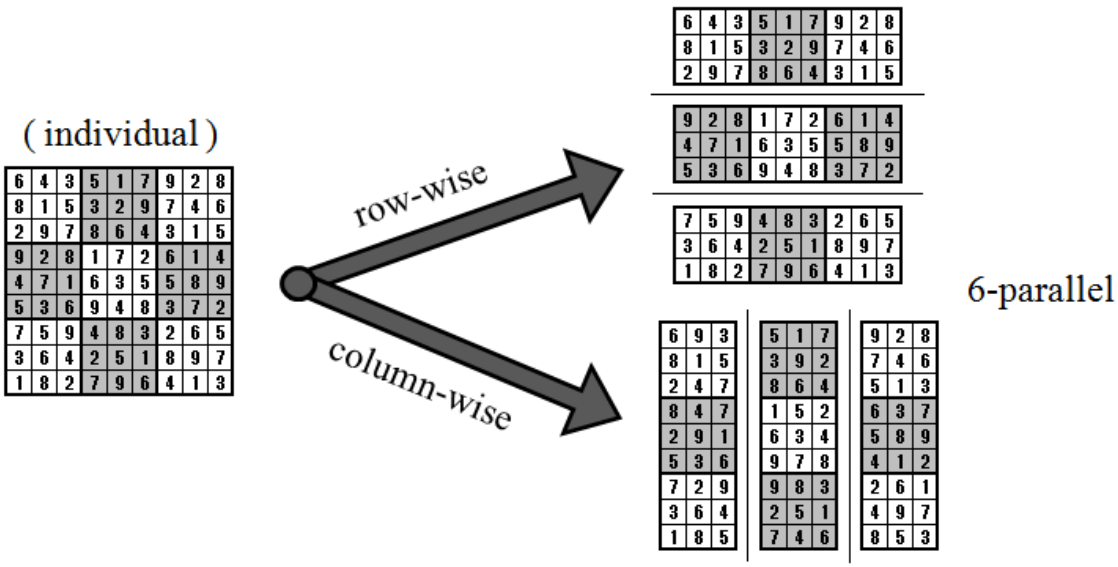

Fig. 14 交叉の分割

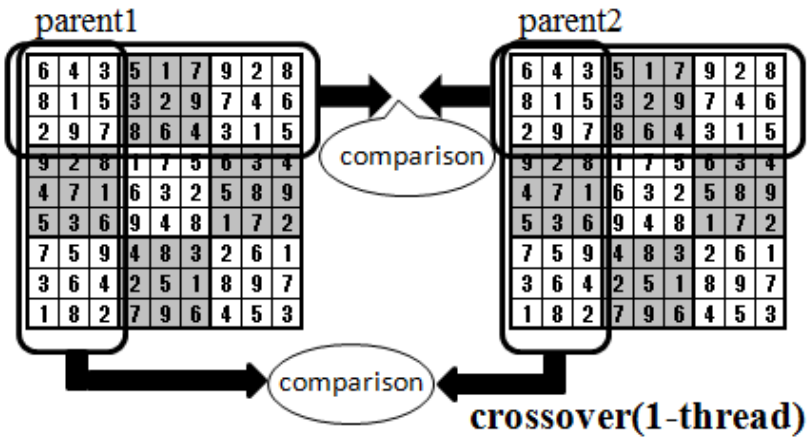

Fig. 15 交叉の並列化

5　実験

5.1 実験方法

今回の手法では、選択はトーナメント選択を用いる。トーナメント選択とは、個体群の中から決められた数の個体を取りだし、その中からもっとも評価値の高い個体を選択する方式である[19]。一度選択された個体も元の個体群に戻され、この過程を欲しい個体数の分だけ繰り返す。また、個体群の中から取り出す決められた数のことを、選択圧と言う。

実験に用いる数独の問題は、問題集[23]から初級・中級・上級問題をランダムに各 2 問ずつ、計 6 問選択した。問題の各クラスにおける初期配置数は、初級問題で 33~38、中級問題で 28~34、上級問題で 24~28 となっている。実験に使用した問題の番号と初期配置数はそれぞれ、初級問題が(No.1:38)、(No.11:34)、中級問題が(No.27:29)、(No.29:30)、上級問題が(No.77:28)、(No.106:24）である。

