GPU によるマルチエージェント・シミュレーション用ライブラリ MasCL の設計と実装

先山賢一 ○芳賀博英 （同志社大学）

Design and implementation of Multi Agents Simulation Library MasCL by GPU

K. Sakiyama and *H. Haga (Doshisha University)

Abstract− This article describes the design and implementation of library named MasCL for large-scale multi agents simulation. Currently, Multi Agent Simulation (MAS) is often used to simulate complex social phenomena such as evacuation behavior and traffic flow by economists and sociologists. Though simulation models of MAS with a single core CPU does not provide sufficient performance to simulate large scale models that often contain over hundreds of thousands elements. Therefore some researchers studied parallel execution to speed-up and scale-up simulations.
Recently, General Purpose computations on GPUs (GPGPU) is becoming increasingly popular for computing other than graphic computation. Some previous works showed that GPGPU contributes effective computing for the large scale MAS. However all previous works about MAS with GPU concentrate on the optimized algorithms such as how to let GPU compute agent actions simultaneously.The way to visualize phenomena that are produced by the interactions of agents is ignored. In MAS, Observing the results of agent actions and their interactions is important to analyze phenomena. However there is no frameworks and libraries which programmers easily use to describe and visualize the simulation with GPU.
In order to solve this problem, we have developed the library for MAS with GPU, which is called MasCL. MasCL consists of OpenCL and OpenGL to render agents and the space where they exist. By using MasCL, programmers can implement large scale multi agent simulation easily.

Key Words: Multi Agent Simulation, Large Scale Simulation, GPU, OpenCL, Parallel Processing

1 はじめに

本論文では，GPU を用いた並列計算の標準フレームワークである OpenCL を利用したマルチエージェント・シミュレーション (Multi Agents Simulation: MAS) 用のライブラリである MasCL の設計と開発について述べる．開発した MasCL を複数のマルチエージェントモデルの記述に適用することでその有用性を示すとともに，今後の展望について述べる．

MAS を CPU の単一プロセッサのみで処理する場合，シミュレーションするモデルのスケールによっては実用的な速度を達成することが困難である．それはモデルのスケールの増大に伴いエージェント数が増加するため，すべてのエージェントの処理に時間を要してしまうからである．この課題の解決のため，これまでいくつかの研究が行われてきている．例えば，蟻川と村田[1] は，グリッド・コンピューティングを用いることで，今までにない大規模な Sugarscape モデル[2] をシミュレーションできると考え，空間を複数に分割し，エージェントの処理を分割した空間ごとに分散させるアルゴリズムを提案して実験を行った．だが，グリッド・コンピューティングは高価な計算機設備と計算機同士の通信を行うための高度なプログラミング技術が必要不可欠となってしまい，容易に行えるものではない．

MAS においては，個々のエージェントは基本的に同一の処理を行う．そのため，SIMD(Single Instruction Multiple Data) 型の並列処理が有効であると考えられる．近年，グラフィックスプロセッサ (GPU) の並列演算性能を汎目的計算に用いる技術である GPGPU(General Purpose computations on GPUs) が並列処理の分野で広く普及している．GPU の内部には数百個の演算ユニットが搭載されており，同時に複数のデータに対し同一のプログラムを実行するような SIMD 型の計算に適している．Mikola Lysenko，Roshan D'Souza，Keyvan Rahmani[5] たちは，Sugarscape モデルの GPU による並列処理可能なアルゴリズムを考案して実装を行った．この報告から MAS に GPGPU を適用することで，大規模スケールのシミュレーションの高速化が行えることを示した．

しかし，従来の GPGPU を利用した MAS フレームワークの研究は，そのすべてがエージェント処理のアルゴリズムに焦点を当てたものであり，MAS で重要となる現象の視認や観察などを軽視したものである．そこで，本研究では，並列計算の標準フレームワークである OpenCL を利用した MAS ライブラリである MasCL を開発した．MasCL は画面表示にグラフィックスライブラリである OpenGL を利用しており，高速に空間とエージェントの描画を行うことが可能である．そのため，ユーザは MasCL を用いることにより，並列処理アルゴリズムを書くだけでシミュレーションを行うことが可能になる．

2 General Purpose computations on GPUs(GPGPU)

2.1 CPU と GPU

グラフィックスプロセッサを汎用計算の目的で使用する技術である GPGPU (General Purpose computations on GPUs) は，GPU の高い並列演算性能を使用できるので，様々な分野で広く普及している．

CPU と GPU はアーキテクチャが大きく異なっている．CPU は逐次プログラムが高速に実行できるように設計されている．1 スレッドからの命令を並行して実行できるように，あるいは逐次実行に見せかけながら

