GPUを用いた高速並列進化計算による組合せ最適化問題へのアプローチ

筒井　茂義

進化計算は，集団ベースの探索手法であることから並列計算に向いている．本稿では，進化計算を概観した後，進化計算の一手法であるアントコロニー最適化 (Ant Colony Optimization, ACO) にタブーサーチを組合せ，組合せ最適化問題のなかでも最も困難な問題の一つである 2 次割当て問題 (Quadratic Assignment Problem, QAP) を超多並列計算手法として注目されている GPU (Graphics Processing Unit) 計算により高速に解く方法を紹介する．

キーワード：進化計算，組合せ最適化問題，アントコロニー最適化，並列計算，GPU，タブーサーチ

1. はじめに

進化計算は，複数の個体（解候補）からなる集団を，対象問題（環境）における各個体の評価値を用いて，よりよい個体を持つ集団に進化させることにより解の探索を行うメタヒューリスティックスの総称である．進化計算は，数学的に定式化が困難な問題や組合せ爆発により厳密解を得ることが困難な問題に有効である．

進化計算としては，生物進化にヒントを得た遺伝的アルゴリズム (Genetic Algorithm, GA) が代表的な手法であるが，2 章で述べるように各種の手法がある．進化計算は集団ベース (population-based) の探索手法であることから並列計算に向いている．

本稿では，進化計算を概観した後，組合せ最適化問題のなかでも最も困難な問題の一つである 2 次割当て問題 (Quadratic Assignment Problem, QAP) をアントコロニー最適化 (Ant Colony Optimization, ACO) とタブーサーチとを組合せ，超多並列計算手法として注目されている GPU 計算により高速に解く研究について述べる [12].

2. 進化計算あれこれ

生物進化にヒントを得た進化計算の源流は 1960 年代にさかのぼり，進化戦略 (Evolution Strategy, ES), 進化プログラミング (Evolutionary Programming, EP) および遺伝的アルゴリズム (Genetic Algorithm, GA) の三つが挙げられる．これらの研究は 1980 年代の後半まではお互いに遭遇することなく研究されてきた．事実，1989 年に出版され GA の研究者に多大な影響を与えた Goldberg の著書 [1] には ES や EP の記述は一切見当たらない．EP に関してはお互いに無視しあっていたのかもしれない．ES に関しては，研究論文がドイツ語で書かれていたことが大きな原因と思われる．

1980 年代の後半からは，従来のコミュニティが互いに本格的に遭遇し，研究交流が国際会議を通して始まった．この交流は，例えば GA のコミュニティが重視している交叉が ES に取り入れられるというように方法論的イミグレーションを伴う．また，1990 年代に入り，従来の生物進化をベースとする手法に加え，群れの集団行動をモデルとする本研究で用いる ACO や粒子群最適化 (Particle Swarm Optimization, PSO)[2], さらには GA に統計的手法を融合する分布推定型アルゴリズム (Estimation of Distribution Algorithm, EDA)[3, 4] などの手法が，集団ベースの探索手法である進化計算の仲間に加わってきた．

すべての問題に万能なアルゴリズムは存在しないという事実がある [5]．各コミュニティはときには強い自己主張を行い，またときには排他的になりつつも，これらによりコミュニティの多様性を維持しつつ，相互に影響しながら発展してきた．また，今後も発展し続けるであろう．このように，進化計算コミュニティは，分散 GA の島モデルのようにして進化し，まさに Evolutionary である．

さて，進化計算により問題を解く際の解の表現法（コーディング法）には，(1) バイナリ表現：{1011 . . . 11}，(2) 実数値列表現：{2.12, 2.55, . . ., 5.55}，(3) 順列表現：{3, 2, 0, 5, 4, 1} の三つが代

つつい　しげよし
阪南大学
〒 580–8502 大阪府松原市天美東 5–4–33

表的である．バイナリ表現はいろいろな問題に適用できる汎用的な方法であり，理論的な研究が進んでいる．実数値列表現は，最適化設計問題など，実数値を扱う問題で最もよく使われている．順列表現は，スケジューリング問題など組合せ最適化問題で多く使われ，本稿の QAP の解法にも用いている．各進化計算と表現法との関係を表 1 にまとめた．

表 1　代表的な進化計算手法における解の表現法

名称	解の表現（コーディング）			
	バイナリ	実数値	順列	その他
–Genetic Algorithm (GA)	○	○	△	
–Genetic Programming (GP)				木構造
–Estimation of Distribution Algorithm (EDA)	○	○	○	
–Evolution Strategy (ES)		○		
–Evolutionary Programming (EP)		○		
–Ant Colony Optimization (ACO)		△	○	
–Particle Swarm Optimization (PSO)		○		
–Diffrencial Optimization (DE)		○		

3. アントコロニー最適化 (ACO)

ACO は，アリの群れによる採餌行動の際の経路生成過程にヒントを得た探索手法であり，巡回セールスマン問題 (TSP) など多くの組合せ最適化問題に適用され，その有効性が報告されている [7]．アリはフェロモンを介したコミュニケーションを行いながら群れで行動し，ある種の秩序を形成する．ACO では，この秩序形成過程を探索に用いる．ACO の基本モデルは，Dorigo らによる Ant System (AS)[6] と呼ばれるアルゴリズムである．その後，Ant Colony System (ACS)[8], Max-Min Ant System (MMAS)[9] など多くの改良型 ACO アルゴリズムが提案されている [7]．

