修士論文

遺伝的アルゴリズムのための
並列処理フレームワークの提案と実装

Proposal and Implementation of Genetic Algorithms Framework

for Running on Parallel Environments

同志社大学大学院 工学研究科 情報工学専攻

博士前期課程 2011 年度 779 番

山中　亮典

指導教授　三木 光範教授

2013 年 1 月 25 日

Abstract

In this research, I propose a framework called GAROP (A Genetic Algorithms framework for Running On Parallel environments). In the GAROP, the logical model of genetic algorithms (GA) is executed on various parallel environments. In addition, I evaluate libraries to achieve the GAROP in terms of parallelization performance and productivity. High programming skill is required to use parallel environments. To solve this problem, administrators of parallel environments implement a system which calculates an evaluation value corresponding to gene about target optimization problems. Developers of GA can execute any GA without involving parallel processing to construct GA with the method of GAROP. I introduce the concept of the Individual Pool as a method for achieving GAROP. The Individual Pool consists of three queues. Libraries for it are implemented for windows clusters with C#, multi-core CPUs with C and GPUs with CUDA. I discuss on descriptions of source code and the execution time about GAs consisted of the libraries. As a result, developers of GAs can reduce the execution time with adding 4 descriptions on these parallel environments.

目 次

1 序論

遺伝的アルゴリズムなどの最適化手法に関する研究開発が進み，テスト問題を対象とした検証ではなく実際に実問題に適用する試みが行われている[1,2]．そのため対象問題が複雑化し，解の評価にかかる演算量が問題となっている．解候補集団による Generate-and-Test によって探索を進める GA では，解操作の演算を膨大な回数繰り返す．実問題では解の評価計算に多くの演算が必要であり，演算量および実行時間が問題となる場合が多い．一方で，計算機アーキテクチャの発展に伴い，マルチコア CPU や GPGPU（general purpose GPU）などの計算資源が容易に入手可能になってきている．多数の解候補を保持しつつ探索を進めるためにアルゴリズム自体が並列性を内包しており，並列処理との親和性が極めて高い．対象問題が複雑化し，計算資源が安価になっている現状の中で，様々な並列 GA が提案されている[3–6]．

ここで問題なのは，並列処理を実現する実装が特殊なプログラミング技術や知識を必要とすること，および計算機アーキテクチャの発展による実装方法の変化である．並列実装ではマルチスレッドプログラミング，排他的制御，通信と計算のオーバーラップ，メモリ階層向け最適化，および計算資源による負荷分散など，通常とは異なる技術および知識が要求される．加えて，近年の著しいアーキテクチャの革新に対応する場合，新しいアーキテクチャに対するこれらの実装を GA の開発者が行うことになる．これは非常に大きな労力である．

我々は，進化計算手法の 1 つである GA を並列実行するためのフレームワーク GAROP（Genetic Algorithms framework for Running On Parallel environments）を提案する[7–9]．GAROP の目的はアルゴリズム，並列処理システム，対象問題，および並列化スキルを問わず，評価計算を並列化するマスタ・スレーブモデルで GA を実行することである．GAROP には GA を実装する人物と並列処理を実装する人物を完全に分離する考えが根幹にある．各並列処理システムに対する実装を専門家に委ね，ユーザ[1]は主眼であるアルゴリズムの考案に専念する．そうすることで，ユーザは並列実装に関する負担を最小に保ちながら，並列処理による実行時間短縮の恩恵を受けられる．

本論文では，提供するライブラリを使用して構築した GA に対し，プログラミングの記述量および実行時間に関する検討を行う．また，GAROP を用いた GA の一例として，エリート個体の近傍個体を GAROP で評価するアルゴリズムを提案し，設計変数空間における解の探索領域を広げ，解の精度および信頼性を向上させる．結果として，Windows クラスタ，マルチコア CPU，および GPU において共通の関数を追加することで，約 2.94〜13.07 倍の速度向上を確認したことを示す．また，提案アルゴリズムにおいて単位時間当たりに約 96 倍の領域を探索したことを示す．

以下では，まず 2 章において GA の一般的な枠組みについて説明し，その並列化手法の現状について説明する．次に 3 章において，GA を並列化する際のモデルを定義した提案フレームワーク GAROP について述べる．4 章では，Windows クラスタ，マルチコア CPU，GPU について並列計算環境の特徴を示し，GAROP を実現するライブラリについて各環境での実装方法を説明する．5 章で作成したライブラリの，並列性能およびプログラム記述における生産性を評価する．6 章で評価の結果について議論する．最後に 7 章では GAROP の有効性について述べ，本論文のまとめとする．

[1] GA の開発者

2　遺伝的アルゴリズム

GA は生物が環境に適応して進化していく過程を工学的に模倣した確率的な最適化手法である [10]．自然界における生物の進化過程においては，ある世代を形成している個体集団の中で環境に適応した個体がより高い確率で生き残る．そして，生き残った個体が次世代に子を残す．この生物進化のメカニズムをモデル化し，環境に対して最もよく適応した個体，すなわち目的関数に対して最適値を与えるような解を計算機上で求めることが GA の概念である．

GA では 1 つの解候補を 1 個体として扱い，個体の集団を用いて解探索を行う．解候補は設計変数からなるベクトル表現や構造体など，問題によって異なる表現をとる．個体は設計変数値をコーディングした染色体により特徴づけられる．そして，染色体をベクトル表現や構造体にデコーディングし，目的関数値を計算する．なお，染色体は複数の遺伝子で構成されている．

各個体は目的関数値に応じた適合度を有し，ある世代を形成している個体集団において，適合度の高い個体ほど高確率で生き残るように選択を行う．加えて，交叉および突然変異といった個体生成の遺伝的操作によって子個体を生成し，次世代の個体集団を形成する．この世代更新の繰返しによって適合度の高い個体が集団内に増加し，最適解に集団を収束させるのが GA のメカニズムである．

GA は多点探索手法であり，多くの世代を繰り返す Generate-and-Test 型アルゴリズムでもある．また，複雑な最適化問題を対象とした場合，個体の評価計算の負荷が非常に大きくなる．そのため，演算量および計算時間が問題となっている。

2.1　遺伝的アルゴリズムの基本動作

GA の基本動作のフローチャートを Fig. 1 に示す．GA の各プロセスの内容は以下の通りである．

初期化（Initialization）初期母集団を形成する複数の個体をランダムに生成し，各個体の適合度を評価する．

終了判定（Terminate Check）あらかじめ定められた終了条件に基づいて，GA の処理を終了する．この時，母集団で適合度の最も高い個体を最適解として採用する．一般的には，世代数による終了条件が使用される．

評価（Evaluation）各個体に環境に合わせた適合度，すなわち目的関数値を計算する．

複製選択（Selection of Parerents）次世代の子を生成するための親個体候補を選択する．

交叉（Crossover）親個体 A の遺伝子と親個体 B の遺伝子を入れ替えることにより新しい子個体を生成する．

突然変異（Mutation）染色体上のある遺伝子を突然変異率に従って他の対立遺伝子に置き換える．

生存選択（Selection of Survivals）交叉，突然変異によって生成された子個体から，次世代に残る個体を選択する．

2.2 並列遺伝的アルゴリズム

GA は複数の解候補をサンプリングすることを膨大な回数繰り返しながら探索を進めていくため，本質的に並列化が可能である．そのため，種々の並列モデルが考案されている [3)]．ここでは，代表的な並列 GA として，マスタ・スレーブモデル，島モデル，およびセルラモデルについて説明する．また，解の探索性能を向上させる並列モデル，および実行時間を短縮する並列モデルの 2 つに関して述べる．

2.2.1 マスタ・スレーブモデル

GA では，評価計算による時間が全体の計算時間の大部分を占めることが多く，対象とする問題が複雑になるほどその傾向が強くなる．そこで，一般的な並列化の考えに基づくマスタ・スレーブモデル [11)] が考えられる．Fig. 2 に示すように，マスタ・スレーブモデルは評価計算以外の全ての操作はマスタとなる 1 つのプロセッサにおいて行われる．評価計算は，マスタプロセッサから複数のスレーブプロセッサに評価すべき個体の染色体を送信し，スレーブプロセッサにおいて実際の計算を行い，結果をマスタプロセッサに返す．そのため，2.1 節に示した GA の基本動作，すなわちアルゴリズムの論理モデルと並列計算環境で実装する物理モデルに違いはなく，マスタ・スレーブ型並列処理による解への影響は全くない．通信はマスタとスレーブ間のみに発生し，比較的通信量が多いモデルである．また，1 資源がマスタとして必要なため大規模な並列計算環境において用いられることが多い．

2.2.2 島モデル

分割母集団モデルとも呼ばれる島モデル [12,13)] では，GA の母集団を複数のサブ母集団に分割し，各サブ母集団ごとに探索を進めていく．そして，数世代ごとにサブ母集団間で個体の交換を行う．この操作を移住と呼ぶ．Fig. 3 のように，1 つの計算資源に 1 つのサブ母集団を割り当てる実装が一般的である．島モデルでは，移住を行う世代周期，サブ母集団内で移住する個体数などのパラメータが必要である．上記パラメータに加え，移住個体の選択法や移住先の選択法が探索能力に大きく影響し，GA の論理モデルとのつながりが非常に大きい．しかし，複数の母集団で探索を進めるため，多様性の維持が可能となり，上記パラメータをうまく設定すれば，単一母集団 GA に比べて高い探索性能を示すことが知られている．

2.2.3 セルラモデル

近傍モデルとも呼ばれるセルラモデルは，個々のプロセッサの能力が極めて低いか，あるいは専用のプロセッサ群からなる並列計算機上で実装されるモデルである．セルラモデルでは，1 つの並列計算機上で 1 つの GA を実行する．プロセッサ 1 つに割り当てられる個体は 1 つまたは極少数であり，プロセッサ間通信のオーバヘッドを減らすために，近くに配置されたプロセッサの個体とのみ交叉を行う．個体の配置状態が 2 次元格子状空間の区分単位であるセル上に配置されているように見えることから，セルラ GA とも呼ばれる．セルラモデルは極めて効率的ではあるが，部分集団の適当な近接構造の設定が困難なことや，近接構造の決定は並列計算機のアーキテクチャに依存することが問題である．この方法は最も原始的で，計算機との関連が非常に強く，用いるアーキテクチャによって探索性能が決定する．