*現在フューチャーアーキテクト（株）

Table 1: CPU と GPU のスペック

	CPU	GPU
製品名	Core i7-3960X	Radeon HD7970
製造元	Intel	AMD
コアの数	6	**2048**
最大クロック周波数 [GHz]	3.9	0.925
最大メモリ帯域幅 [GB/秒]	51.2	**264**
最大メモリサイズ [GB]	64	3

順不同で実行できるような，高度で複雑な制御ロジックが組み込まれている．また，汎用性を考慮してアーキテクチャを進歩させる必要もあるため，演算能力を急激に向上させることが難しい．一方 GPU は，グラフィックス処理専用の演算装置である．コンピュータグラフィックスは並列に処理することが可能な計算が多いため，GPU は並列計算の実行速度を重視して開発されている．演算性能が低い小さな演算ユニットを多数搭載し，それらは完全にマルチスレッド化され，単一の命令をインオーダー形式で実行する．加えて，CPU のように従来アーキテクチャとの互換性を考慮する必要がない．またメモリの規格も最新のものを使用し高速にすることができる．Table.1 はコンシューマ向けに販売されている CPU と GPU の仕様を比較したものである．Intel の Core i7-3960X は 3.9GHz のコアを 6 個搭載しているのに対し，AMD の Radeon HD7970 は 0.925GHz のコアと演算性能は低いが，2048 個のコアが搭載されて並列処理の性能を上げている．また，メモリと演算ユニット間のデータ転送速度を表すメモリ帯域幅も，HD7970 は 264GB/秒と非常に高速に動作させることが確認できる．

2.2 データ転送

Fig.1 は CPU と GPU のデータアクセスの関係を簡潔に表したものである．GPU 側のメモリである VRAM はメインメモリから独立しており，互いにデータを共有していない．そのため，何らかのデータを GPU に処理させる場合には，メインメモリから VRAM にデータ転送をする必要がある．GPU が計算した結果を CPU が受け取る場合も同様である．加えて，その転送速度は CPU と GPU のメモリ帯域幅に比べて低速である．GPGPU による高速化の恩恵を受けるためには，アプリケーション開発者はこのデータ転送を最小限にするアルゴリズムを実装しなければならない．

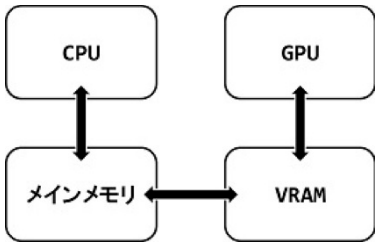

Fig. 1: Data transfer between CPU and GPU

3 GPGPU プラットフォーム

GPU を汎用計算に用いる場合には，GPU 特有の各種の規約があり，それゆえに一からプログラムを開発するのは困難である．そこで通常は，GPU 特有の様々な制約を抽象化したモデルであるプラットフォームを用いる．GPU 計算のプラットフォームとして，現在広く用いられているものとして，NVIDIA 社の GPU に特化したプラットフォームである CUDA(Compute Unified Device Architecture)[10] と非営利団体である Khronos Group が標準を策定している OpenCL(Open Computing Language)[11] がある．本研究では，汎用性を考慮して OpenCL を対象とした．

3.1 OpenCL プラットフォーム

OpenCL(Open Computing Language) とは，非営利団体である Khronos Group が標準を策定している，並列計算機環境に適した並列プログラミングのための標準フレームワークである [11]．CPU や GPU だけでなく，Cell.B.E や DSP など，ハードウエアに依存しない並列計算を行えることを目的として提案された．

OpenCL では制御側のプロセッサと演算側のプロセッサを分類しており，それぞれがホストと OpenCL デバイス (Compute Device) として定義されている．OpenCL デバイスは中に複数の CU(Compute Unit) を搭載しており，CU は実際に計算を行う PE(Processing Element) を複数持つ，というモデルとなっている．

3.2 OpenCL プログラミングモデル

OpenCL デバイス内の各 PE が別々のデータを処理するために，OpenCL はインデックス空間という概念を提供している [12]．インデックス空間は複数のワークグループに分割され，ワークグループは並列計算の最小単位であるワークアイテムに分割される．インデックス空間は OpenCL デバイスに，ワークグループは CU に，ワークアイテムは PE にそれぞれ対応して実行される．ハードウェアに依存するが，インデックス空間とワークグループの次元，ワークグループの数とワークアイテムの数は任意に設定できるため，チューニングを行い OpenCL デバイスのリソースを使い切ることが可能となる．