AS はアリの行動原理に比較的忠実なアルゴリズムであるので，以下では，TSP の解法への AS の適用を例に ACO の概要を説明しよう．各アリを以下ではエージェントと呼ぶ．各エージェントは，各都市に均等もしくはランダムに配置され，そこを出発点として TSP の巡回路を形成する．このとき，各エージェントは，フェロモン濃度に比例して確率的に経路を選択する．二つの都市 i, j 間の経路（エッジ）のフェロモン濃度を τ_{ij} としよう．このとき，一度行った都市は訪問しないという TSP の規則に従う．この経路選択の過程を図 1 に示す．

同図 (a) は，都市 1 から出発するエージェント k を示している．都市 1 にいるエージェント k が次に訪問できる都市の集合は $\{2,3,4,5,6\}$ である．そこで，これらの都市 $j(j \in \{2,3,4,5,6\})$ を選択する確率 p_{1j}^k は，(a) に示されているようになる．(b) は，この確率に基づいて都市 2 が選ばれた状況である．都市 2 にいるエージェントが次に訪問できる都市の集合は $\{3,4,5,6\}$ であるので，このエージェントがこれらの都市 $j(j \in \{3,4,5,6\})$ を選択する確率 p_{2j}^k は (b) に示されているようになる．(c) は，この確率に基づいて都市 4 が選ばれた状況である．以下，同様に訪問する都市を順次決定してエージェント k は TSP の巡回路を完成させる．

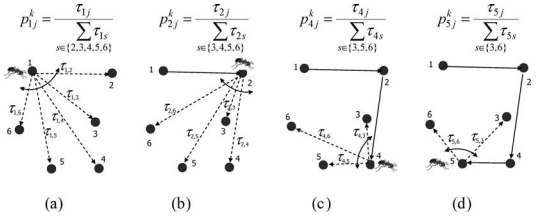

図 1　フェロモン濃度に基づく巡回路生成過程

各都市に配置された m 個のエージェントが TSP の巡回路を完成させる動作を 1 サイクルとする．このときフェロモン濃度は，次式によって更新される．

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^k. \tag{1}$$

ここで，ρ は蒸発係数と呼ばれ，(1–ρ) がサイクル t と t+1 との間にフェロモンが蒸発する割合を示す．また，$\Delta\tau_{ij}^k$ はエージェント k により経路 (i,j) に新たに排出されるフェロモン濃度である．この値は，エージェント k の巡回路 T_k の長さ C_k が短いほど大きな値となるようにするため，次式のように C_k の逆数とする．

$$\Delta\tau_{ij}^k = \begin{cases} 1/C_k, & \text{if agent } k \text{ uses edge } (i,j) \text{ in its tour } T_k, \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

初期状態ではすべての経路に同じ濃度 τ_0 のフェロモンが存在すると考える．すなわち，$\tau_{ij}(0) = \tau_0$ とする．図 1 で説明した選択確率は，ACO では式 (3) のように一般化されている．すなわち，サイクル t で都市 i にいるエージェント k が都市 j に移動する確率 $p_{ij}^k(t)$ を次式で定義する．

$$p_{ij}^k(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]\cdot[\eta_{ij}]^\beta}{\sum_{s\in J_k(i)} [\tau_{is}(t)]\cdot[\eta_{is}]^\beta}, & \text{if } j \in J_k(i), \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

ただし，$J_k(i)$ は，都市 i にいるエージェント k が訪問できる都市（まだ訪問していない都市）のリストであり，β はフェロモン濃度 $\tau_{ij}(t)$ と η_{ij} との重要性の度合いを制御するパラメータである．η_{ij} は式 (3) の $\tau_{ij}(t)$ に基づく選択確率にバイアスを与えるものであり，TSP の場合には経路の選択でより近くの都市を選ぶことが好ましいというヒューリスティックから，$\eta_{ij}=1/d_{ij}$(d_{ij} は都市 i,j 間の距離) が使われる．

なお，本稿で述べる QAP への ACO の適用では，そのようなヒューリスティックがないので $\eta_{ij}=1$ としている．また，TSP では，τ_{ij} には，都市 i と j とが隣接する好ましさの度合いを表しているが，QAP においては，τ_{ij} には，部門 i が場所 j に割当てられる好ましさの度合いを表している．このように ACO では，フェロモン濃度の定義は問題に依存する．式 (3) に基づいてすべてのエージェントが巡回路の生成を終了したとき，フェロモン濃度が式 (1) に基づいて更新され，$t \leftarrow t+1$ として終了条件が満たされるまでこのサイクルが繰り返される．

本稿で用いる ACO は筆者が開発した cAS[10, 11] である．cAS は，AS を拡張したもので，解を生成する際に既存解を部分的に借用し，残りの部分を式 (3) に基づいて生成する．これにより Exploration と Exploitation との均衡が図られ，高性能 ACO となっている．

一般に進化計算の応用では対象問題領域におけるローカルサーチを進化計算に組み合わせる場合が多い．図 2 に本稿における ACO アルゴリズムの全体の流れを示す．ここではローカルサーチに 4 章で述べるタブーサーチ (Tabu search, TS) を用いる．