5.2 精度実験

精度実験では、打切り世代を 10 万世代に設定し、100 回試行した平均を表示する。

比較のため、提案手法(Proposed Method)の他に、ランダムサーチ(Random Search)と 3.1 節に示したリンケージを考慮した遺伝的操作を実現した方式である GOL (Genetic Operation that consider Linkage) 手法においても同様の実験を行う。

また、従来手法[2]との比較のため、初期配置数が同一の問題を用いて、実験を行う。

5.2.1　パラメータ

パラメータ条件は以下の通りに設定し、実験を行う。

[個体数]	540	[打切り世代数]	100000
[交叉率]	0.4	[突然変異率]	0.4
[選択圧]	3	[子供候補の個体数]	1080
[実行回数]	100		

ただし、突然変異率については、GOL では 0.3、ランダムサーチでは 0.1 としている。また、子供候補の個体数とは、3.2 節で説明したローカルサーチ機能向上のために整数倍生成するとした子供の候補となる個体数のことである。

5.2.2　実験結果

実験結果のグラフと表を Fig.16 と Fig.17、Table.1 に示す。Fig.16 では x 軸方向に初期配置数、y 軸方向に世代数の対数を取る。Fig.17 では x 軸方向に初期配置数、y 軸方向には世代数を取っているが、GOL と提案手法をより詳しく比較するため、25000 世代までにフォーカスしたグラフとなっている。

また、Table.1 は Fig.16,17 の基となったデータを表にしたものである。Table.1 ではそれぞ

れの問題・手法において、Generations では最適解に到達するまで世代数、Count では打ち切り世代数である 10 万世代以内に収束した回数を記している。