Fig.2 は，通常の C 言語のコードと OpenCL 用に並列化したコードを比較したものである．OpenCL で並列に実行されるプログラムコードはカーネルと呼ばれ，C 言語に文法が類似した OpenCL C 言語で記述される．Fig.2 の例では 2 つの配列 `a` と `b` の同一インデックスの値を加算して，配列 `c` に格納している．C 言語の場合は for ループを使用して各要素に対して同一の処理を行なっている．それに対して OpenCL C のコードでは，組み込み関数の `get_global_id` を使用して，各ワークアイテムが別々の値を取得することができる．このような組み込み関数を用いることで，それぞれのワークアイテムが同一のコードを実行しながら別々のデータ

を処理することが可能となる．

```
void addVec(int n, float *a, *b, *c)
{
   for (int i =0; i<n; i++)
      c[i] = a[i] + b[i];
}
```

(a) Ordinal C program code

```
__kernel
void addVec(__global float *a,
            __global float *b,
            __global float *c)
{
   int idx = get_global_id(0);
      c[idx] = a[idx] + b[idx];
}
```

(b) OpenCL C program code

Fig. 2: Ordinal C code and its corresponding OpenCL C code

4 MasCL ライブラリの開発

4.1 GPU を利用した MAS の開発

GPU を利用した MAS を開発するためには，GPGPU が使用できるフレームワークとエージェントと空間を描画するための GUI フレームワークが必要となる．OpenCL から GPU による並列処理を利用する場合，OpenCL の初期化処理が必要となる．具体的には，使用するプラットフォームの選択やデバイスの指定，OpenCL デバイスに命令を発行するためのキューの生成や OpenCL デバイスが使用するバッファの確保などである．OpenCL は多くのハードウェアを低レイヤな部分まで扱えるように設計されているため，その初期化処理は複雑である．加えて，OpenCL で計算した結果を GUI フレームワークに渡し，エージェントと空間を描画しなければならない．そのため，並列に MAS を行うという本質以外の部分で工数が発生し，シミュレーションの実装まで時間を要してしまう．そこでこれら細部の情報を隠蔽し，容易にプログラムが作成できるように，ライブラリを開発することとした．

4.2 言語と環境

ライブラリの実装には C++言語を用いた．MasCL 利用者は C++言語からライブラリを使用できる．並列計算を担う部分は OpenCL を採用し，シミュレーションの表示方法には OpenGL を使用した．OpenGL を採用した理由として，OpenCL と低レイヤの部分で連携ができるため，他の GUI フレームワークを使用する場合と比べて高速に描画処理を行えることが挙げられる．MasCL は開発者から OpenCL と GLUT(OpenGL Utility Toolkit) の API を隠蔽し，代わりに開発者に独自の機能を提供する．

4.3 OpenGL

OpenGL(Open Graphics Library) とは SGI を中心に開発されたグラフィックス処理のためのアプリケーションプログラミングインターフェイス (API) である．OpenCL と同様に Khronos Group が標準を策定している．ハードウェアや OS に依存しないように設計されており，現在でも Windows や Mac OS X，Linux だけでなく組み込み向け機器でも幅広く利用されている．

OpenGL はハードウェアに近い低レイヤのライブラリであるため，よりソフトウェアに近い GLUT のようなライブラリが複数存在する．GLUT とは，OpenGL を C 言語から利用できるライブラリである．OpenGL のライブラリでは最も広く普及しており，開発が中止された現在でも多くの開発者に利用されている．

```
#include "MasCL.h"

/* Inherit "mcl::MasCL" */
class Sample : public mcl::MasCL {
  Sample(int num_agents,
         int space_width,
         int space_height) {
    /* Definition of constructor and initialization */
  }
  void run() {
   /* Define one step of simulation */
  }
};

int main() {
  int width = 2000, height = 1000;
  int population = 500000;
  Sample sample(width, height, population);
  sample.start();
  return 0;
}
```

Fig. 3: Example class which inherits `mcl::MasCL`

4.4 MasCL の機能

MasCL ライブラリは C++言語から使用することができるライブラリである．`MasCL.h` ファイルをインクルードすることで，すべての機能を使用することができる．`MasCL.h` には `mcl::MasCL` クラスが定義されており，開発者はこのクラスを使用してシミュレーションの開発を行う．また MasCL は MasCL は `mcl::MasCL` クラス以外にも様々な機能を提供する．

4.4.1 シミュレーションの定義

MAS は，エージェントと空間の間のインタラクションを定義し，それをループ処理のように繰り返すことでシミュレーションが進行する．ここではそのループ処理 1 回のことを 1 ステップと定義する．