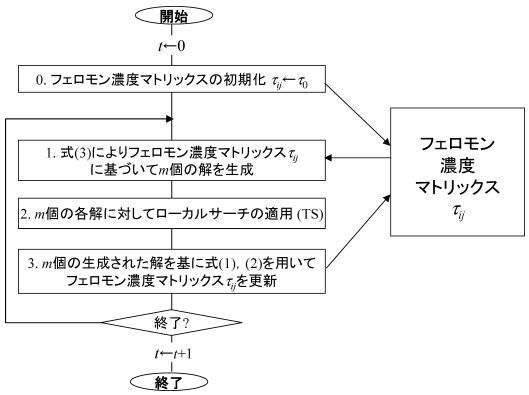

図 2　ACO とタブーサーチとの組合せ

4. QAP におけるタブーサーチ

4.1 QAP の概要

QAP は，n 個からなる部門を n 個の場所に，式 (4) で定義される関数値が最小になるように割当てを決定する組合せ最適化問題である．

$$f(\phi)=\sum_{i=1}^{n}\sum_{j=1}^{n} b_{ij}a_{\phi(i)\phi(j)}. \tag{4}$$

ここで，$A=(a_{ij})$ および $B=(b_{ij})$ はそれぞれ $n\times n$ のマトリックスであり，ϕ は $\{1, 2, \ldots, n\}$ の順列である．マトリックス A と B は，それぞれ，場所 i,j 間の距離，部門 i,j 間の流量（物流や人的交流の強さ）を表している．

QAP は式 (4) からもわかるように評価関数が距離と流量との積になっているため，TSP に比べてはるかに解くことが困難である．また，QAP はベンチマーク問題としてよく使われるが，実際の応用問題も多くある．例えば，大きなビルにおける部門の最適配置問題や，グローバル企業における事業所立地の最適配置問題などである．その他，2 次割当て問題は，最適割当て問題を一般化した形式になっているので，多くの割当て問題にも適用できる．

QAP の簡単な例を図 3 に示す．割当て状況は順列 ϕ で表現される．$\phi=\{2,1,4,3\}$ は，部門 1 を場所 2 に，部門 2 を場所 1 に，部門 3 を場所 4 に，部門 4 を場所 3 にそれぞれ割当てることを示す．図 3 の $f(\phi)$ の値は式 (4) から，$f(\phi)=1524$ である．

4.2 ローカルサーチとしてのタブーサーチ

QAP におけるローカルサーチとして 2-opt 法がよく知られている．一方，TS [13] は，組合せ最適化問題の解法に適用される強力なメタヒューリスティック

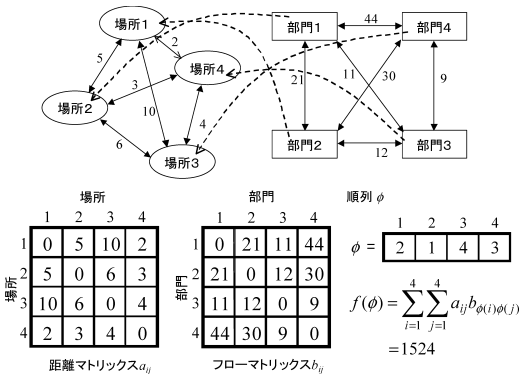

図 3　QAP の簡単な例 ($n=4$)

スの一つであるが，進化計算と組み合わせてローカルサーチとしてもよく用いられる [9].

本稿では，TS を ACO と組み合わせてローカルサーチに用いる．QAP における TS は 2-opt 法によく似たアルゴリズムであるが，2-opt 法では常に最良近傍に移動させるのに対して，TS では現在解の最良近傍解の値が現在解の値よりも悪くあっても最良近傍解に移動させる．この場合移動先の近傍探索において移動元の解が最良近傍になる場合が当然起こりうる．このような場合現在解が移動元に戻ってしまうことになる．TS はその名のとおり，そのような近傍への移動をある期間（禁断期間）禁止することを特徴とする．移動が禁止される近傍（禁断近傍）は禁断リスト (Tabu list) と呼ばれるデータ構造で管理される．

図 4 に本稿で用いた TS の擬似コードを示す．禁断近傍でもある基準（アスピレーション基準）を満たせば移動を許す場合がある．ここでは標準的な基準に倣い，最良解が得られた場合に移動を許すことにした．また，Taillard の Ro-TS[14] にならって，禁断期間はランダム要素を取入れ，$TS_{list\_size} \times r^3$ とした．ただし，r は [0,1) の一様乱数であり，TS_{list_size} はパラメータである．

```
記号の説明
φ: 現在の解, φ*: 繰り返し中に見つかった最良解
f*: f(φ*), N(φ): φの近傍,
N'(φ) : N(φ)の許容近傍:
        ➢ 禁断でない
        ➢ アスピレーション
φ0: スタート時の解, IT_TS-counter: カウンター

Tabu search(int[] φ0){
    set φ ←φ0; f*←f(φ0); φ*←φ0;
    Tabu list←∅; IT_TS_counter=0;
    while(IT_TS_counter<IT_TS_MAX){
        choose the best φ' ∈ N'(φ)
        if(f(φ') < f*){
            f*←f(φ'); φ*←φ';
        }
        update Tabu list; φ←φ'; IT_TS-counter ++;
    }
    φ0← φ*;
}
```