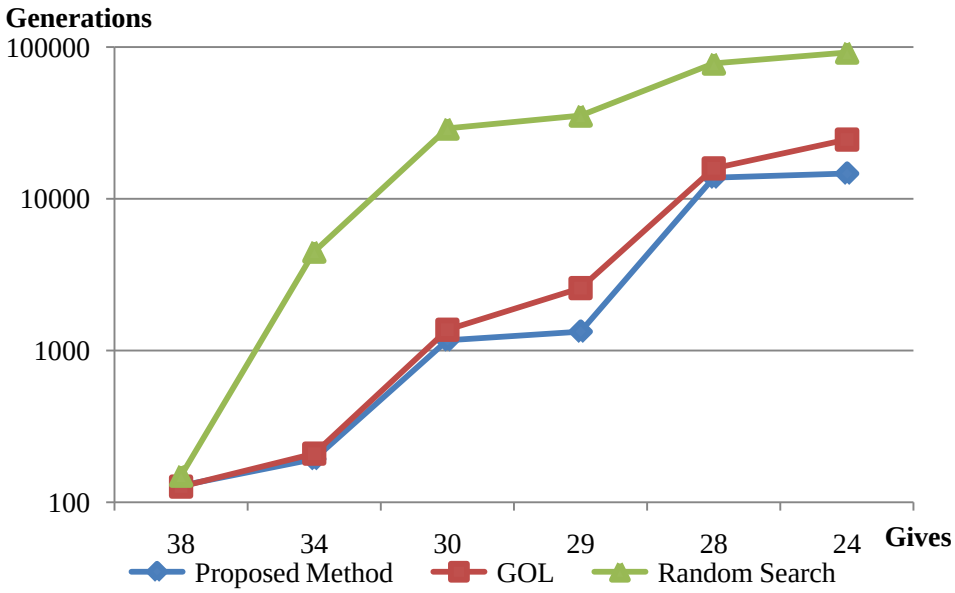

Fig. 16 初期配置数と世代数の関係（対数）

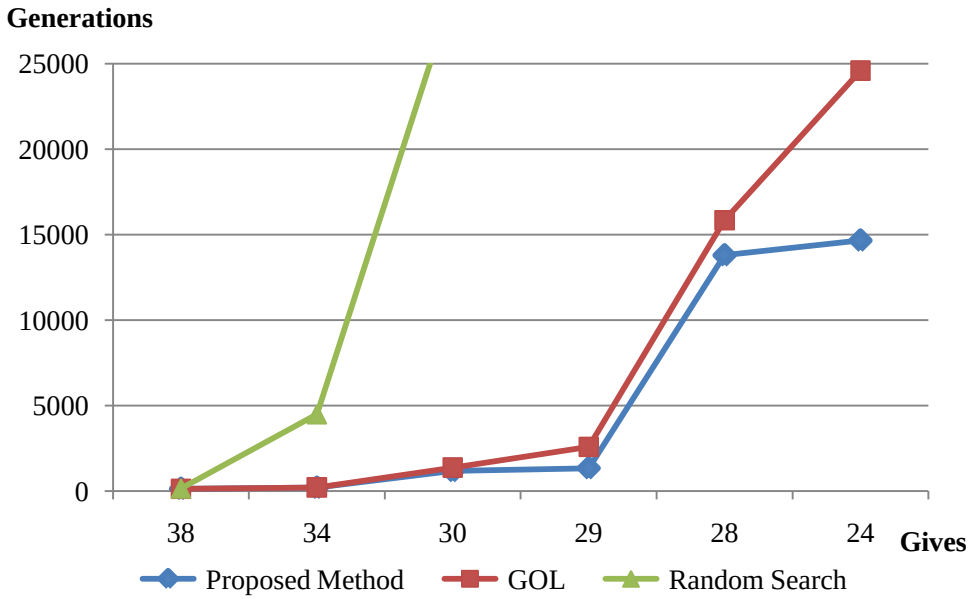

Fig. 17 初期配置数と世代数の関係（25000 世代まで）

Table. 1 精度実験の結果

	No.	1	11	29	27	77	106
	Gives.	38	34	30	29	28	24
Proposed Method	Generations	129	193	1169	1333	13786	14652
	Count	100	100	100	100	100	100
GOL	Generations	127	209	1370	2577	15834	24589
	Count	100	100	100	100	100	100
Random Search	Generations	150	4494	29084	35421	77978	92144
	Count	100	96	76	73	31	14

Fig.16,17、Table.1 より、ランダムサーチと、GOL を比べると、GOL の方が早く収束していることが分かる。よって、3.1 で説明したビルディングブロックを破壊しないための交叉が、効果をあげていると考えられる。次に、GOL と提案手法とを比べると、提案手法の方が少ない世代で収束している。よって、追加したローカルサーチの機能が、効果をあげていると考えられる。

したがって、今回提案した二手法ともに、数独解法の精度を向上させることに有効であると考えられる。

次に、従来手法(Existing Method)[2]と提案手法(Proposed Method)の成功回数を比較した表を Table.2 に示す。表では 7 つの異なる初期配置数(Givens)の問題において、打ち切り世代を 10 万とし 100 回試行した内最適解を導き出した回数を表示している。ただし、従来手法の問題が入手出来なかったため、従来手法と提案手法では初期配置数が同じだけの、異なる問題を用いている。

Table. 2 従来手法との比較

Givens	36	33	30	28	25	23	22
existing methods	100	100	69	46	30	4	6
proposed method	100	100	100	100	100	100	93

Table.2 から、従来手法では求解の難しかった初期配置数が 22~23 の問題においても、提案手法は 9 割以上の確率で求解に成功している。

この結果から、提案手法が従来手法と比較しても、大いに有効であると言える。

5.3 性能実験

性能実験では、打切り世代を問題毎に設定し、打ち切り世代数までの実行時間を 30 回計測した平均を表示する。また、比較のため、並列処理を行わない方式(Original)、交叉のみ並列処理した方式(P-Crossover)、評価計算のみ並列処理した方式(P-Evaluation)、突然変異のみ並列処理した方式(P-Mutation)、評価計算と突然変異を並列処理した方式(P-E&M)の各方式に対して、同じ環境とパラメータを用いて実験を行う。

5.3.1パラメータと実験環境

パラメータ条件は以下の通りに設定し、Table.3 で概要を記した PC にて実験を行う。

[個体数] 540 [交叉率] 0.4

[突然変異率] 0.4 [選択圧] 3

[子供候補の個体数] 1080 [実行回数] 30

打ち切り世代数は問題毎に異なる。また、打ち切り世代以内に最適解が求まっても、打ち切り世代まで遺伝的操作を繰り返すこととする。ただし、各問題における打ち切り世代数とは、Table.1 で示した、精度実験で得られた求解に掛かる平均世代数を用いている。