`mcl::MasCL` クラスにはオーバイライドできる `run()` が定義されており，開発者は `run()` を再定義することで，シミュレーションの 1 ステップの処理を記述する．`start()` を実行するとシミュレーションが開始し，`run()` で定義された内容が繰り返し実行される．Figure.3 は，`mcl::MasCL` クラスを継承して使用する例である．

4.5 ウィンドウの自動生成

シミュレーションを開始すると同時に，GLUT を使用したウィンドウが自動で生成され，そのウィンドウにシミュレーション結果が表示される．そのウィンドウには以下の機能があらかじめ登録されている．ユーザはこれら機能を用いることで，大規模なシミュレーションの現象を細かく観察することが可能となる．

- キーボード操作
  - シミュレーションの再生

- シミュレーションの一時停止
- シミュレーションの 1 ステップ再生
- シミュレーションの終了

- マウス操作
  - シミュレーションの拡大・縮小
  - シミュレーション表示範囲の移動

4.6 エージェント・空間の描画

MasCL はエージェントと空間の描画をシミュレーション開始後，自動的に行う．また，高速に描画を行う仕組みを使用しており，開発者は描画に要する処理時間を意識することなく開発を行うことができる．

MasCL はエージェントと空間を描画する手法として OpenGL を採用している．その理由として，OpenCL が OpenGL のバッファを読み書きが可能な点が挙げられる．

例えば，OpenGL が作成したウィンドウ上に複数のエージェントを描画するため，描画に必要な座標・色データ (以下，GL バッファ) が VRAM 上に記録されているとする．1 ステップ終了後，エージェントの座標が変化した場合，その座標データを使用してウィンドウ上の座標を計算し，新しい座標にエージェントを描画しなければならない．大規模な MAS の場合，数十万体におよぶエージェントの座標データが存在するため，その処理に膨大な時間を要してしまう．

通常では GL バッファの更新を逐次的に処理しなければならないが，OpenCL は GL バッファに直接書き込むことが可能なため，並列にウィンドウ上の座標を計算し，GL バッファを書き換えることが可能である．OpenGL はその更新された GL バッファを使用して描画を行う．そのため，毎ステップで高速にエージェントと空間の描画が実現を行う．

4.7 エージェントの設定

MAS を行う場合，エージェントの種類を形で区別して観察を行うと，エージェントの視認が容易になる．MasCL では，以下の関数でエージェントの設定を行うことができる．

- `void addAgent(int num_agents, int shape, float size)`
  エージェントの数，ウィンドウ上の形と大きさを指定することができる．
  - `num_agents`：エージェントの数
  - `shape`:`Normal`(デフォルト)，`Square`，`Arrow` のいずれか
  - `size`：エージェントの大きさ (デフォルト：1)

`addAgent` を用いて描画することができるエージェントのサンプルを Fig.4 に示す．左から順に `Normal`，`Square`，`Arrow` で定義されている．

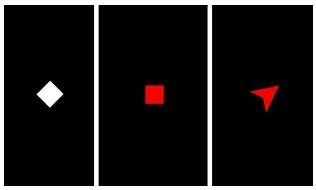

Fig. 4: Kinds of agents in MasCL

4.7.1 OpenCL バッファの自動生成

MasCL は OpenCL を使用して並列コンピューティングを行う．メインメモリ上にあるデータを OpenCL デバイスに計算させるためには，メモリオブジェクトという OpenCL デバイス上に生成されるバッファを用意し，メインメモリからコピーしなければならない．MasCL では，エージェントや空間のパラメータのメモリオブジェクト生成の処理を簡潔に行えるよう特殊なクラスを提供している．以下に記載するクラスは，C++のテンプレート機能を使用して型を指定することが可能である．ただし，OpenCL C 言語が C++をサポートをしていないため，`T` には `int`，`float` のみ指定可能である．これらのクラスは定義した際に，エージェント数や空間のサイズを指定することで，その数と同数のデータを生成する．

- `class SpaceData<T>`：空間のパラメータを扱うクラス
- `class AgentData<T>`：エージェントのパラメータを扱うクラス
- `class AgentPosition<T>`：エージェントの座標を扱うクラス

これらのクラスでパラメータを定義することで，以下の `createBuffer` 関数を用いてメモリオブジェクトを作成することができる．

- `void createBuffer<T>(SpaceData<T>&)`
- `void createBuffer<T>(AgentData<T>&)`
- `void createBuffer<T>(AgentPosition<T>&)`