図 4　TS の擬似コード

4.3　QAP における移動コストの計算

図 4 からわかるように TS を QAP に適用する場合，現在の解 ϕ のすべての近傍 $N(\phi)$ への移動による適応度の変化量（以下，移動コストと呼ぶ）を計算しなければならない．近傍としては図 5 に示すように ϕ の二つの位置の値を交換したものを採用する．

ϕ' を ϕ の r 番目の要素と s 番目の要素を交換して得られた近傍解とすると，移動コスト $\Delta(\phi, r, s) = f(\phi') - f(\phi)$ の計算量は以下のように，$O(n)$ となる．

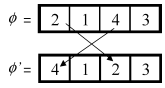

図 5　QAP における ϕ の近傍の例

$$
\begin{aligned}
\Delta(\phi, r, s) = {} & a_{rr}(b_{\phi(s)\phi(s)} - b_{\phi(r)\phi(r)}) \\
& + a_{rs}(b_{\phi(s)\phi(r)} - b_{\phi(r)\phi(s)}) \\
& + a_{sr}(b_{\phi(r)\phi(s)} - b_{\phi(s)\phi(r)}) \\
& + a_{ss}(b_{\phi(r)\phi(r)} - b_{\phi(s)\phi(s)}) \\
& + \sum_{k=1, k\neq r,s}^{n} \left(\begin{array}{l} a_{kr}(b_{\phi(k)\phi(s)} - b_{\phi(k)\phi(r)}) + \\ a_{ks}(b_{\phi(k)\phi(r)} - b_{\phi(k)\phi(s)}) + \\ a_{rk}(b_{\phi(s)\phi(k)} - b_{\phi(r)\phi(k)}) + \\ a_{sk}(b_{\phi(r)\phi(k)} - b_{\phi(s)\phi(k)}) \end{array} \right).
\end{aligned} \tag{5}
$$

ここで，もし $\Delta(\phi, r, s)$ の移動コストを記憶する $n \times n$ 個の記憶領域を用いると，ϕ' からの移動コスト $\Delta(\phi', u, v)$ は，$\{u, v\} \cap \{r, s\} = \emptyset$ が満たされる $\{u, v\}$ に対しては次式のようになり，その計算量は $O(1)$ となる [13].

$$
\begin{aligned}
\Delta(\phi', u, v) = {} & \Delta(\phi, u, v) \\
& + (a_{ru} - a_{rv} + a_{sv} - a_{su}) \\
& \times (b_{\phi'(s)\phi'(u)} - b_{\phi'(s)\phi'(v)} \\
& + b_{\phi'(r)\phi'(v)} - b_{\phi'(r)\phi'(u)}) \\
& + (a_{ur} - a_{vr} + a_{vs} - a_{us}) \\
& \times (b_{\phi'(u)\phi'(s)} - b_{\phi'(v)\phi'(s)} \\
& + b_{\phi'(v)\phi'(r)} - b_{\phi'(u)\phi'(r)}).
\end{aligned} \tag{6}
$$

5.　ACO にタブーサーチを結合したアルゴリズムの GPU 計算への実装

5.1　GPU の計算の概要

実装について述べる前に，GPU 計算による超並列計算の概要を簡単に述べておきたい [15].

5.1.1　GPU アーキテクチャ

図 6 は，代表的な GPU の一つである NVIDIA 社の GTX285 のアーキテクチャである．GPU 内におけるプロセッサは，スレッドプロセッサ（thread processor, 以下 TP）と呼ばれ，8 個の TP が一つのグループとしてマルチプロセッサ (Multi-Processor, MP) を構成している．MP 内の TP は，16 KB の高速共有メモリ（shared memory, 以下 SM）を介してデータを共有することができる．一方，MP 間のデータ共有は

VRAM を介して行われる．表 2 に代表的 GPU の仕様を示す．GPU 計算では VRAM が GPU のメインメモリとなる．

プログラムはスレッドとして実行されるが，注意しなければならないことは，各 MP では，各スレッドはウォープ (warp) と呼ばれる 32 スレッドを単位として SIMD (Single Instruction Multiple Data) 風に実行されることである．したがって，ウォープ内のスレッドの計算が分岐などにより相互に大きく異なった場合には処理待ちに伴うアイドリングタイムが発生する．

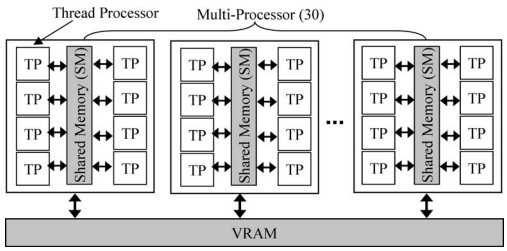

図 6　GPU アーキテクチャ例 (GTX 285)

表 2　GPU の仕様

GPU	GTX285	Fermi アーキテクチャ	
		GTX480	GTX580
コア/MP	8	32	
総コア数	240	480	512
プロセッサクロック	1477MHz	1401MHz	1544MHz
最大スレッド数/MP	1024	1536	
最大スレッド数/ブロック	512	1024	
SM	16KB	16KB/48KB	
L1 キャッシュ	なし	16KB/48KB	
L2 キャッシュ	なし	768KB	
メモリ転送バンド幅	159GB/秒	177.4GB/秒	192.4GB/秒

5.1.2　CUDA プログラミングモデル

GPU 計算では，NVIDIA 社が GPU 計算向けに提供している C 言語を拡張した統合開発環境 CUDA (Compute Unified Device Architecture) が最もよく利用されている．