Table. 3 実験機の性能

OS	Microsoft Windows 7 Enterprise
Processor	Intel Core i5 M520 @ 2.40GHz
Core	2/4 (logical / physical)
Memory	4.00GB

5.3.2実験結果

実験結果の図表を Table.4, 5 と Fig.18, 19 に示す。

Table.4 では表頭に問題番号と初期配置数、当該問題における打ち切り世代数を、表側に各方式を表示し、打ち切り世代数まで実行するのに掛かった時間を記している。Table.4 における数値の単位は秒であり、小数点第四位で四捨五入を行っている。Fig.18 では、*x* 軸方向に初期配置数、*y* 軸方向に処理時間を取り、Table.4 の結果をグラフで表している。

また、Table.5 では Table.4 の結果から各方式の高速化率を算出している。本稿における高速化率とは並列処理を施していないオリジナルのプログラムと比較して、何%実行時間が短縮されたかを算出したものである。Table.5 における数値の単位は%であり、小数点第二位で四捨五入を行っている。Fig.19 では、*x* 軸方向に初期配置数、*y* 軸方向には高速化率を取り、Table.5 の結果をグラフで表している。

Table. 4 各問題の並列化手法毎の実行時間(秒)

No.	1	11	29	27	77	106
Givens	38	34	30	29	28	24
Generations	129	193	1169	1333	13786	14652
Original	0.5002	0.7215	4.1658	4.8387	44.6273	46.9960
P-Crossover	0.5056	0.7318	3.9666	4.8848	45.1123	47.5009
P-Evaluation(E)	0.4890	0.6738	3.5120	4.3715	39.9477	41.7359
P-Mutation(M)	0.4214	0.6205	3.4889	4.2214	40.2115	42.9701
P-E & M	0.3648	0.5329	3.0282	3.5770	34.4594	36.9264

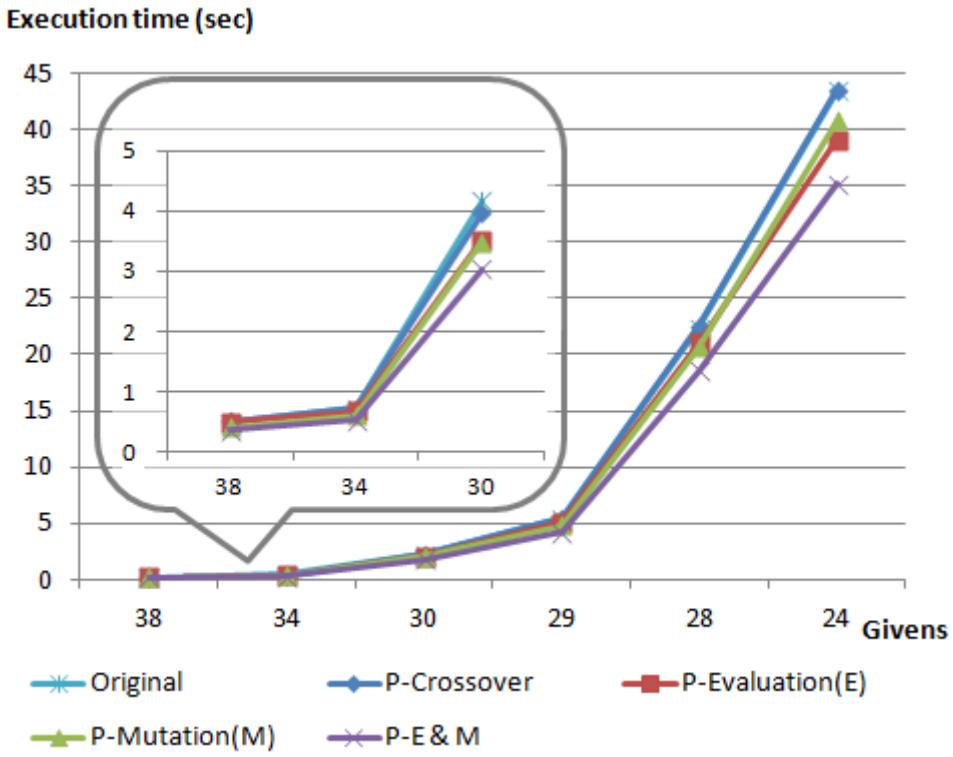

Fig. 18 並列化による実行時間の変化

Table.5 各問題の並列化手法毎の高速化率(%)