2.2.4 論理モデルと実装モデル

GA の並列化には，探索性能を向上させるためのアルゴリズムの並列化と，計算時間を短縮するための並列化の 2 つがある．

論理モデル 並列に探索を進めることによって，解の多様性や近傍探索を行う．島モデルやセルラモデルが代表的な論理モデルとして挙げられる．

実装モデル 実際に並列計算資源を用いて複数のプロセスを同時に実行することで，実行時間を短縮する．マスタ・スレーブモデル，島モデル，セルラモデルが該当する．

Fig. 4(a) は論理モデルとして島モデルを，実装モデルとして逐次モデルを採用した GA である．Fig. 4(b) は論理モデルとして島モデルを，実装モデルとしても島モデルを採用した GA である．

このように，論理モデルは並列的に探索を行うモデルであるが，実際には逐次処理を行うことも可能である．また，実装モデルの制約で，論理モデルが限定される場合も存在する．例えば，実装モデルとして島モデルを採用した場合，論理モデルとして単一母集団 GA を採用することはできない．つまり，用いる並列モデルによってアルゴリズムが制限されてしまう．

2.3 遺伝的アルゴリズムの並列化と計算機アーキテクチャ

GA の並列化において，特定の計算環境に特化した実装モデルがしばしば見られる．これらは，計算環境を最大限に利用できるが，そのアルゴリズムを他の計算環境において使用することはできない．また，近年の GPGPU やメニーコアを有した計算資源を最大限に利用するためには，非常に高度なプログラミング技術が必要である．そのため，新たなアーキテクチャが登場するたび，GA 開発ユーザが新たな実装を行う必要があり，ユーザにとって大きな負担となる．

GA の並列化に関する現状として，通信効率の良い島モデルが採用されることが多い．Harding らは GPU 環境を用いて，遺伝的プログラミングにおけるテスト問題の評価計算を CPU 比最大 x1351 にまで速度向上できることを示している [14]．小野らは，グリッド環境を用いて，シングル CPU では 200 日かかる蛋白質の立体構造決定問題における GA の計算を数時間で正常に終了できることを示している [15]．Pospical らは，島モデルとマスタ・スレーブモデルを組み合わせたハイブリッドモデルを GPU 環境で実装し，大きな効果をあげている [16]．

GA はアルゴリズム自体に並列性を内包しているが，その並列度数は個体数に依存する．そのため，数千〜数万規模の並列計算能力を持つ GPU などの環境を有効に利用するには適していない．そこで，Harding らのアプローチのように個体の評価計算自体を並列化する手法が取られている．この方法は，アルゴリズムとは無関係な対象問題に依存するため，GA の本質的な並列化ではない．一方，小野らのアプローチは，GA の世代交代モデルと並列方式が密接に関係している．加えて，グリッド上に構成された計算資源を有効に活用できる実装である．そのため，アルゴリズムおよび計算環境を変更することが非常に難しい．前者の手法では対象問題を変更する度に並列実装をやり直さなければならない．後者の手法では，アルゴリズムを変更するためには並列方式まで見直す必要がある．また，どちらの手法においても，計算環境を変更する場合には使用する環境に合わせて再び並列実装を行わなければならない．

3　遺伝的アルゴリズムの並列処理フレームワーク：GAROP

GAROP は GA を並列計算環境下で実行する際のモデルを定義し，そのモデルを実現するためのフレームワークである．GAROP の目的は，ユーザが特別な並列化プログラミング技術を有することなく，複数の計算機上でマスタ・スレーブモデルの並列処理を実現することである．

3.1　要件

GAROP の目的を達成し，2 章で述べた問題を解決するためには，以下の要件を満たすべきであると考えられる．

任意の GA を実行可能な並列化方式

GA 開発ユーザが用いる計算資源の構成を意識せず，どのような GA でも並列化できる方式が必要である．つまり，任意の論理モデルを実行できる実装モデルを採用しなければならない．2.2.4 項で言及したように，論理モデルを制限しない実装モデルはマスタ・スレーブモデルのみである．よって，GAROP ではマスタ・スレーブモデルの採用が不可欠である．

逐次プログラムと同等の生産性

GA 開発ユーザの負担を軽減するため，どのような並列計算環境であっても共通のプログラム記述で使用できる必要がある．つまり，並列処理に関する実装を記述した API（Application Programming Interface）を提供しなければならない．また，その API は利用方法の異なるアーキテクチャに対して共通である必要がある．

3.2　設計

GAROP では，ユーザと並列計算システムを結ぶインターフェースとして個体プールの概念を導入する．本節では，個体プールの概念を説明した後，API について詳細を説明し，並列計算資源が実行する評価関数のテンプレートについて説明する．

3.2.1　個体プール

個体プールは個体の溜まり場であり，評価すべき個体および評価済み個体が格納される．そして，内在する評価すべき個体は自動的に並列評価される．ユーザは評価したい個体を個体プールに登録する．その後，必要な時に個体プールから個体を取得することで評価済みの個体を得ることができる．

個体プールの概要を Fig. 5 に示す．個体プールは 3 つのキューから構成されている．登録された個体を格納する Throw キュー，登録個体の順序を格納する Label キュー，および評価済み個体を格納する Get キューである．GAROP では Throw キューを常に監視し，格納された個体データを次々に計算資源へ送信する．計算資源は受信した個体データを評価し，評価値もしくは個体データを Get キューに格納する．この時，登録個体と取得個体の順序を保つために Label キューを利用する．また，GAROP には遺伝子をキー，評価値を値とした DB が付加価値として用意されている．個体をプールに登録する際にキーと照合し，一致した場合は評価計算を行わず評価値を取得する．DB によって評価回数を削減しつつ，Label キューによって登録個体と取得個体間の整合性が保証される．

個体プールを用いた GA の処理を Fig. 6，7 に示す．Fig. 6，7 はそれぞれ，SGA および島モデル GA を GAROP で並列実行する場合のイメージである．

3.2.2　プログラミングインターフェース

GAROP はライブラリ形式で構築され，Table 1 に示す 6 つの関数を提供する．initialize 関数は並列処理システムの初期化，および個体プールに必要なメモリ領域を確保する．throw 関数は個体データを個体プールに登録し，get 関数は個体プールから個体データを取得する．その際，個体を表現するデータ構造を制限しないために BYTE 単位のデータ列に変換する．また，個体サイズが可変な GA に対応するため，throw/get するデータのサイズを指定する．finalize 関数は確保したメモリの解放および並列計算システムとの接続を切断する．get_queue_num 関数は Get キューに格納されている個体の数を取得する．clear_pool 関数は個体プール内の 3 つのキューに格納されているデータを破棄する関数で，主に世代終了時に使用する．これら 6 つの関数はどのような並列処理システムにおいても不変である．

ライブラリ API を用いた際の擬似コードを List. 1 および List. 2 に示す．List. 1 は単一母集団の最も一般的な論理モデルである Simple GA（SGA）を GAROP に従って実装したコードであり，List. 2 は島モデルを論理モデルとした場合の GA コードである．プログラム記述に関して，List. 1，2 より並列処理に関する記述を完全に隠蔽可能である．

3.2.3　評価計算テンプレート

評価計算を実行するのは並列計算環境下にある計算資源である．そのため，評価関数は使用する計算資源上でコンパイル，実行可能な記述を行う必要がある．対象問題に依存する評価計算の実装を GAROP 提供者が担うのは現実的ではないため，評価計算の実装はユーザが行う．この時，GAROP の提供するテンプレートを用いることで，並列計算の恩恵を受けられる．テンプレートは用いるプログラミング言語によって異なる．List. 3 に C 言語における評価関数のテンプレートを示す．throw/get 関数と同様に，個体のデータ構造を制限しないために引数および返り値は unsigned chat 型ポインタとして記述する．parameter は，評価計算に必要な個体データ以外の値を指定するための変数である．

3.3　GAROP に基づく GA の構築

前節で示した設計に基づく，GAROP の概要を Fig. 8 に示す．また，GAROP では以下の流れで GA を実行する．

（1）並列計算環境の準備
　クラスタの構築や GPU の搭載など，用いる計算資源を用意する．

（2）テンプレートを用いた評価関数の実装
　与えられたテンプレートに従い，計算資源上で実行可能な評価計算を実装する．

（3）ライブラリの API を用いた GA の実装
　提供されるライブラリ API を用い，GA そのものを実装する．

(4) コンパイル

指定されたコンパイラでソースコードから実行ファイルを生成する．

(5) 実行ファイルを計算資源がアクセス可能なメモリ領域に配置

生成された実行ファイルを，計算資源が利用可能な場所に配置する．

(6) 実行

プログラムを実行する．

ユーザは任意の GA を構築し，評価部以外の部分を実装する．用いる並列処理システムに応じた評価部のテンプレートを用い，対象問題のコードと組み合わせることにより評価部を実装する．このテンプレートを利用することで，特殊な通信と評価タスクのスケジューリングをユーザから隠蔽できる．すなわち，ユーザは通信や計算資源に関する知識が無くとも，実行する計算環境に適したアルゴリズムを構築できる．

4 GAROP ライブラリの実装

我々は GAROP を実現するためのライブラリを作成している．対応する並列計算システムは，WCF（Windows Communication Foundation）による Windows クラスタ，pthread によるマルチコア CPU，CUDA（Compute Unified Device Architecture）による GPU である．

4.1 Windows クラスタ

Windows クラスタは，Windows Server を OS とする計算ノードによって構成されたクラスタである．普段使用している OS と同等の GUI（graphic user interface）で使用可能であるため，クラスタ利用の敷居は比較的低い．Windows クラスタを対象とするライブラリでは，並列化方式として WCF（windows communication foundation）を用いる．WCF はクライアント・サーバ方式のアプリケーションを作成するためのプログラミングモデルである．WCF では，関数をサービスという形式で定義しサーバに配布することで，クライアントから独立した関数を呼び出すことができる．WCF を用いた並列化では，一般的なクライアント・サーバ方式の概念とは異なり，1 台のクライアントと複数台のサーバが存在する．サーバを計算ノードに見立て，クライアントからサーバにサービス（処理）を要求する．つまり，クライアントマシンがマスタプロセッサとなり，サーバマシンがスレーブプロセッサとして扱われる．サービスとは具体的には，DLL（dynamic link library）形式で提供される．すなわち，評価関数を DLL 形式で定義し，各計算ノードに配置する．WCF の利点として以下が挙げられる．

- マスタとスレーブが互いに独立
- 評価関数のみをスレーブに設置可能

マスタとスレーブの間でソースコードや実行ファイルを共有しないため，環境構築の手間を省くことができる．また，関数のみをスレーブに設置し，実行時に呼び出す方式であるため，計算資源を有効に利用できる．