CUDA のプログラミングモデルは，基本的にマルチスレッドプログラミングである．図 7 に CUDA のプログラミングモデルを示す．CUDA プログラムでは，スレッドはグリッド (grid) とブロック (block) の 2 階層構成をとる．ブロックは，スレッドの集合であり，1 次元，2 次元，または 3 次元構成をとることができる．一方，グリッドはブロックの集合であり，1 次元または 2 次元構成をとることができる．

各スレッドは，カーネル関数 (kernel function) に記述された同じコードを実行する．スレッドのスケジューリングは，ハードウェアにより自動的に行われる．カーネル関数は，通常のデータ引数の他，グリッドとブロックの定義を引数としてもつ．カーネル関数がコールされると，グリッドとブロックの定義にしたがってスレッドが生成され，それらのスレッドが一斉に実行を開始する（図 7 参照）．

MP のリソースである SM やレジスタ（ローカル変数はレジスタに割当てられる）は，一つの MP に同時に割当てられるブロック間で分割して使われる．したがって，一つの MP に割当てられるブロック数は，これらの制限によって決まり，同時に実行されるスレッド数もこのブロックの割当て状況に依存する．

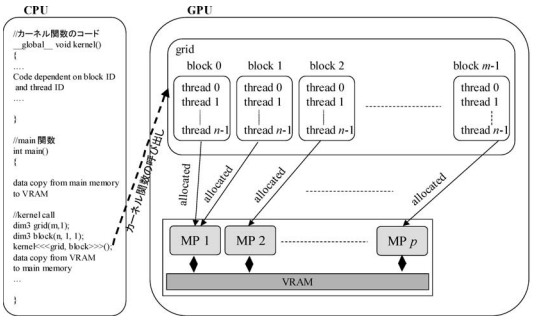

図 7　CUDA プログラミングモデル

5.2　CUDA アーキテクチャに基づく並列 ACO の全体構成

進化計算を GPU で高速並列計算させる研究は，既に多く試みられている [18–21]．本研究において，QAP を解くための CUDA アーキテクチャに基づく並列化 ACO の全体構成を図 8 に示す．今回の実装に当たっては，ACO の各ステップの機能はすべて GPU で実行されるカーネル関数としてコード化した（図 8 には主要な三つのカーネル関数，Construct_solutions (...), Apply_TS (...) および Update_Pheromone_Density (...) を示している）．CPU は ACO の各カーネル関数を順次呼び出して ACO の繰返し制御を行うのみである．これにより，CPU と GPU との間のデータ転送は，GPU から CPU へ探索の進行状況および最終解のデータ転送のみである．したがって，この方式では，CPU と GPU との間の通信オーバヘッドは小さくなる．

VRAM には，エージェントの集団，フェロモンマトリックス τ_{ij}，TS で用いる作業領域（4.3 節の $n \times n$ の記憶領域 $\Delta(\phi, r, s)$），タブーリストならびに QAP のデータ（距離マトリックス A= (a_{ij}) およびフローマトリックス B= (b_{ij})）を配置している．

図 8 の構成では，SM に配置するデータは各スレッド

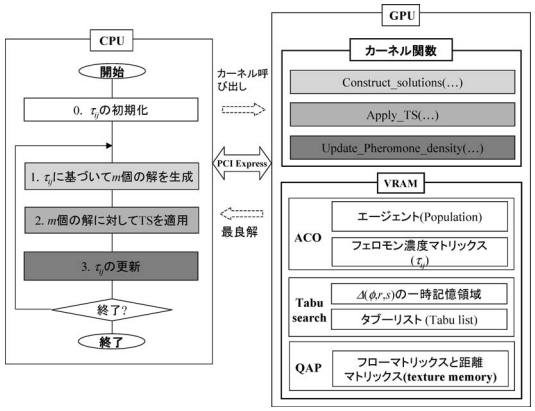

図 8　ACO の GPU への実装の全体構成

がブロック内で共有しなければならなく，かつ高速アクセスが必要なものに限定した．実験には Fermi アーキテクチャに基づく GTX480 を用いているが，SM のサイズが 16 KB, L1 キャッシュが 48 KB となるモードに設定（デフォルト設定）し，VRAM アクセスへの高速化を優先した（表 2 参照）．

5.3　GPU による移動コスト計算の並列化の課題

表 3 に，図 2 の ACO の処理を CPU によりシーケンシャル実行した場合の各ステップの処理に要する時間分布を示した．この表からわかるように，TS が占める時間が 99.9%以上となっている．この処理時間の内訳から高速処理のためにはステップ 2 の TS の効率的な実装が重要であることがわかる．したがって以下では，TS の部分（図 8 におけるカーネル関数 Apply_TS (...)）を中心にその並列化について述べる．

表 3　ACO アルゴリズム処理時間の内訳

QAP	解生成	TS	τ_{ij}更新
tai50a	0.01%	99.99%	0.00%
tai100a	0.00%	100.00%	0.00%
tai50b	0.02%	99.98%	0.00%
tai100b	0.01%	99.99%	0.00%