Givens	38	34	30	29	28	24
P-Crossover	-1.08	-1.43	4.78	-0.95	-1.09	-1.07
P-Evaluation(E)	2.24	6.61	15.69	9.66	10.49	11.19
P-Mutation(M)	15.75	14.00	16.25	12.76	9.89	8.57
P-E & M	27.07	26.14	27.31	26.08	22.78	21.43

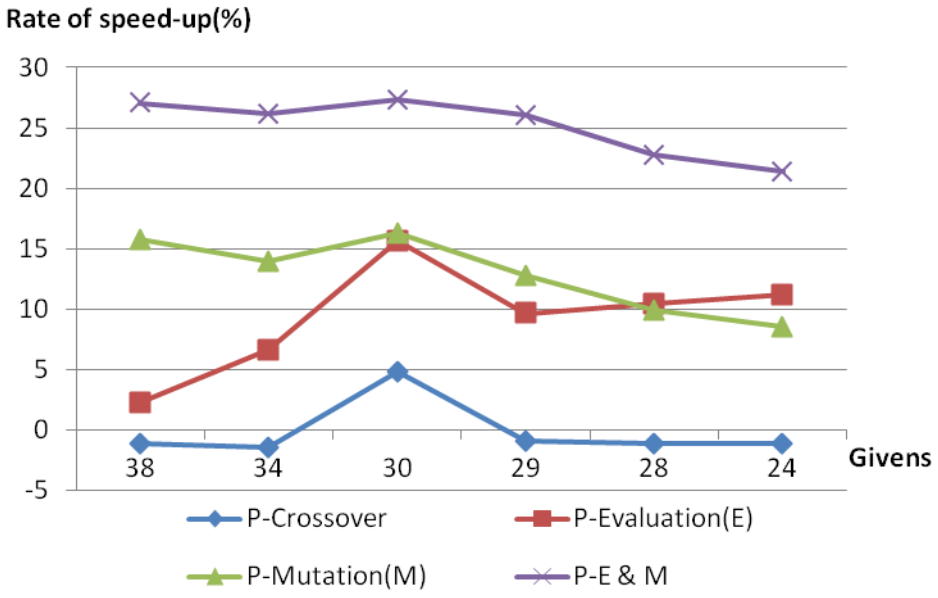

Fig. 19 初期配置毎の高速化率の変化

Table.4, 5、Fig.18, 19 より、オリジナルと比較して、評価計算や突然変異の並列化は 2~16%の高速化に成功し、双方を並列化した方式では 20~30%の高速化に成功している。評価計算では特に初期配置数が少ない、難解な問題ほど高速化の効果が現れる傾向にある。一方、突然変異では、初期配置数が多い簡単な問題の方が、高速化の効果が若干大きく現れる傾向にある。交叉の並列化の高速化率は-2~5%となっており、オリジナルと同程度か僅かに遅くなってしまった。

5.4 考察

精度実験において、Fig.17 のグラフや Table.1 を見ると、ランダムサーチと GOL に比べ、GOL と提案手法の差は少なく見える。これは今回の実験では初めから個体数を多く設定していたため、突然変異で子個体の個体数を増やしてもその効果が得られにくかったからと考える。そのため、パラメータ設定において個体数を少なくした場合、より顕著に効果が現れると考える。また、Fig.16 において、初期配置数が少なくなるとランダムサーチと他 2 手法の値が近づいているように見えるが、これは 10 万世代で操作を打ち切っているためであり、打切り世代を無くせば平均世代数の差はさらに開くと考えられる。

GA を用いた従来手法[2]では、打切り世代が 10 万世代まで実行した結果、有効世代以内に最適解を求めることができた確率が、初期配置数が 30 の問題では 69%、25~28 では 30~50%、最難解である 22、23 の問題に至っては、約 5%となっている。対して提案手法では、Table.2 で示したように、初期配置数が 23～36 の問題全てにおいて、100%の最適解を導き出しており、22 の問題においても、9 割以上が 10 万世代以内に最適解を導き出している。よって、従来例と比べても飛躍的に精度が向上していると考えられる。その理由として、従来例においては交叉での個体群多様性を求めたために、ビルディングブロックを破壊していることが大きいと考えられる。一般的に GA では多様性を持たせるように設計するが、今回の数独においてはそれが却って弊害になっていると考えられる。