5 MasCL を用いた実験

MasCL を用いて記述したエージェントモデルについて，どの程度工数を削減できたか調べるため，コード数の比較を行う．MasCL と比較を行う対象として，同じモデルを C++から OpenCL と GLUT を用いて実装し，同等の機能を持ったオリジナルプログラムを用いる．記述実験は 5 つのモデルについて行ったが，本報告では 2 つのモデルについて述べる [1]．

5.1 格子状の空間をエージェントが移動するモデル

エージェントモデルで最も種類の多いモデルである．格子状の空間があり，エージェントはその空間を移動する．空間 1 マスに存在できるエージェントは 1 体のみであり，1 マスにエージェントが複数存在することはない．ここでは分居モデルを記述する．

[1] 本稿で述べた以外の 3 つのモデルについては，発表で報告する．

分居モデルとは，米国の経済学者 Thomas C. Schelling が提案したモデルである [14]．白人と黒人が，個々人の人種差別意識がさほど強くなくても，白人が多く住む地域と黒人が多く住む地域に分かれてしまう現象を MAS 的に説明した [2]．MAS の分野では非常に著名なモデルのひとつである．Fig.5 は分居モデルのシミュレーションの様子を表したものである．赤と緑の 2 種類のエージェントが存在し，エージェントが存在しないセルは黒で表示されている．シミュレーション開始時にエージェントたちはランダムに配置されているが，シミュレーションが進行していくほど，互いに同じ種類のエージェントでクラスタを形成するようになる．このモデルの規模は，空間サイズが 500×500，エージェントの数が 7,500 である．

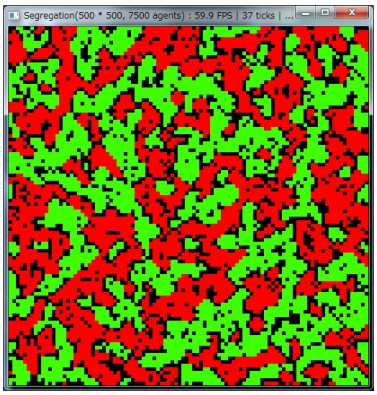

Fig. 5: Segregation Model

5.2 格子状に並べられたセルをエージェントとして扱うモデル

動かないエージェントたちを格子状に並べ，1 マスを 1 体のエージェントとして扱うエージェントモデルである．MAS の中ではセル・オートマトンに近い分類である．そのモデルのサンプルとして，森林火災モデルがある [17]．森林火災モデルとは，森林で発生する火災がどのように広がっていくのかをシミュレーションしたものである．図 6 は森林火災モデルの様子を表したものである．

Fig.6 は空間サイズが 500×500 のシミュレーション結果である．

5.3 コード数の比較

Table.2 は先に述べた 2 つの記述実験のコード数の比較である．この表からわかるように，MasCL を利用することによって，大幅にコード数を削減することができた．

Table 2: Compare of the LOC of MasCL and Scratch code

Model	Scratch	MasCL
Segregation	768	96
ForestFire	770	125

[2]Shelling はこの実験を，チェス盤とコインを用いておこなったので，MAS 的な意識は無かったと思われるが，実質的には完全に MAS である．

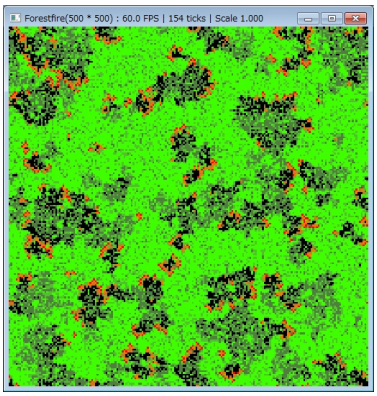

Fig. 6: ForestFire Simulation

5.4 実行速度

MasCL を使用して実装したプログラムが，OpenCL を直接使用して実装したプログラムに比べて，速度向上が行えたかどうか調べるために実験を行い，互いの実行速度を比較した．また，CPU の単一プロセッサでの実行速度とも比較を行い，並列計算を利用することでどの程度の速度向上が得られたか調べた．全プログラムとも C++言語を使用し，GUI フレームワークには GLUT を使用して実装を行った．実験の対象となるエージェントモデルにはシェリングの分居モデルを採用した．Table.3 は実験に使用したコンピュータの構成である．

Table 3: Experimental environment

OS	Windows 7 Professional 64bit
Main Memory	16GB
CPU	Intel Core i7 - 3960X
GPU	Radeon HD 7970