カーネル関数 Apply_TS (...) では，各ブロック内で TS の繰返しごとに現在解 ϕ の近傍の移動コスト $\Delta(\phi,u,v)$ の計算をしなければならない．ϕ の近傍 $N(\phi)$ のサイズを C とすると $C = n(n–1)/2$ である（n は問題サイズ）．TS の計算時間がアルゴリズム全体の大部分を占める理由は，これら C 個の移動コストの計算に多くの時間を要するからである．したがって，この処理をブロック内で効率よく並列実行させることが重要となる．並列化を行う単純な方法は，各近傍に対して図 9 のように番号を付与し，ブロック内でこの番号に対応する C 個のスレッドを並列に実行させる方法である．同図において，$n = 15, C$ =105 であり，現在解 ϕ' が ϕ の 5 と 11 の要素を入れ替えて得られたものと仮定し，ϕ' の移動コストの計算量が $O(n)$ であるものを背景が黒の文字で示している．

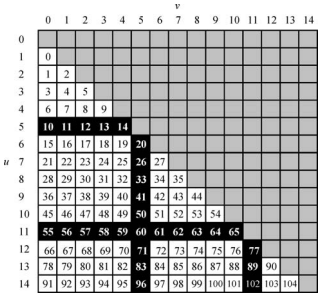

図 9　近傍の番号付け

このような単純な方法では，まず次のような問題が発生する．C 個の近傍のうち，式 (5) により $O(n)$ の計算量で移動コストの計算ができる近傍数は $C_{O(n)} = 2n-3$ であり，式 (6) により $O(1)$ の計算量で移動コストを計算できる近傍数は $C_{O(1)} = (n-2)(n-3)/2$ である．表 4 は各 n に対する $C, C_{O(1)}$ および $C_{O(n)}$ を示している．これからわかるように，$C_{O(n)}$ は C の 10%以下であり，問題サイズが大きくなるにしたがってその割合は小さくなる．各スレッドは 32 スレッドを単位（ウォープ）として SIMD 風に実行されるが（5.1.1 項参照），このようにわずかな割合である $O(n)$ の計算量のスレッドが，大部分を占める $O(1)$ の計算量のスレッドと同じウォープ内で混在すると，そのウォープに属するスレッドの処理時間は図 10 に示すように $O(1)$,$O(n)$ ともアイドリング時間が生じ，処理時間が長くなる．

表 4　近傍サイズの例

問題サイズ	C / $n(n-1)/2$	$C_{O(1)}$ / $(n-2)(n-3)/2$	$C_{O(n)}$ / $2n-3$	$\frac{C_{O(n)}}{C}$
40	780	703	77	9.87%
50	1225	1128	97	7.92%
60	1770	1653	117	6.61%
80	3160	3003	157	4.97%
90	4005	3828	177	4.42%
100	4950	4753	197	3.98%

もう一つの問題は，一つのブロックにおける最大スレッド数の制限である．Fermi アーキテクチャでは一つのブロックにおける最大スレッド数が 1024 である（表 2 参照）．表 4 からも明らかなように例えば問題サ

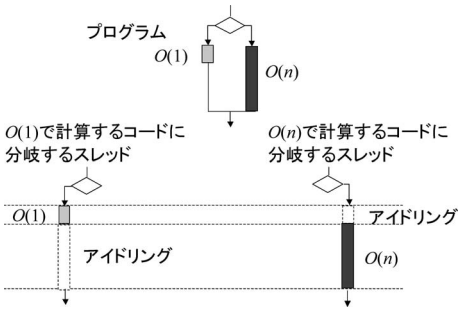

図 10　ウォープ内のアイドリング時間によるスレッドの遅延

イズ 50 でも近傍サイズ C は 1225 であり，既にこの問題で最大スレッドの制限を超えてしまう．

5.4　移動コスト計算の効率的並列計算法 MATA

5.3 節で述べた問題に対して以下のような工夫を行い実装した．

- $O(1)$ の処理を行うスレッドと $O(n)$ の処理を行うスレッドとを別のウォープとなるようにスレッド番号を割当てる．
- $O(n)$ の計算を行うスレッドには一つの近傍計算のみを割当てるのに対して，軽い処理である $O(1)$ の計算を行うスレッドには，$N_{O(1)}(N_{O(1)}>1)$ 個の近傍計算を割当てる．

これを実現する方法には各種の方法が考えられるが，ここでは処理オーバヘッドを伴わない以下のような方法を用いる．まず，交換ペア（近傍）(u, v) に対して，図 9 に示したように番号 $k(k=0, 1, 2, \ldots, C-1)$ を付与する．あるスレッド番号 $t=\lfloor k/N_{O(1)} \rfloor$ には，k が $t \times N_{O(1)}$ から $t \times N_{O(1)} + N_{O(1)} - 1$ までの近傍の計算を割当てる（$O(1)$ の処理の数は (n–2)(n–3)/2 であるが，実装の容易さから C 個割当て，k が $O(n)$ となる近傍計算の場合には何も計算しないこととする）．したがって，$O(1)$ の近傍計算が割当てられるスレッドの総数は $T_{O(1)} = \lceil C/N_{O(1)} \rceil$ となる．

$O(n)$ の近傍計算は，ウォープのサイズが 32 であることから，32 単位の区切れ目となるスレッド番号 $T_{O(n)\_start} = \lceil T_{O(1)}/32 \rceil \times 32$ から始まる $T_{O(n)} = 2n$ のスレッドに $O(n)$ の処理を割当てる（$O(n)$ の個数は $2n$–3 であるが，ここでは先の場合と同様，実装の容易性から $2n$ 個のスレッドを使い，不要な組合せの 3 個のスレッドは何もしない）．