初期配置数が同一で初期配置が別の異なる問題において、最適解が求まるまでにかかる世代数の平均値に大きな差が出る場合が観測された。その原因の一つとして、初期配置に依存していることが考えられる。さらに、評価関数の定義が簡素なために、初期配置に左右され易くなっているとも考えられる。よって、評価関数の設定方法を変更することにより、改善できる可能性があると考えられる。現在の評価関数は各行・列の要素数をそのまま評価値とし、すべての行列の和を個体の評価値としている。評価関数の変更方法としては、これを線形スケーリングするなどの手法が考えられる。

また、同一問題においても、最適解が求まるまでの世代数に大きな差が出る場合が観測される。この差は、初期配置数の少ない問題ほど顕著に現れている。これは問題の初期値依存性が高く、局所解に陥りやすくなっているためと考えられる。初期配置数の少ない問題では、解の探索範囲が広大になっている。今回の手法においては、初期配置数の少ない問題では、最適解からは遠いが評価値の高い局所解が多く存在し、局所解からなかなか抜け出せず、初期値依存性が高い問題になっていると考えられる。そのため、局所解からの脱出方法を検討する必要がある。例として、一定の世代毎に突然変異率を高くする、一定世代で解が収束しない場合は初期値を設定し直すなどの手法が考えられる。

次に性能実験について、Table.5 と Fig.19 が示すように、突然変異と評価計算の並列化を合わせると、20～30%の高速化の効果が得られた。提案手法は個体内での遺伝的操作の並列化なので、個体による並列化と併用すると、それぞれの効果を乗算した効果が得られると考える。

突然変異の並列化において、初期配置数が少ない難解な問題の方が並列化の効果が小さくなっている。これは突然変異では、スワップするマスを決定する際に多くの時間を取られるためと考えられる。初期配置数が少ない問題では、スワップで選択できるマスが多くなり、スワップするマスを決定するループが少なくなるため、結果的に突然変異に掛かる時間が減少するためである。

評価計算の並列化においては、突然変異とは反対に初期配置数の少ない難解な問題の方が、並列化の効果が若干高く現れている。これは個体の得点が高いほど評価計算に時間が掛かるため、個体群が高得点のまま長く操作を続ける難解な問題の方が、得点が上がるとすぐに最適解へ到達し操作が終了する簡単な問題よりも、並列化の効果がより大きく現れたと考える。

交叉の並列化において高速化の効果が現れなかった理由は、突然変異や評価計算の並列化と比べ、処理時間が短かったためであると考える。交叉だけでなく、突然変異や評価計算の並列化においても、高速化による効果が大きくなかったのは、プログラム全体におけるそれぞれの操作の割合が少なく、また、並列化して高速になった分とほぼ同じだけの時間をスレッドの生成やスレッド間の通信などに費やされてしまったためであると考える。

今後はスレッドの生成コストを減らす並列化の方法や、GPU など異なる並列化の手段を用いた実験との比較検討が必要と考える。

6 まとめ

本稿では GA の確率的探索手法としての有用性を実証するために、数独を GA に適用したプログラムの精度と性能の向上を試みた。

精度向上では、交叉がビルディングブロックを破壊することを防ぐためのオペレーションの提案と、ローカルサーチの性能向上の二つの提案により、従来手法では求解が難しく、実行したほとんどで有効世代内に最適解が求まらなかった問題と同等の初期配置数の問題においても、実行回数の半数で解を導き出すことが出来た。一方、数独の初期配置が少ない、解の探索範囲が広大な問題においては、最適解からは遠いが評価点の高い局所解が多数存在しており、初期値依存性が高く局所解に陥りやすくなっているという問題も見られた。

また、性能向上では個体単位ではなく遺伝的操作を並列化することによる高速化を提案し、その有用性を評価した。その結果、一定の高速化に成功し、性能を向上させることが出来た。本稿で提案した遺伝的操作の並列化と、個体の並列化は競合しないため、双方の並列化を併せることによって、双方の効果を乗算した高速化の効果が得られると考える。