4.2 マルチコア CPU

マルチコア CPU は，1 つのメモリ領域を複数の演算コアが利用する共有メモリ環境である．一般に普及しているマルチコア CPU の特徴として，演算コア数が 2〜8 とそれほど多くない点が挙げられる．マルチコア CPU を対象とするライブラリではスレッド並列として pthread を用い，Fig. 9 に示す構成で実装する．1 つのスレッドを 1 つの計算資源として扱い，複数のスレーブスレッドが Throw キューを監視する．Throw キューに個体がある場合はそれぞれのスレーブスレッドが順次データを取り出し評価計算を行う．この構成は，少ない演算コアしか持たず，スレッド間でメモリ領域を共有できる特徴を活かすことができる．

4.3 GPU

GPGPU を対象とするライブラリでは CUDA を用いる．そのため，CUDA に対応している GPU が対象となる．CUDA は GPU 向け並列計算アーキテクチャである．CUDA C/C++という C/C++言語を拡張したプログラミング言語を使用するが，GAROP においてユーザの記述する部分は完全に C/C++と同一である．CUDA における GPU アーキテクチャを Fig. 10 に示す．GPU チップ内部には，ストリーミングマルチプロセッサ（streaming multi processor: SM）が複数ある．さらに SM 内部には，ストリーミングプロセッサ（streaming processor: SP）と呼ばれる最小単位の演算コアがある．また，容量およびアクセス速度の異なる複数種類のメモリを搭載している．

本節では，CUDA 対応 GPU におけるスレッドおよびメモリの階層構造を説明した後，GAROP ライブラリの実装について述べる．

4.3.1 スレッドの階層構造

CUDA では，数千〜数万のスレッドを起動し SP によって演算を行う．しかし，膨大な数のスレッドを 1 系列の整理番号で管理するのは困難である．そこでグリッド及びブロックという概念を導入し，その中で階層的にスレッドを管理する．概念的には，グリッドの中に複数のブロックがあり，ブロックの中に複数のスレッドがある．ハードウェア的には，スレッドは SP によって処理され，ブロックは SM によって処理される．グリッドに対応するのは GPU そのものであり，使用する GPU が 1 つの場合はグリッドを意識する必要はない．ブロック，スレッドの数は CPU から GPU へ処理を命令する際に指定する必要がある．その際，用いる GPU の SM 数や SP 数を考慮し，適切に値を設定しなければ速度向上を実現することは難しい．

4.3.2 メモリの階層構造

ビデオカードには大きく分けて 2 種類のメモリが搭載されている．GPU 内に搭載されているオンチップメモリ，およびビデオカード上に搭載されているオフチップメモリである．オンチップメモリは，容量は少ないが高速にアクセスできる．オフチップメモリは，容量は大きいがアクセスが低速である．CUDA では，レジスタメモリ，シェアードメモリ，グローバルメモリ，テクスチャメモリ，およびコンスタントメモリを使用可能である．GPU を用いてパフォーマンスを向上させるには各メモリの特徴を理解し，アクセス速度を考慮するプログラミングが重要である．特に，CPU 上のメインメモリとデータをやり取りするグローバルメモリ，および SM 内の SP で共通に使用できるシェアードメモリを有効に利用する必要がある．

4.3.3 GPU のための GAROP ライブラリ実装

前述のように，GPU を用いたプログラミングにおいてスレッド数，使用メモリのパラメータは非常に重要である．しかし，CUDA 対応 GPU はバージョン毎にアーキテクチャが異なり，最適なパラメータは変化する．また，今後も新しいアーキテクチャが登場すると予想される．GAROP では，用いる GPU の構成を静的に取得し，使用スレッド数を決定する．また，シェアードメモリの容量と個体プールに throw される個体サイズからシェアードメモリの使用可否を判別し，使用可能な場合はシェアードメモリに個体データを配置する．

GPU は多くの演算コアを持ち，CPU とは異なるメモリ領域を持つ分散メモリ並列計算環境である．

GPU 用の GAROP ライブラリは，Fig. 11 に示すように，CPU 上に GA を実行するメインスレッドがあり，さらに個体プールを構成するキュー，キューを監視するサブスレッドが CPU 上に存在する．サブスレッドは，Throw キューに存在する全ての個体を一度に GPU へ送信する．この方法は，前節で述べたマルチコア CPU と比べ，CPU と GPU 間の通信回数が少なく，分散メモリ環境で効果を発揮する．

5 GAROPの評価

本章では，提供するGAROPライブラリを使用してGAを実行し，その評価を行う．Table 2，3，4に示す構成のWindowsクラスタ，マルチコアCPU，GPUを用いて使用資源数に対する実行時間のスケーラビリティを検証する．そして，3つの環境におけるライブラリ使用時のプログラミング記述量および速度向上率について確認する．なお，GPUはTable 3に示すマシンに搭載されている．

5.1 並列計算環境の利用効率

動作確認として，使用資源数に対する実行時間のスケーラビリティを検証する．GAROPライブラリは，用いる環境に合わせて使用する資源の数を自動的に決定する．本実験では，実装の正確性を確認するため使用資源数を指定し，対象環境を利用できていることを確認する．

5.1.1 Windowsクラスタ

Table 5に示すパラメータで実行したSGAの実行時間，および1ノード使用時に対する速度向上率をFig. 12示す．横軸はスレーブプロセッサとして使用したノードの数であり，縦軸は総実行時間および速度向上率である．対象問題はハイブリッドロケットエンジン（Hybrid Rocket Engine: HRE）概念設計最適化問題[17]を使用した．今回用いた環境では，1個体の評価にかかる時間は約19.33秒であり，SGAの全処理のうち99 %以上の処理時間を占めていた．Fig. 12より，計算ノードの増加に伴う実行時間の短縮が確認できた．しかし，速度向上率に関して，16ノードを使用した場合に約13倍に留まる結果となった．これは，計算ノードの利用に関するスケジューリングを考慮せず，マスタノードの1スレッドで個体の送受信を行なっているため，クラスタ内に存在する計算資源の空時間が発生しているためだと考えられる．

5.1.2 マルチコアCPU

Table 6に示すパラメータで実行したSGAの実行時間，および1スレーブスレッド使用時に対する速度向上率をFig. 13に示す．横軸はスレーブとして使用したスレッドの数であり，縦軸は総実行時間および速度向上率である．対象問題は1-max問題であるが，大規模問題を模擬するために無駄な演算を行わせた．今回用いた環境では1個体の評価にかかる時間は約7 msecであり，全処理のうち99 %以上の処理時間を占めていた．Fig. 13より，スレーブスレッド数が7まではスレーブスレッド数の増加に伴う実行時間の短縮が確認できた．しかし，スレーブスレッド数が8の場合は実行時間が長くなる結果となった．これは，Table 3に示すように，使用したマシンの論理コア数が8つであることに起因すると考えられる．4.2章に示す方法では，1つのスレッドはマスタとしてGAを実行する．スレーブスレッド数を8つにした場合，総スレッド数は9つとなり論理コア数を超過する．そのため，速度向上が停滞したと考えられる．

5.1.3 GPU

Table 6に示すパラメータで実行したSGAの実行時間，および1CUDAスレッド使用時に対する速度向上率をFig. 14に示す．横軸はCUDAスレッド数であり，縦軸は総実行時間および速度向上率である．対象問題は5.1.2項に示した1-max問題を使用した．Fig. 14より，64個のCUDAスレッド

を使用した際に実行時間が最短になった．また，2〜8，10，11，13，16，22，32 の CUDA スレッド数で速度が向上しており，その他の CUDA スレッド数では速度が停滞していることが確認できた．

5.2　プログラム記述量と速度向上率

GAROP ライブラリを使用して構築した SGA の速度向上を 1 資源時と比較検証する．また，逐次実行プログラムに比べて追記，変更するプログラム記述を確認する．

List. 4，List. 5，および List. 6 にそれぞれ Windows クラスタ，マルチコア CPU，および GPU 環境を用いた場合にユーザとして記述したソースコードを示す．それぞれ，initialize，throw，get，および finalize 関数を追加するのみで実行できることを確認できた．Fig. 15 にシングルコア実行時を 1 とした場合の，GAROP ライブラリを用いた並列実行時の速度向上率を示す．結果として，Windows クラスタでは 16 ノード使用で 13.07 倍，マルチコア CPU では 7 スレッド使用で 5.25 倍，GPU では 64CUDA スレッドで 2.94 倍の速度向上が確認できた．List. 4，List. 5，List. 6 および Fig. 15 より，逐次プログラムと同等の記述で実行時間の短縮が確認できた．

6　考察

本章では，5 章で得られた結果に対する要因を検討する．Windows クラスタおよびマルチコア CPU を用いた結果は，使用資源数に対して実行時間が順当にスケーリングしていた．そのため，GPU に関して結果の検討を行う．また，GAROP を有効利用した GA に関して，基礎的なアルゴリズムを用いて実験的に考察する．

6.1　CUDA スレッド数に対する速度向上率

5.1.3 項にて得られた結果は，GPU へ処理を命令する回数に依存すると考えられる．1CUDA スレッドが 1 個体を評価するため，母集団サイズが 64 の場合，10 個体では 7 回の命令を行い，31 個体では 3 回の命令を必要とする．Fig. 16 に CUDA スレッドの数に伴う GPU への命令回数を示す．横軸は CUDA スレッド数，縦軸は命令回数である．Fig. 16 より，Fig. 14 の実行時間と対応していることが確認できる．また，GPU への 1 回の命令で処理する個体数（CUDA スレッド数）と実行時間の関係を Fig. 17 に示す．5 章で使用した GPU は 448 コアを保有しているため，一度に評価させる個体数が 1〜64 のどれであっても処理時間は変わらない．よって，保有コア数以下の個体数を処理させる場合には，GPU への命令回数によって処理時間が決定すると結論付けられる．

6.2　GAROP を用いた GA の一検討

5.2 節において GAROP の有効性を確認したが，6.1 節で考察したように GPU のような多資源環境下では個体数以上の並列性能を示すことはできない．本節では，GAROP を用いた個体数以上の並列性能を持つ GA について基礎的なアルゴリズムを検討する．単位時間当たりの解探索領域に関して，各種テスト関数を用いて GAROP の有効性を再確認する．

6.2.1　エリート個体の近傍を用いた GAROP 利用 GA

GAROP を有効利用する GA の流れを Algorithm 1 に示す．評価計算を行う際，すなわち Get キューから個体を取得する際に，Get キューに格納されている個体数が 0 ならばエリート個体の近傍個体をプールに登録する．その後，登録した近傍個体を取得し，エリート個体より評価値が高い場合に，エリート個体を更新する．本アルゴリズムでは，膨大な計算資源に比例した数の個体を評価できる．これによってより評価値の高い個体がエリートになるだけでなく，エリート個体周辺の設計変数空間を探索できるために，最良解の精度が大きく向上する．