Fig.7 は，MasCL，OpenCL，C++のプログラムの実行速度を FPS(Frames Per Second) で表し，比較したものである．なお，C++のプログラムでは，GPU の機能（並列処理）は用いず，すべて逐次処理を行った．このとき，空間のサイズを 1000×1000 に設定し，エージェントの数を人口密度で算出し，密度を 70%，75%，80%に変化させて実行した結果である．Fig.8 は，MasCL プログラムと OpenCL プログラムを，空間のサイズを 2000×2000 に拡大し実行速度を比較したものである．結果から，OpenCL で実装したプログラムが最も高速にシミュレーションを行えていることが確認できた．しかし，MasCL で実装したプログラムは，C++で実装した CPU 単一プロセッサのみでの実行と比較して，10 倍以上の高速化が行えたことが確認できた．なお，空間のサイズが 2000×2000 の場合は，CPU だけの処理では，実用性を持たない速度しか得られなかったので，グラフではデータを省略している．

6 考察

6.1 ソースコードの行数の削減について

5.1 節と 5.2 節の実験結果から，MasCL を使用することで大幅にソースコードの行数を削減することができた．その理由として，OpenCL の冗長な初期化工程と GUI を利用してエージェントや空間の描画処理をさ

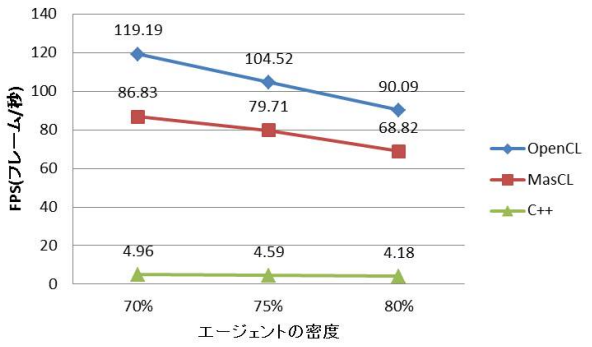

Fig. 7: Execution speed for Space size 1000×1000

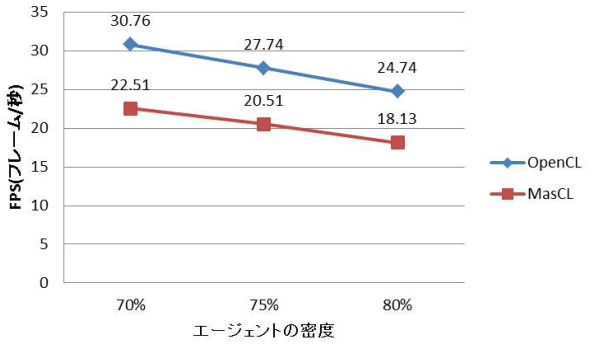

Fig. 8: Execution speed for space size 2000×2000

せる複雑さに原因がある．

OpenCL はプラットフォームに依存しない並列計算のフレームワークを目的として策定されたが，開発者がハードウェアを抽象的に，かつ低レイヤの部分まで扱えるようにされている．そのため，並列計算を行うために様々なリソースを準備しなければならず，初期化処理でソースコードの行数を占めてしまう．MasCL ではその初期化処理の部分を隠蔽しており，開発者は OpenCL のリソースの確保や解放を意識することなく開発が行える．そのため，MasCL を使用して実装したプログラムの方が，ソースコードの行数が数百行単位で大きく削減することができた．

加えて，MasCL を使用したプログラムと比較対象のプログラムは GUI フレームワークとして GLUT を使用してウィンドウを作成し，OpenGL を呼び出してエージェントと空間を描画している．GUI アプリケーションを開発する場合，ウィンドウをマウスでクリックした場合やキーボードが押された場合に，ウィンドウにどのような処理を行わせるかを定義しなければならない．GLUT でウィンドウを作成した場合は，ウィンドウが再描画された場合やアイドル状態になった場合のコールバック関数を定義，登録する工程が必要となる．MasCL を利用してエージェントモデルを構築する場合，シミュレーションを観察するために必要なシミュレーションの再生，一時停止や表示範囲の拡大・縮小などの機能があらかじめ登録されている．比較対象のプログラムはそれらを定義，登録するための工程が必要だったため，MasCL を使用したプログラムが大幅にソースコードの行数の削減を行えた．

6.2 実行速度

5.4 節の結果から，MasCL を使用したプログラムより OpenCL を直接使用して実装されたプログラムのほうが高速にシミュレーションを行えたことがわかる．MasCL は GPGPU を利用するために OpenCL を利用しているが，MasCL から使用できる API が OpenCL の複雑な並列計算のコードを隠蔽しているがゆえに，その中は一般化されており，最適化されたものではないからである．また，OpenCL で直接実装したプログラムは，カーネルの呼び出しを最小限にしている．これは，カーネルの実行のたびに発生する命令発行のレイテンシを減らすためである．しかし，MasCL の並列計算を使用する API は一般化されているため，その機能によって細かく分類されている．そのため，同じエージェントモデルを MasCL で実装する場合は複数の API を使用することで同等のアルゴリズムを実現する．つまり，OpenCL で直接実装したプログラムよりカーネルの呼び出し回数が増えてしまい，その分レイテンシが発生してしまう．結果，パフォーマンスに影響を及ぼし速度が低下してしまったと考えられる．