今後の課題としては、精度向上のため、一定の世代毎に突然変異率を高くするなどの、局所解からの脱出方法を検討する必要があると考える。また、性能向上のため、並列化におけるスレッドの生成コストや通信コストを減らす方法や、GPU など異なる並列化の手段を用いた実験との比較検討も必要と考える。

参照文献

[1] Wikipedia. "数独" http://ja.wikipedia.org/wiki/%E6%95%B0%E7%8B%AC

[2] Timo Mantere and Janne Koljonen "Solving and Rating Sudoku Puzzles with Genetic Algorithms" New Developments in Artificial Intelligence and the Semantic Web Proceedings of the 12th Finnish Artificial Intelligence Conference STeP 2006

[3] Miguel Nicolau and Conor Ryan "Genetic Operators and Sequencing in the GAuGE System" 2006 IEEE Congress on Evolutionary Computation

[4] Timo Mantere and Janne Koljonen "Solving and Analyzing Sudokus with Cultural Algorithms" 2008 IEEE Congress on Evolutionary Computation

[5] 井上はづき, 佐藤裕二, "",第 2 回進化計算フロンティア研究会, (2009.10.02)

[6] Yuji Sato and Hazuki Inoue "Solving Sudoku with genetic operations that preserve building blocks", 2010 IEEE Symposium on Computational Intelligence and Games (CIG)

[7] V.S.Gordon and D.Whitley, "Serial and parallel genetic algorithms as function optimizers, " in Proc. Of the 5th International Conference on Genetic Algorithms. Morgan Kaufman, 1993, pp.177-183

[8] H.Muhlenbein, "Parallel Genetic Algorithms, Populations Genetics and Combinational Optimization" Proceedings of the 3rd Int. Conf. on Genetic Algorithms 1989, pp.416-421

[9] H.Muhlenbein, "Evoltion in time and space – the parallel genetic algorithm," in Foundations of Genetic Algorithms. Morgan Kaufmann, 1991pp.316-337

[10] Shonkwiler R. "Parallel Genetic Algorithms" Proceedings of the 5th Int. Conf. on Genetic Algorithms 1993, pp.199-205

[11] E.Cantu-Paz, "Efficient and Accurate Parallel Genetic Algorithms," Kluwer Academic Publishers, 2000

[12] 筒井茂義, "マルチコア計算機における進化計算の並列化,"2009 年度人工知能学会全国大会論文集, 2009

[13] 筒井茂義, "進化計算の並列化へのアプローチ:マルチスレッドプログラミングから超多スレッドプログラミングへ," 第 1 回進化計算フロンティア研究会, (2009.05.29)

[14] 佐藤裕二, 佐藤未来子, "進化計算からみたマルチコアプロセッサへの提案", 進化計算シンポジウム 2009, 4-03 (2009.12)

[15] 佐藤裕二, 佐藤未来子, 並木美太郎, "マルチコアプロセッサへの進化計算からの提案と評価", 第 5 回進化計算フロンティア研究会, pp.72-79 (2010.10.08)

[16] 棟朝雅晴, "遺伝的アルゴリズム-その理論と先端的手法-", 森北出版, 2008 pp.125-136

[17] H.Inoue, N.Hasegawa, M.Sato and Y.Sato, "Parallelization of Genetic Operations that Takes Building-Block Linkage into Account" AROB 17th 2012

[18] Gary McGuire, Bastian Tugemann, Gilles Civario "There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem" nature 2012

[19] 井庭崇, 福原義久, "進化と遺伝的アルゴリズム"複雑系入門, pp.102-117、(社)NTT 出版, 1998

[20] 佐藤裕二, 北咲也, 高嶺和也, 長谷川直広, "遺伝的操作を用いた数独解法と GPU による高速化について", 第 3 回進化計算シンポジウム 2010
論文集, pp.47-52

[21] 菅原清文, "C/C++プログラマーのための OpenMP 並列プログラミング", (株)カットシステム, 2009

[22] 北山 洋幸"OpenMP 入門―マルチコア CPU 時代の並列プログラミング", (株) 秀和システム, 2009

[23] ナンプレベスト 110 選 15, ナンプレプラザ編集部(編), (社)コスミック出版