6.2.2　単位時間当たりの解探索領域

Table 4 に示す GPU を用いて，前述した GA の単位時間当たりの解探索領域について確認する．対象問題として，最適化テスト問題である Rastrign，Rosenbrock，Griewank，Ridge，および Schwefel 関数を用いる．各関数は 2 次元とする．GA のパラメータを Table 7 に示す．

各問題に対して，設計変数空間における解探索領域を Fig. 18〜22 に示す．横軸は設計変数 1，縦軸は設計変数 2 であり，色の濃淡は評価値を表している．また，黒い点が既探索点である．どの問題でも逐次実行の SGA と比べて多くの領域を探索していることが確認できる．また，単位時間当たり

の評価個体数では約 96 倍の個体を評価している．

これらの結果より，GAROP は単純に実行時間を短縮するのみでなく，どのような並列計算システムでも実行可能なアルゴリズムの開発を支援することが可能であると結論付けられる．

7　結論

本論文では，遺伝的アルゴリズムのための並列処理フレームワーク GAROP を提案している．GAROP はユーザの並列実装に関する負担を軽減し，アーキテクチャを問わないアルゴリズムの開発を支援するフレームワークである．ユーザと並列処理システムとのインターフェースとして個体プールの概念を導入することで，演算と通信のオーバーラップやメモリ階層向け最適化をユーザから隠蔽できる．

我々は，GAROP の考え方を実現するためのライブラリを作成している．並列計算環境および使用言語に関係なく，共通の関数群を提供することで個体プールの概念を実現している．Windows クラスタ，マルチコア CPU，および GPU 環境を GAROP に基づいて利用するためのライブラリが用意されている．

GAROP ライブラリを用いた GA を，上記 3 つの環境で評価した．Windows クラスタ環境では，実問題のひとつである HRE 概念設計最適化問題，マルチコア CPU および GPU 環境では，テスト問題の 1-max 問題を用い，ライブラリを用いて構築した SGA と比較した．結果として，Windows クラスタでは 13.07 倍，マルチコア CPU では 5.25 倍，GPU では 2.94 倍の速度向上を確認した．その際のコーディングは評価計算部の記述をライブラリ関数に置き換えるのみであり，逐次プログラムとほぼ同等，すなわち並列処理に関する記述を全く行わなかった．特殊な技術，知識を必要としないコーディングで並列計算環境を用い，速度向上を実現する GAROP ライブラリは非常に有用であると考えられる．

また，GAROP 利用に関する検討として，エリート戦略を組み込んだ SGA を GAROP に従って実行し，検討した．指示した評価計算が終了しておらず，マスタプロセッサに空時間ができる際にエリート個体の近傍個体を評価する，という基礎的なアルゴリズムである．この GA を用いて最適化テスト問題を解いた場合，GPU 上では単位時間当たりに約 96 倍の個体を評価できることを確認した．これは，母集団サイズ以上の計算資源を有効利用しながら，GA の各世代における最良解の精度および信頼性を向上させたことを示している．

謝辞

本修士論文は，筆者が同志社大学 工学研究科 情報工学専攻博士前期課程在学中に知的システム研究室において行った研究をまとめたものである．本研究に関してご指導ご鞭撻を頂きました本学廣安知之教授，三木光範教授および電気通信大学吉見真聡助教授に心より感謝致します．また，本論文をご精読頂き有用なコメントを頂きました本学下原勝憲教授および小板隆浩先生に感謝致します．

本論文の執筆にあたっては，廣安知之教授，および生命医科学部 医療情報システム研究室の大堀裕一君，布川将来人君，真島希実さん，大久保祐希君，杉田出弥さんに推敲して頂き明瞭な文章にして頂きました．心より感謝しております．

最後になりますが，最後まで一緒に頑張ってきた研究室の同期の皆様，研究を手伝ってくれた後輩に心より感謝しております．ありがとうございました．

参考文献

1) 古田均, 亀田学広, 伊藤弘之. 遺伝的アルゴリズムを用いたコンクリート橋梁群の最適維持管理計画の策定. 応用力学論文集, Vol. 5, pp. 919–926, 2002.

2) 花田良子, 廣安知之, 三木光範. 遺伝的アルゴリズムによる工場の生産スケジュールの自動生成. 同志社大学理工学研究報告, Vol. 48, No. 4, pp. 21–28, 2008.

3) Erick Cantú-Paz. A Survey of Parallel Genetic Algorithms. *Calculateurs Parallels, Reseaux et Systems Repartis*, Vol. 10, No. 2, pp. 141–171, 1998.

4) 三木光範, 廣安知之, 畠中一幸, 吉田純一. 並列分散遺伝的アルゴリズムの有効性. 日本計算工学会論文集, Vol. 3, pp. 29–34, 2001.

5) Heinz Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In J. Becker, I. Eisele, and F. Mündemann, editors, *Parallelism, Learning, Evolution*, Vol. 565 of *Lecture Notes in Computer Science*, pp. 398–406. Springer Berlin / Heidelberg, 1991.

6) Mitsunori Miki, Tomoyuki Hiroyasu, Mika Kaneko, and Kazuyuki Hatanaka. A Parallel Genetic Algorithm with Distributed Environment Scheme. *IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 1, pp. 695–700, 1999.

7) Tomoyuki Hiroyasu, Ryosuke Yamanaka, Masato Yoshimi, and Mitsunori Miki. GAROP: Genetic Algorithm framework for Running On Parallel environments. 情報処理学会研究報告. MPS, 数理モデル化と問題解決研究報告, Vol. 2012, No. 5, pp. 1–6, 2012.

8) Tomoyuki Hiroyasu, Ryosuke Yamanaka, Masato Yoshimi, and Mitsunori Miki. A Framework for Genetic Algorithms in Parallel Environments. 情報処理学会研究報告. MPS, 数理モデル化と問題解決研究報告, Vol. 2011, No. 6, pp. 1–6, 2011.

9) 山中亮典, 吉見真聡, 廣安知之, 三木光範. 遺伝的アルゴリズムの並列計算システム向けフレームワークの提案. 情報処理学会研究報告. 計算機アーキテクチャ研究報告, Vol. 2010, No. 8, pp. 1–7, 2010.

10) David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

11) Erick Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. Technical report, University of Illinois at Urbana-Champaign, Urbana, IL. IlliGAL Technical Report No. 97004, 1997.

12) Reiko Tanese. Distributed Genetic Algorithms. *Proc. 3rd International Conference on Genetic Algorithms*, pp. 434–439, 1989.

13) Theodore C. Belding. The Distributed Genetic Algorithms Revisited. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pp. 114–121, 1995.

14) Simon Harding and Wolfgang Banzhaf. Fast Genetic Programming on GPUs. In *Genetic Programming*, Vol. 4445, pp. 90–101. Springer Berlin Heidelberg, 2007.

15) 小野功, 水口尚亮, 中島直敏, 小野典彦, 中田秀基, 松岡聡, 関口智嗣, 楯真一. Ninf-1/Ninf-G を用いた NMR 蛋白質立体構造決定のための遺伝アルゴリズムのグリッド化. 情報処理学会論文誌. コンピューティングシステム, Vol. 46, No. 12, pp. 369–406, 2005.

16) Petr Pospichal and Jiri Jaros. GPU-based Acceleration of the Genetic Algorithm. *GPU competition of GECCO competition*, 2009.

17) 小杉幸寛, 大山聖, 藤井考臧, 金崎雅博. ハイブリッドロケットエンジンの概念設計最適化. 宇宙輸送シンポジウム，講演論文集, 2010.

付 図

付 表

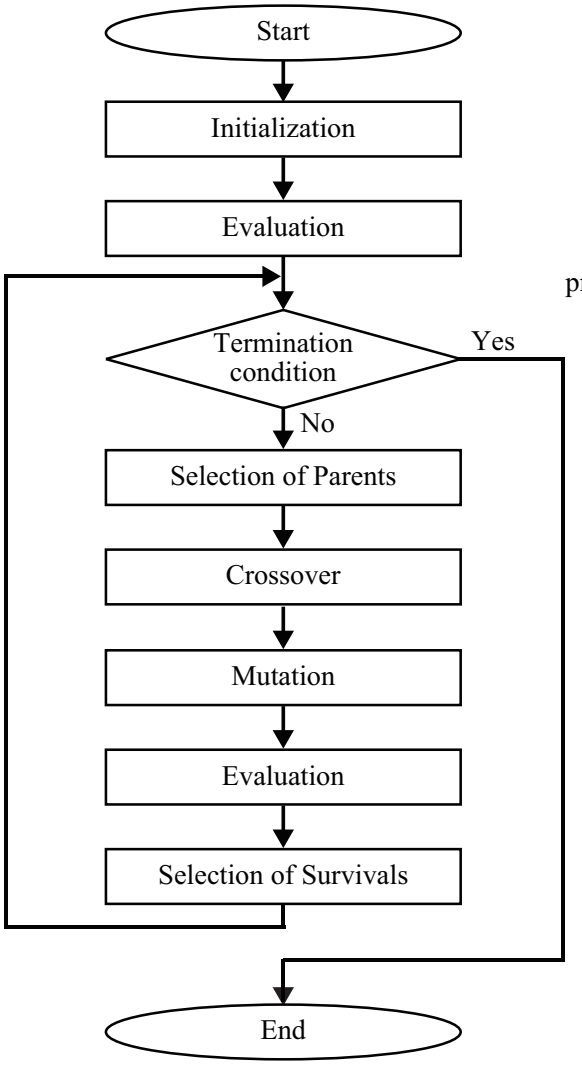


Fig. 1 Flowchart of GA.

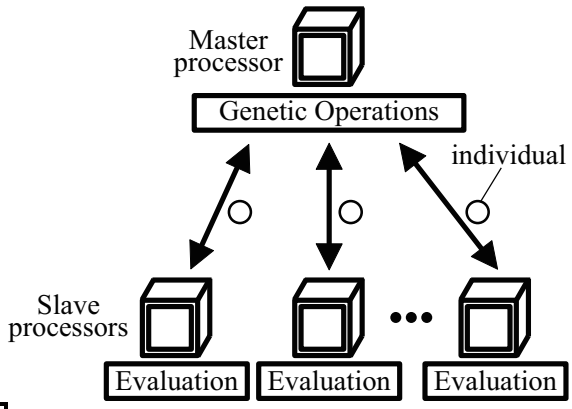


Fig. 2 Master-slave model.