6.3 簡単な応用–拡張分居モデル–

MasCL を用いて，Shelling の分居モデルを拡張したモデルをシミュレートした．オリジナルの分居モデルでは，エージェント（住人）は 2 種類であるが，拡張モデルでは，これをさらに増加させた．

よく知られているように，分居モデルにおけるエージェントは，各々の属性として幸福値を持っており，このパラメータは隣接した周囲 8 マスにいるエージェントの種類によって決まる．幸福値 H は下の式で求められる．この値がユーザの設定する閾値より低い場合，エージェントは空いているスペースへランダムに移動を行い，高い場合はその場に留まる．

$$H = \frac{\text{周囲 8 マスにいる同じグループのエージェントの数}}{\text{周囲 8 マスにいるエージェントの数}}$$

今回の拡張モデルでは，グループの数と幸福度を変更してシミュレーションを行った．その結果，以下のような現象が観察された．

このシミュレーションの空間サイズは 1,000×1,000 で，エージェントの数は 750,000 である．そしてエージェントが 5 グループ (それぞれ同数のエージェント) で，シミュレーションを行った．Fig.9 と Fig.10 は，閾値が 0.55 と 0.5 の場合の分居現象である．幸福度 H の閾値が 0.55 の場合，Fig.9 に示すように，広域に同グループのエージェントが集まり，大きなクラスタが形成されることを確認した．通常は Fig.10 のような小さなクラスタを形成するが，$H = 0.55$ の場合（Fig.9）は通常では見られない大きなクラスタを確認できた．

現象は以下の段階を踏んで発生する．

1-400 ステップ： エージェント 20 体ほどの分居が至る所で発生するが，すぐにエージェントたちは移動し，その集まりは消える．

400-800 ステップ： エージェントたちが移動せずに集団を維持し続けるものが 2，3 確認できるようになる．

600-800 ステップ： 前段階で発生した数十体の集団が，200 体程度の集団にまで拡大する．また，新たな集団が 2，3 発生しはじめる．

800-1500 ステップ： 集団がさらに拡大をはじめる．

1500-2500 ステップ： 10000 体におよぶ大規模な分居が確認できるようになる．目視で大小に及ぶ集団を 30 程度発見できるようになる．

2500-3000 ステップ： 移動するエージェントが少なくなり，やがてすべてのエージェントが移動しなくなる．

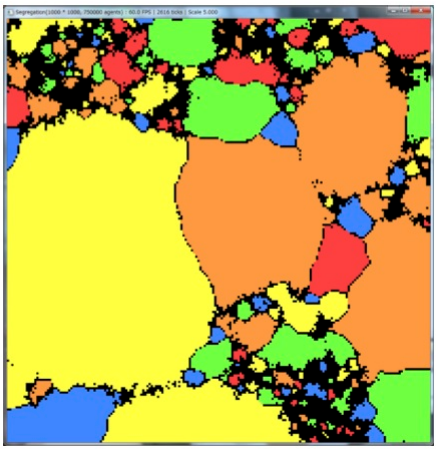

Fig. 9: Segregation with threshold value of $H = 0.55$

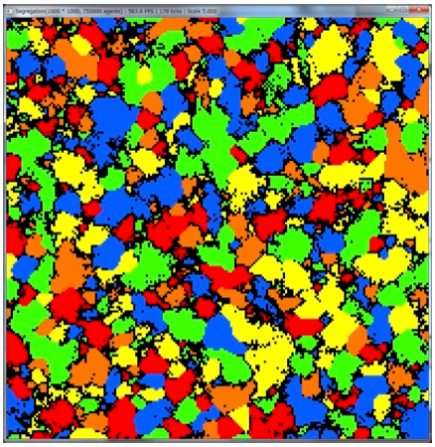

Fig. 10: Segregation with threshold value of $H = 0.5$

このことが何を意味するのかについては，現在のところ不明である．現実の問題として何か意味を持つ結果なのか，それとも今回のシミュレーションで用いたモデルに起因するものなのかについては，今後詳細な検討を加えてゆく予定である．