以上から，ブロック内において使用される総スレッド数は $T_{total} = T_{O(n)\_start} + 2n$ となる．このスレッド割当て法を MATA (Move-Cost Adjusted Thread Assignment) と呼ぶ．図 11 に MATA による近傍コスト計算のスレッドの構成を示す．この方式により，各スレッドのアイドリング時間を大幅に減少させることができる．

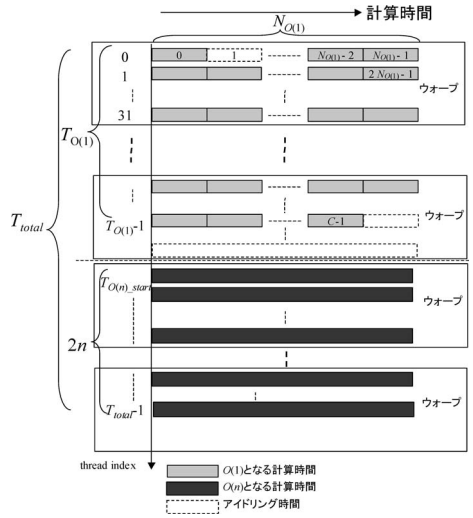

図 11　MATA による移動コスト計算のブロック内のスレッド構成

6.　並列計算の高速化の効果の実験

6.1　実験条件

本実験で用いた計算機はインテル社の Core i7 965 (3.2 GHz) プロセッサと NVIDIA 社の GTX480（表 2 参照）を 1 個搭載した PC である．OS は Windows XP Professional で CUDA プログラムのコンパイルには，Microsoft Visual Studio 2008 Professional Edition（最適化オプションは/O2）および CUDA SDK 4.0 を用いた．

テスト問題には QAPLIB ベンチマークライブラリ [16] の問題を用いる．QAPLIB のテスト問題は，1) ランダム生成問題，2) グリッド距離ベースランダム問題，3) 実問題，4) 実問題に似せて生成した問題，の 4 クラスに分けられる [9]．このうち，本実験では，Taillard によるクラス 1) および 4) の問題を用いる．Taillard のクラス 1) の問題には tai*a シリーズ，クラス 4) の問題には tai*b シリーズがあるがここでは問題サイズが 40 から 150 までの問題，すなわち，tai40a, tai50a, tai60a, tai80a, tai100a（以上クラス 1) の問題）および tai50b, tai60b, tai80b, tai100b, tai150b（以上，クラス 4）の問題) の 10 個の問題を用いる．クラス 1) のランダム生成問題は極めて困難な問題として知られ，またクラス 4) の実問題風に生成された問題は，クラス

表 5 パラメータ値

パラメータ		値	
		クラス 1) QAP	クラス 2) QAP
ACO	エージェント数 m	n	n
	蒸発係数 ρ	0.4	0.4
TS	$IT_{TS\ MAX}$	$64n$	n
	$IT_{list-size}$	$4n$	$n/2$
	$N_{o(1)}$	$n/4$	$n/4$

表 6 ACO に TS を結合した場合の GPU 計算の効果

QAP instances		GPU 計算 (GTX480) 平均実行時間 T_{avg}（秒） MATA	non-MATA	$\frac{\text{non}-\text{MATA}}{\text{MATA}}$	平均誤差 *Error* (%)	GPU 計算 (i7 965 3.2GHz) $T_{avg}(sec)$	平均誤差 *Error* (%)	スピードアップ (T_{avg}) $\frac{\text{CPU}}{\text{MATA}}$
クラス 1)	tai40a	9.5	70.6	7.4	0.14	191.0	0.14	20.0
	tai50a	18.3	136.0	7.4	0.34	463.7	0.35	25.3
	tai60a	32.6	269.1	8.3	0.32	962.8	0.36	29.6
	tai80a	154.9	728.2	4.7	0.41	3108.7	0.35	20.1
	tai100a	431.8	2352.9	5.4	0.36	7894.3	0.33	18.3
クラス 4)	tai50b	0.3	1.7	6.4	0	6.8	0	24.9
	tai60b	0.5	4.0	7.9	0	15.4	0	30.5
	tai80b	5.5	30.6	5.6	0	145.0	0	26.5
	tai100b	14.6	80.8	5.5	0	374.4	0	25.7
	tai150b	2893.2	16348.8	5.7	0.07	48948.6	0.05	16.9
総合平均		—	—	6.4	—	—	—	23.8

1) に比べてやや容易である．

ACO により生成された一つの解に対する TS の繰返し回数を IT_{TS_MAX}（図 4 参照），ACO の最大繰返し回数 IT_{ACO_MAX} とすると $IT_{TOTAL} = m \times IT_{ACO\_MAX} \times IT_{TS\_MAX}$ は，アルゴリズム中の TS の総繰返し数となる．この実験では $IT_{TOTAL} = m \times n \times 3200$ に固定し，アルゴリズムの終了条件を TS の総繰返し数が IT_{TOTAL} に達した，もしくは最適解が得られたときとした．表 5 にその他の代表的なパラメータを示す．