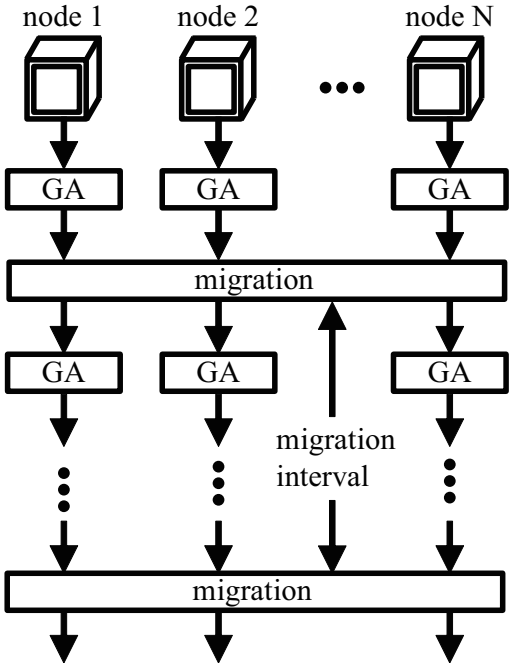
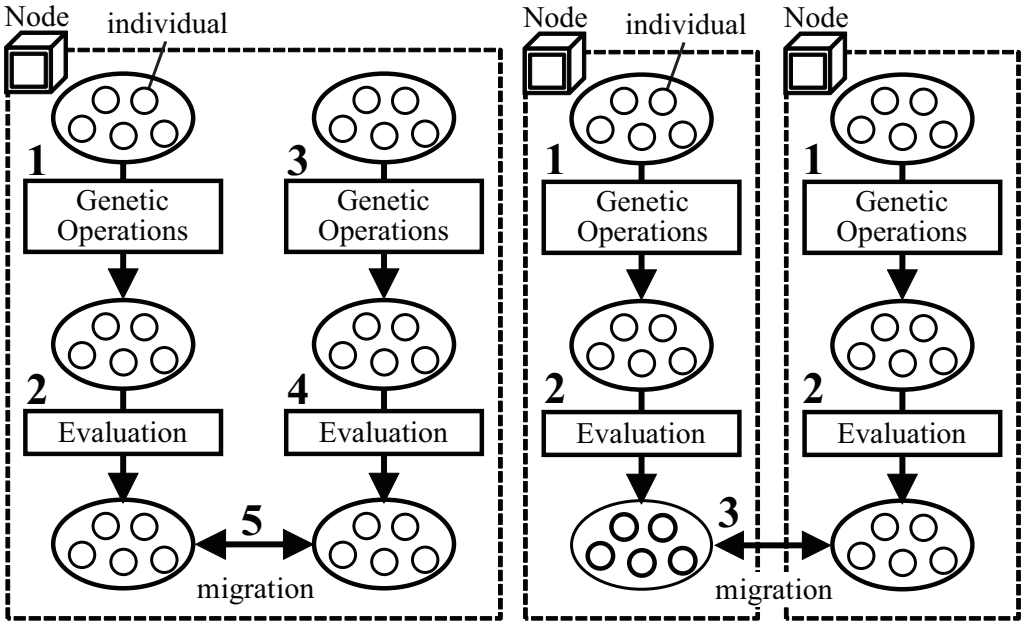


Fig. 3 Island model GA.



(a) Serial implementation. (b) Parallel implementation.

Fig. 4 Two island models as the logical model.

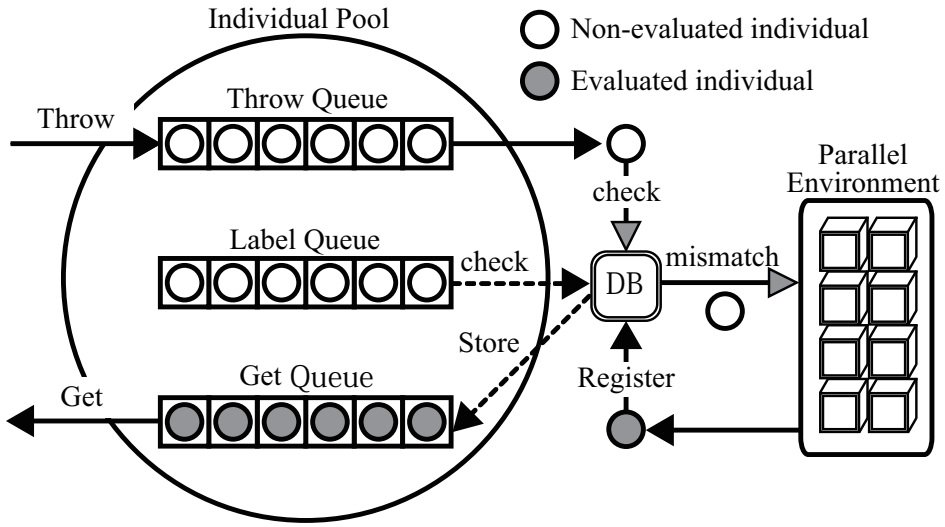


Fig. 5 Individual pool.

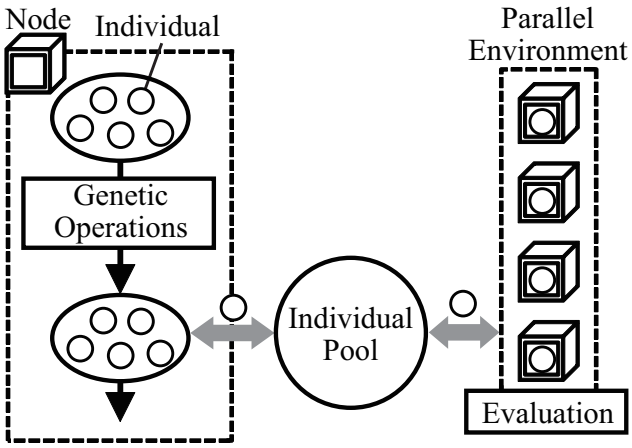


Fig. 6 SGA with the GAROP (concept).

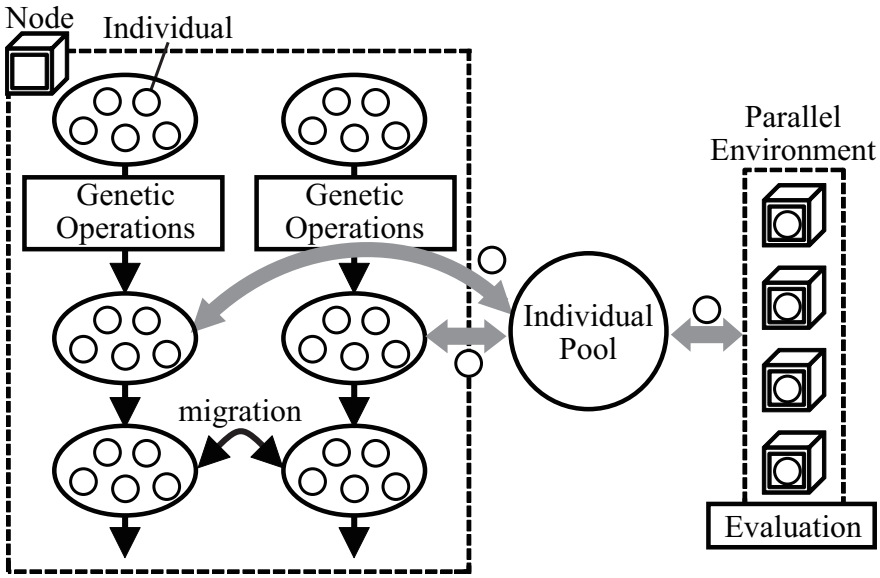


Fig. 7 Island model GA with the GAROP (concept).

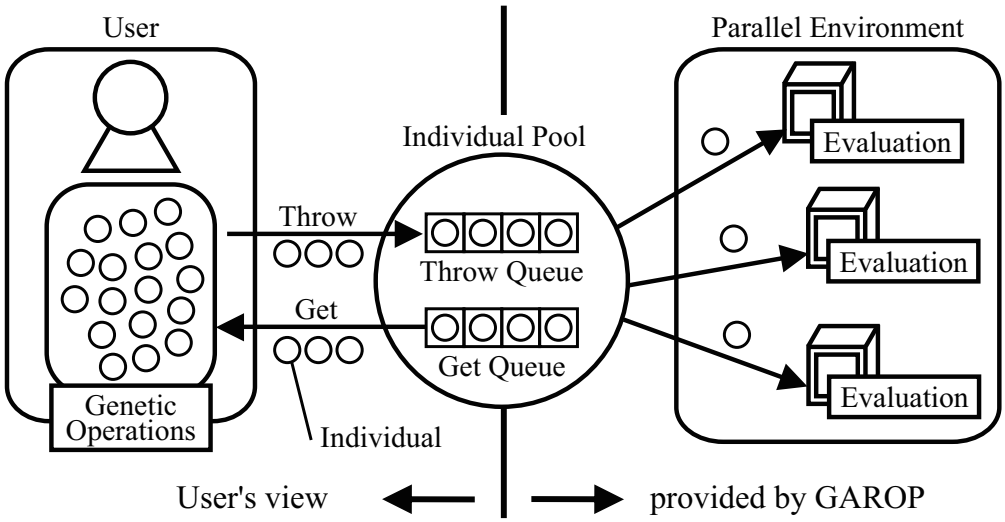


Fig. 8 Concept of GAROP.

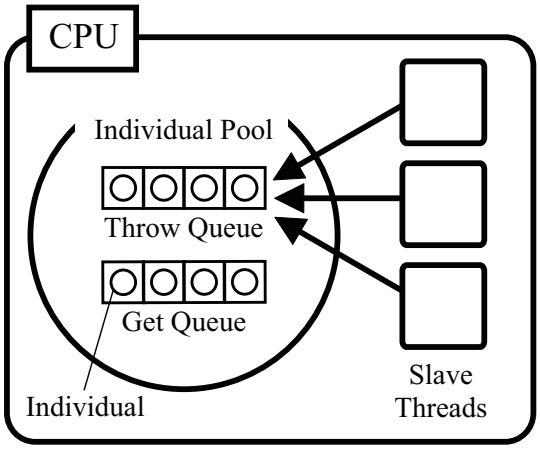


Fig. 9 Implementation of the GAROP library for multi-core CPU.

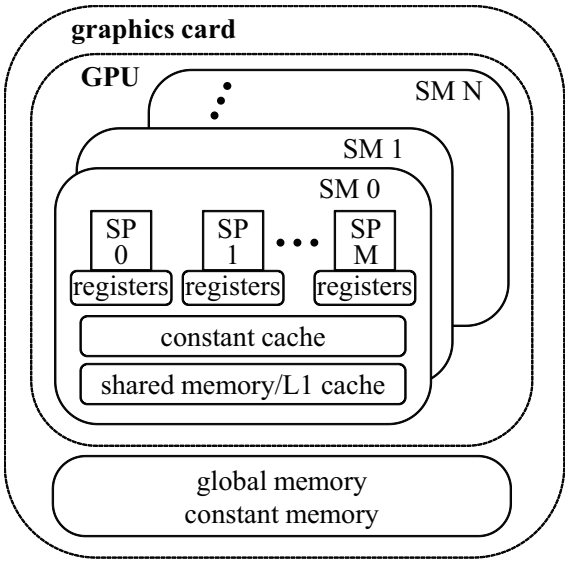


Fig. 10 Architecture of CUDA-capable GPU.

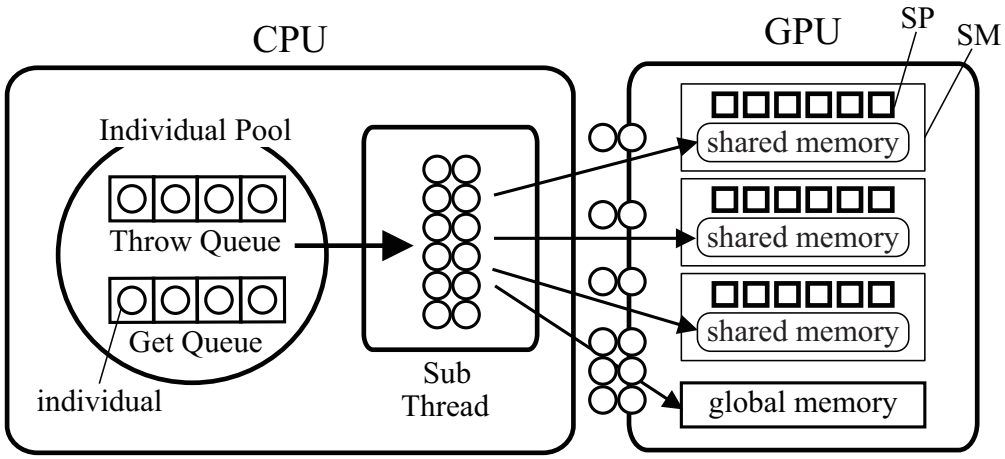


Fig. 11 Implementation of the GAROP for GPU.

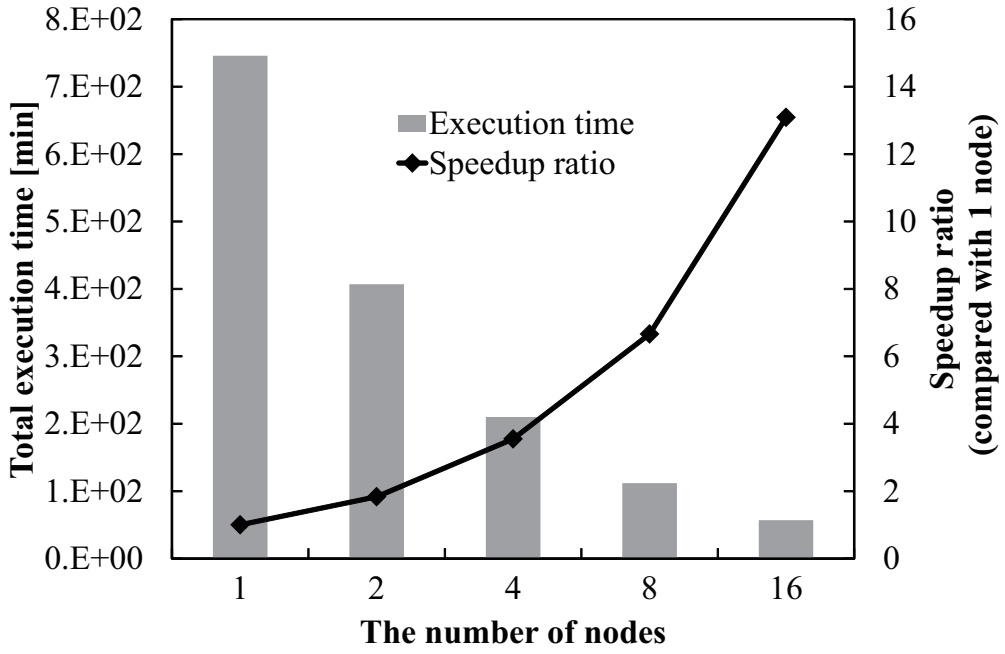


Fig. 12 Execution time and speedup on the windows cluster environment depending the number of nodes.

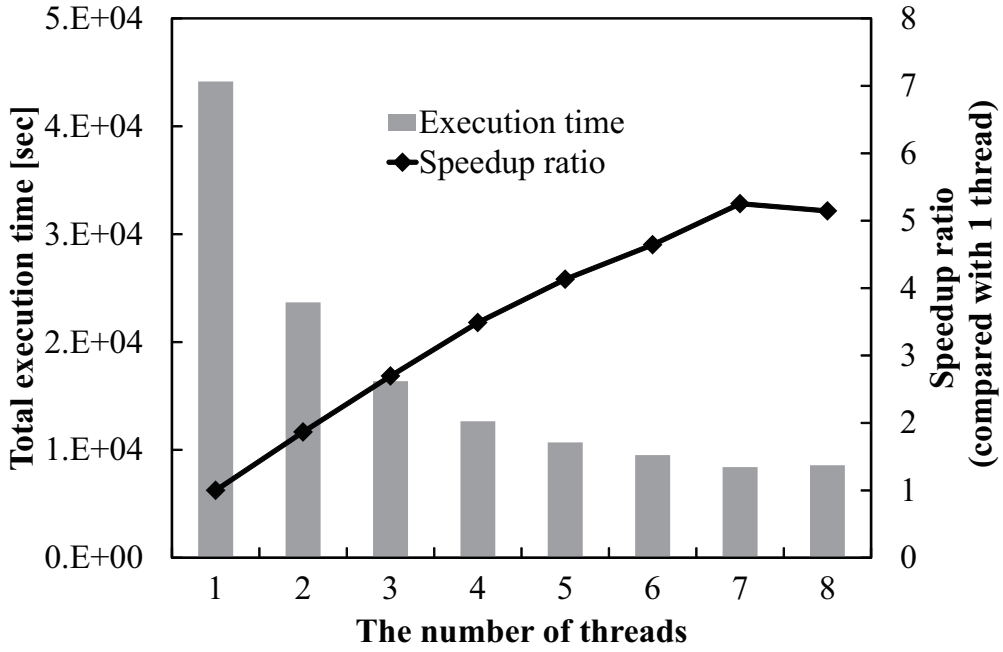


Fig. 13 Execution time and speedup on the multi-core CPU environment depending the number of threads.

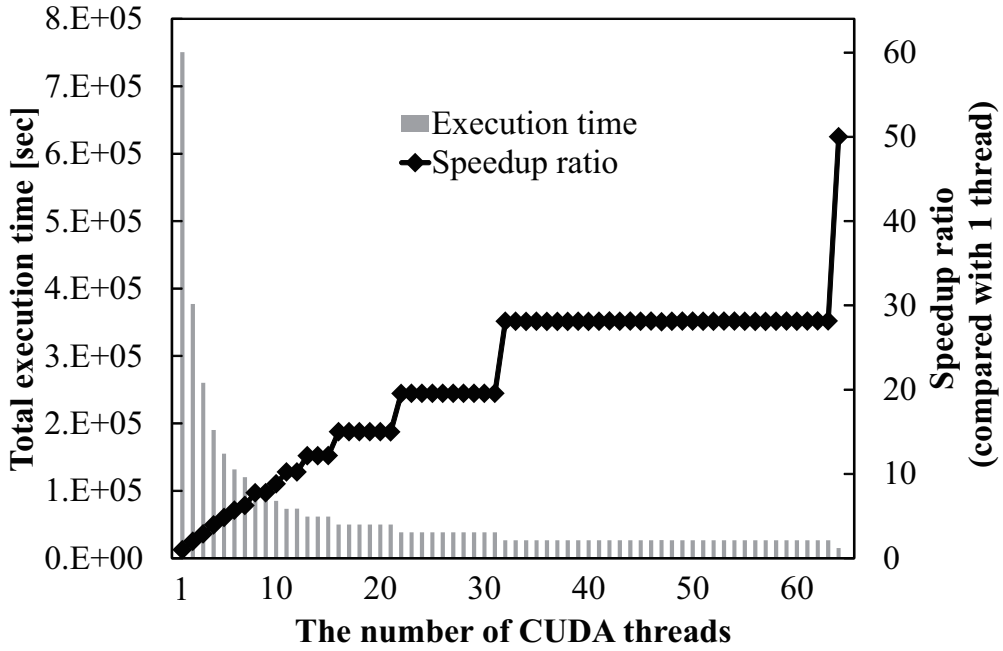


Fig. 14 Execution time and speedup on the GPU environment depending the number of CUDA threads.

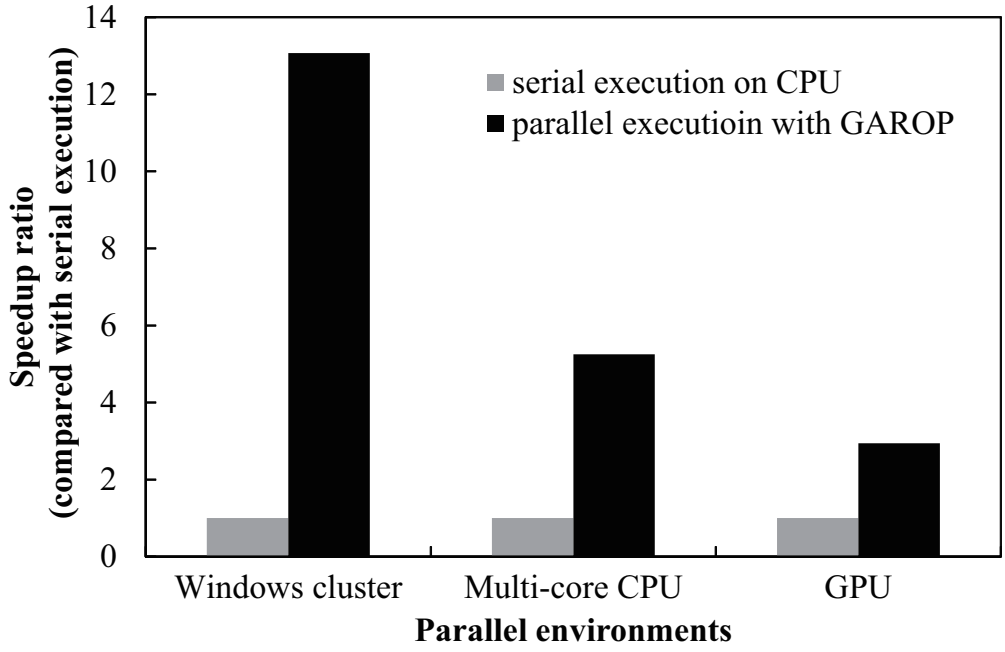


Fig. 15 Speedup on each environment compared with executing with single resource.