6.2 実験結果

表 6 に実験結果を示す．この実験では，各問題に対して (1)MATA を用いた GPU 計算，(2)MATA を用いない，すなわち，ウォープ内で $O(1)$ と $O(n)$ の混在を許す GPU 計算（以下，non-MATA）および (3) シングルスレッドで行った CPU 計算の 3 種類の実験を行い，実行時間および最適解からの誤差 (%)（(実行で得られた最良解 − 最適解)/最適解 ×100）の 10 回の実行の平均を同表に示した．なお，non-MATA では，一つのブロック内のスレッド数は $\lceil C/N_{O(1)} \rceil$ としている．

まず，MATA と non-MATA の T_{avg} 実行時間の比を見ると，tai100a では 5.4 倍，tai100b では 5.5 倍 MATA の方が高速である．全体の平均で見ると 6.4 倍高速となっている．これは 5.4 節で述べた MATA が有効であることを示している．このように，GPU 計算では問題の性質に応じて並列化の工夫が必要になる．本稿では言及しなかったが，メモリアクセスにおける遅延を最小化するような工夫も必要になる．

つぎに GPU 計算と CPU 計算の T_{avg} の比を MATA で見ると，GPU 計算は CPU 計算に対して 16.9〜30.5 倍高速になることを示しており，全体の平均で見ると 23.8 倍の高速化が得られた．

7. むすび

本稿では，進化計算を概観した後，組合せ最適化問題のなかでも最も困難な問題の一つである 2 次割当て問題を取上げ，アントコロニー最適化とタブーサーチとを組合せ，GPU 計算により高速に解く方法について述べた．

GPU 計算はエネルギー効率からも CPU 計算よりも有利であるといわれている．一般の科学技術計算では，東京工業大学の TSUBAME2 に代表されるように大規模なスーパーコンピュータが GPU を組み合わせて開発されている．進化計算の応用にはスケジューリング問題など実時間で高速に解を得ることが必要な問題が多くある．本稿では単一の GPU を用いる並列進化計算の実装について述べたが，複数の GPU 環境

でより高速に進化計算のアルゴリズムを実行する研究などが今後重要になると思われる．海外では既にこのような研究の例が見られる（例えば EASEA[18]）．

GPU の世代交代は 1.5 年ごとに進んでいる．今後はプログラミング環境を含めてより汎用的な開発環境が出現すると思われ，より多くの分野に適用を広めていくためにはその発展が期待される．

参考文献

[1] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[2] J. Kennedy and R. C. Eberhart. Particle swarm optimization, *Proc. of the IEEE Int. Conf. on Neural Networks*, pp. 1942–1948, 1995.

[3] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distribution I. binary parameters, *Proc. of the Parallel Problem Solving from Nature (PPSN IV)*, pp. 178-187, 1996.

[4] P. Larranaga and J. A. Lozano (eds). *Estimation of distribution algorithms*. Kluwer Academic Publishers, 2002.

[5] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Trans. on Evolutionary Computation*, 1(1): 67–82, 1997.

[6] M. Dorigo, V. Maniezzo and A. Colorni. The ant system: Optimization by a colony of cooperating agents, *IEEE Trans. on SMC-Part B*, 26(1): 29–41, 1996.

[7] M. Dorigo and T. Stützle. *Ant Colony Optimization*, MIT Press, Massachusetts, 2004.

[8] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Trans. on EC*, 1(1): 53–66, 1997.

[9] T. Stützle and H. Hoos. Max-min ant system. *Future Generation Computer Systems*, 16(9): 889–914, 2000.

[10] S. Tsutsui. *c*AS: Ant colony optimization with cunning ants. *Proc. of the 9th Int. Conf. on Parallel Problem Solving from Nature (PPSN IX)*, pp. 162–171, 2006.

[11] 筒井茂義. *c*AS：カニングアントを用いた ACO の提案. 人工知能学会論文誌, 22(1): 29–36, 2007.

[12] S. Tsutsui and N. Fujimoto. ACO with Tabu Search on a GPU for Solving QAPs using Move-Cost Adjusted Thread Assignment, *Proc. of the Genetic and Evolutionary Computation Conference (GECCO 2011)*, pp. 1547–1554, 2011.

[13] F. Glover and M. Laguna. *Tabu Search*. Kluwer, Boston, 1997.

[14] E. Taillard. Comparison of iterative searches for the quadratic assignment problem. *Location Science*, 3(2): 87–105, 1995.

[15] NVIDIA, 2010. www.nvidia.com/page/home.html.

[16] QAPLIB—a quadratic assignment problem library, 2009. www.seas.upenn.edu/qaplib.

[17] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma and P. Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA, *Genetic and Evolutionary Computation Conference*, pp. 1403–1410, 2009.

[18] H. Bai, D. OuYang, X. Li, L. He and H. Yu. Max-Min ant system on GPU with CUDA, *Int. Conf. on Innovative Computing*, pp. 801–804, 2009.

[19] W. Banzhaf, S. Harding, W. Langdon and G. Wilson. Accelerating genetic programming through graphics processing units. *Genetic Programming Theory and Practice VI*, 12(12): 1–19, 2009.

[20] M. L. Wong. Parallel multi-objective evolutionary algorithms on graphics processing units, *Genetic and Evolutionary Computation Conference (Companion)*, pp. 2515–2522, 2009.

[21] T. V. Luong, N. Melab and E.-G. Talbi. Parallel hybrid evolutionary algorithms on GPU, *IEEE Congress on Evolutionary Computation*, pp. 2734–2741, 2010.