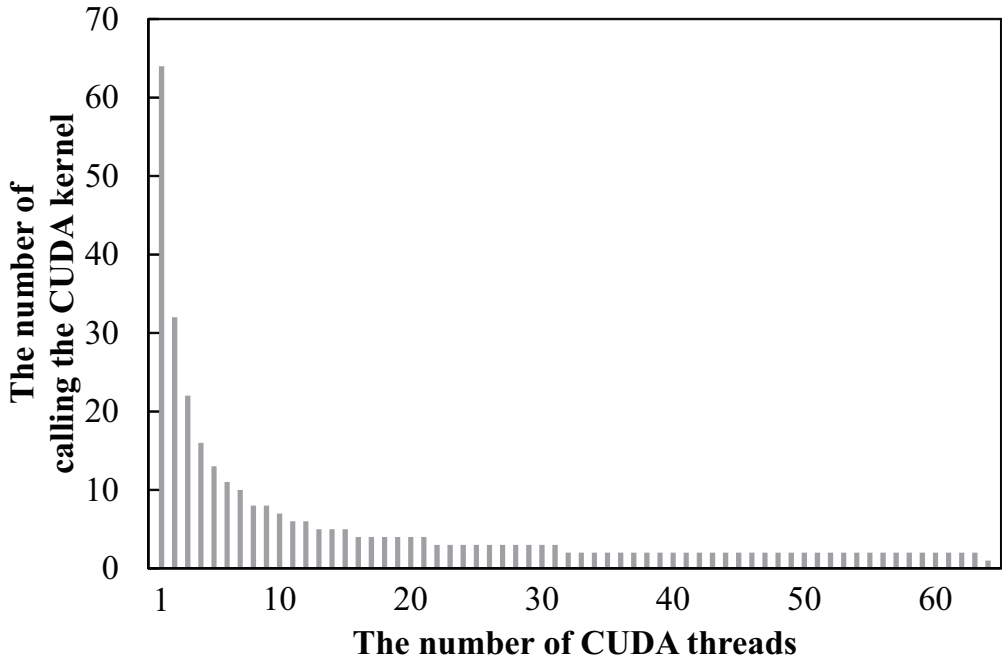


Fig. 16 The number of calling GPU depending the number of CUDA threads.

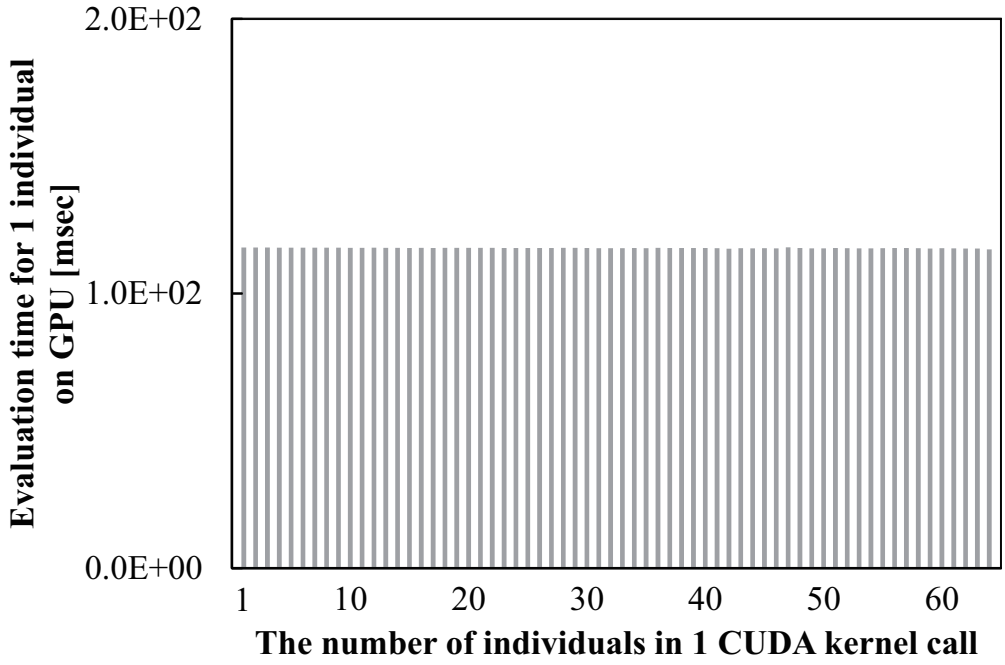
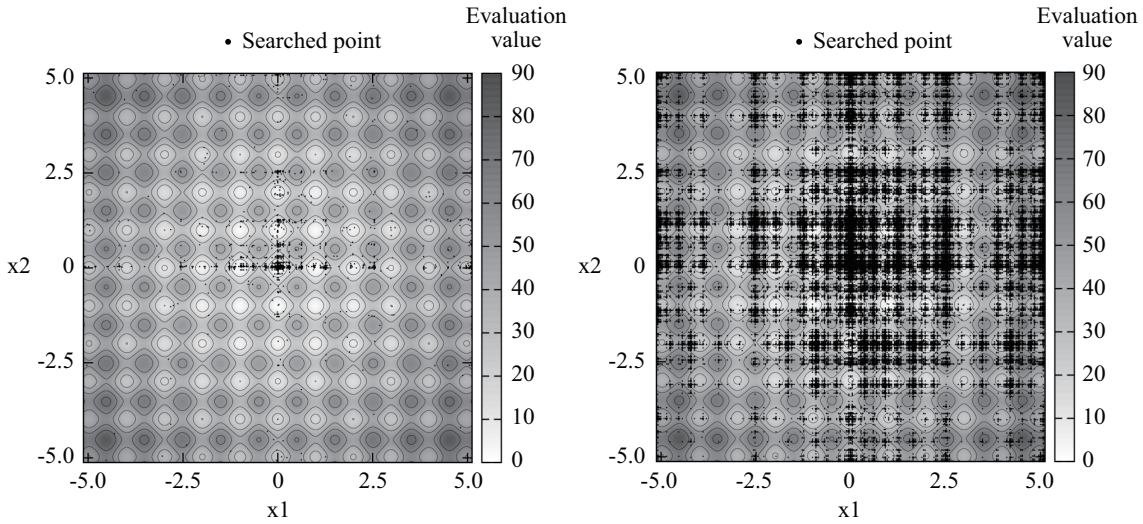


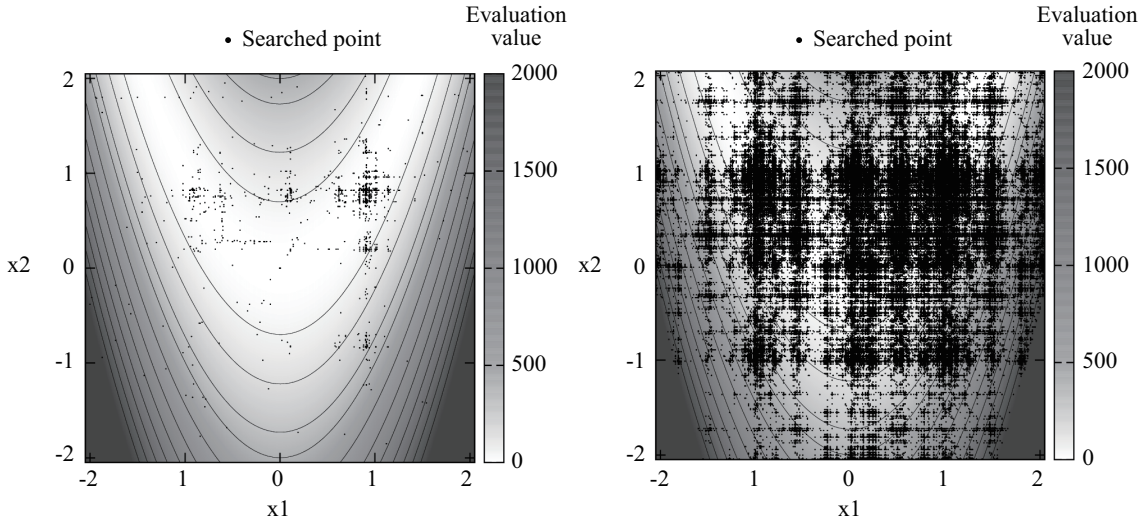
Fig. 17 Evaluation time depending the number of individuals in 1 CUDA kernel call.



(a) SGA with single core.

(b) GA with GAROP using GPU.

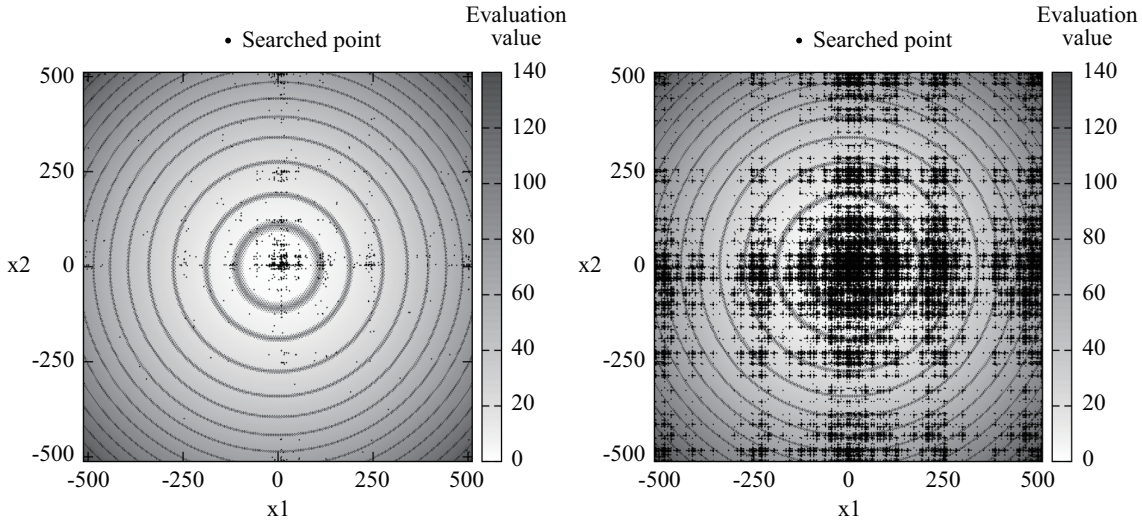
Fig. 18 Searched design variable area on rastrign's function.



(a) SGA with single core.

(b) GA with GAROP using GPU.

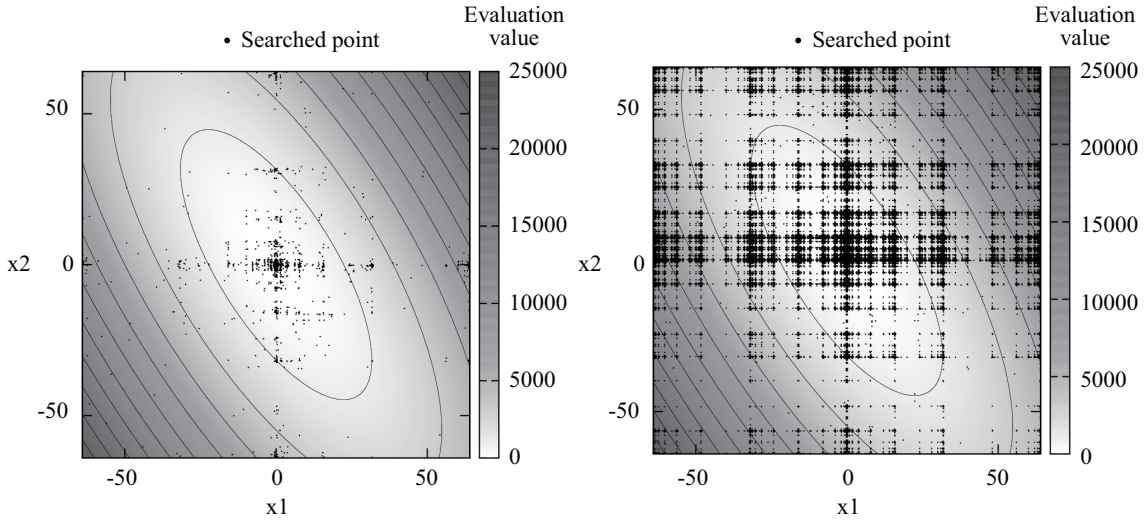
Fig. 19 Searched design variable area on rosenbrock's function.



(a) SGA with single core.

(b) GA with GAROP using GPU.

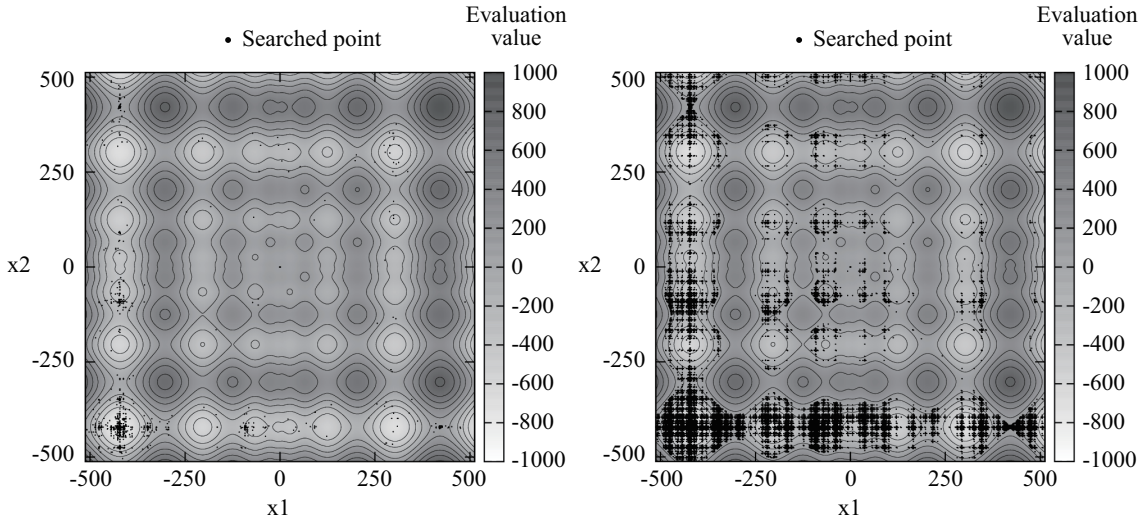
Fig. 20 Searched design variable area on griewank's function.



(a) SGA with single core.

(b) GA with GAROP using GPU.

Fig. 21 Searched design variable area on ridge's function.



(a) SGA with single core.

(b) GA with GAROP using GPU.

Fig. 22 Searched design variable area on schwefel's function.

Table 1 API of GAROP.

Name	Function
initialize	initialize parallel resources and create individual pool.
throw	throw an individual to individual pool.
get	get an evaluated individual from individual pool.
finalize	disconnect parallel resources and free memories for individual pool.
get_queue_num	get number of data in the get queue.
clear_pool	eliminate individuals in all queues.

Table 2 Specification of windows cluster.

OS	Windows Server 2008 HPC Edition
memory [GB]	8
processor	AMD Opteron 2356 × 2
clock rate [GHz]	2.30
number of nodes	16

Table 3 Specification of machine with multi-core CPU.

OS	Debian 5.0.10
memory [GB]	6
processor	Intel Xeon W3530 × 2
clock rate [GHz]	2.80
number of logical cores	8

Table 4 Specification of GPU.

architecture	Tesla C2050
global memory [GB]	2.68
number of multiprocessors	14
number of cores	448
clock rate [GHz]	1.15

Table 5 Parameter of SGA solving the conceptual design optimaization problem of HRE.

population size	64
chromosome length	41
max generation	32

Table 6 Parameter of SGA solving the 1-max problem.

population size	64
chromosome length	64
max generation	100

Table 7 Parameter of ga with GAROP.

population size	40
chromosome length	20
Evaluation time [msec]	25
max generation	50

List. 1 Simple GA with the GAROP (pseudo code).

```
population = InitPopulation();
Initialize(); // initialization of framework
FOR j = 0 to generation limit DO
  FOR i = 0 to population num DO
    // throw individuals to GA Pool
    Throw( population[i] );
  ENDFOR
  FOR
    // get individuals from GA Pool
    Get( population[i] );
  ENDFOR
  selection( population );
  crossover( population );
  mutation( population );
ENDFOR
Finalize(); // finalization of framework
```

List. 2 Island model GA with the GAROP (pseudo code).

```
population1 = InitPopulation();
population2 = InitPopulation();
Initialize(); // initialization of framework
FOR j = 0 to generation limit DO
  FOR i = 0 to population num DO
    // throw individuals to GA Pool
    Throw( population1[i] );
    Throw( population2[i] );
  ENDFOR
  FOR
    // get individuals from GA Pool
    Get( population1[i] );
    Get( population2[i] );
  ENDFOR
  selection( population1 );
  selection( population2 );
  crossover( population1 );
  crossover( population2 );
  mutation( population1 );
  mutation( population2 );
  IF j % 10 == 0 THEN
    migration()
  ENDIF
ENDFOR
Finalize(); // finalization of framework
```

List. 3 Template of evaluation function on C language.

```
void evaluate(
    unsigned char*  indata,
    unsigned char*  retdata,
    unsigned char** parameter
);
```

List. 4 Source code by users with the windows cluster environment.

```
Individual[] population = InitPopulation();
// initialization of GAROP
GAROP g = new GAROP();
for( j = 0; j < generation_limit; j++ ) {
  for( i = 0; i < population_size; i++ )
    // throw individuals to Individual Pool
    g.Throw( population[i] );
  for( i = 0; i < population_size; i++ )
    // get individuals from Individual Pool
    g.Get( population[i] );
  selection( population );
  crossover( population );
  mutation( population );
}
g.Finalize(); // finalization of GAROP
```

List. 5 Source code by users with the multi-core CPU environment.

```
Individual[] population = InitPopulation();
// initialization of GAROP
Initialize( sizeof(Individual) );
for( j = 0; j < generation_limit; j++ ) {
  for( i = 0; i < population_size; i++ )
    // throw individuals to Individual Pool
    Throw( (BYTE*)&population[i] );
  for( i = 0; i < population_size; i++ )
    // get individuals from Individual Pool
    Get( (BYTE*)&population[i] );
  selection( population );
  crossover( population );
  mutation( population );
}
Finalize(); // finalization of GAROP
```

List. 6 Source code by users with the GPU environment.sga-garop.

```
Individual[] population = InitPopulation();
// initialization of GAROP
Initialize( sizeof(Individual) );
for( j = 0; j < generation_limit; j++ ) {
  for( i = 0; i < population_size; i++ )
    // throw individuals to Individual Pool
    Throw( (BYTE*)&population[i] );
  for( i = 0; i < population_size; i++ )
    // get individuals from Individual Pool
    Get( (BYTE*)&population[i] );
  selection( population );
  crossover( population );
  mutation( population );
}
Finalize(); // finalization of GAROP
```

Algorithm 1 GAROP を利用した GA

$t \Leftarrow 0$;
初期母集団 $P(t)$ をランダムに生成 ;
$P(t)$ 内の個体を個体プールに登録 ;
個体プールから母集団サイズの個体を取得 ;
$P(t)$ よりエリート個体 E_p を保存 ;
while 終了条件が満たされない **do**
  $t \Leftarrow t+1$;
  $P(t-1)$ より複製選択で選ばれた個体群を $P(t)$ とする ;
  $P(t)$ に対して交叉処理を行う ;
  $P(t)$ に対して突然変異処理を行う ;
  $P(t)$ 内の個体を個体プールに登録 ;
  for $i = 0$ to $i <$ 母集団サイズ **do**
    while Get キュー格納個体数 $== 0$ **do**
      E_p の近傍を生成してプールに登録 ;
    end while
    $P(t)_i$ にプールから 1 個体取得 ;
    $i \Leftarrow i+1$;
  end for
  while Get キューの格納個体数 > 0 **do**
    $N(t)$ にプールからの取得個体を追加 ;
  end while
  $N(t)$ よりエリート個体 E_n を保存 ;
  if $E_n.fitness < E_p.fitness$ **then**
    $E_p \Leftarrow E_n$;
  end if
  $P(t)$ の最悪個体 $\Leftarrow E_p$;
  $E_p \Leftarrow P(t)$ の最良個体 ;
  プールをクリア ;
end while