7 結言

本稿では，GPU を利用したマルチエージェント・シミュレーション用のライブラリである MasCL を開発し，そのライブラリを使用して 5 つのエージェントモデルを実装した．作成したモデルは，ライブラリを使用せずに実装した場合に比べて，大きく行数を削減することができた．

MasCL は並列計算のフレームワークに OpenCL を採用している．しかし，現状の MasCL は開発者から完全に OpenCL C 言語のプログラミング，カーネルを実装する部分は隠蔽することができていない．そのため，開発者は少なからず並列計算についての知識は必要となり，プログラミングを専門としないユーザに利用してもらうためには，まだライブラリとしての敷居が高いと考えられる．今後はこの敷居を下げるために，開発者が並列計算の知識を必要とせずに開発を行えるような仕組みを提供しなければならない．

また，5 節で実装したモデルは，人工社会の基礎的なモデルであるため，実際の社会現象をシミュレーションしたものではない．今後は，GPGPU を利用した実社会スケールでの MAS が妥当であるかの検証も踏まえて，より厳密なモデルの実装を行いたいと考えている．現在計画しているものの一つとして，選挙制度のシミュレーションがある．

選挙制度のシミュレーションは，現在の日本の選挙制度（小選挙区比例代表並立制）が本当に民意を反映しているのかどうか，という疑問から出発している．例えば，2012 年末に行われた衆議院総選挙では，現在の与党の主力である自由民主党の，対投票者の得票率は，おおむね小選挙区で 42%，比例区では 27%であった．しかし獲得議席数は小選挙区で約 80%，比例区で 32%であった．一方民主党（総選挙実施時の主力与党）の対投票者の得票率は約 23%であり，自民党の半分以上の票を獲得しているにもかかわらず．議席数は約 10%の 27 議席となった．比例区はその特性上，ほぼ得票率を反映した獲得議席数になっている [19]．従って，現在の政権を支える衆議院の議席配分は，必ずしも民意を反映しているものであるかどうかは疑問が残る．これについては，選挙直後から様々なメディアで，今回の選挙の結果の解釈が発表されている [20, 21] が，選挙制度の是非，という立場からの意見は少ない．選挙制度の是非は簡単に論じられるものでは無いが，その大きな評価基準として，民意の反映があることは間違いない．民意と乖離した選挙結果は，選挙制度の不備を示す一つの証左の一つとなる．

2012 年の総選挙の結果が民意と乖離している，ということについては，一概に断ずることはできないが，素朴な感情として，投票者の 40%，全有権者の比率で言うと約 25%の支持しかない政党が，議席の 80%を占めるというのは，不自然と感じる．したがって，何らかの形でより民意を反映しやすい選挙制度を立案し，シミュレートしてゆくことが必要である．そのためのツールとして，MAS は大きな武器となると考える．例えば 6.3 節に述べた，巨大なクラスターの出現は，あるいはこの選挙結果を説明するモデルなのかもしれない．投票率が約 59%，主要政党を「自民党」「民主党」「日本維新の会」「公明党」「次世代の党」[3] の 5 党と考えると，6.3 節のモデルの数値と，妙に符合する．むろんこれはおそらく偶然の一致であり，このことから単純な結論は導けないが，このことからも，MAS が一つの強力なツールとなり得ることがうかがえる．選挙制度をシミュレートするためには，小規模なシミュレーションではなく，エージェントの数やエージェントが動く空間のサイズを大きくする必要がある．そのための強力なツールとして，MasCL を活用していきたい．

[3] これは 2014 年 7 月現在，衆議院で 2 桁以上の議席を持っている党を議席数上位から 5 党選らんだものであり，それ以上の意味は無い．

参考文献

1) 蟻川浩，村田忠彦：“環境情報を考慮した大規模マルチエージェントシミュレーションの並列計算機上での実現”，知能と情報，Vol.22，No.2，211/222 (2010)

2) J.M.Epstein，R.L.Axtell，*Growing Artificial Societies –Social Science from the Bottom Up–*，MIT Press，(1996)

3) Wen-mei WHwu，*GPU Computing Gems - Emerald Edition*，Morgan Kaufmann (2011)

4) 床井浩平，”GLUT による OpenGL 入門 – OpenGL Utility Toolkit で簡単 3D プログラミング”，工学社 (2005)

5) Mikola Lysenko，Roshan D'Souza，Keyvan Rahmani：A Framework for Megascale Agent Based Model Simulations on the Graphics Processing Units，*Journal of Artificial Societies and Social Simulation*，vol.11，no.4 (2008) http://jasss.soc.surrey.ac.uk/11/4/10.html

6) 谷本淳，藤井晴行：“マルチ・エージェント・シミュレーションによる情報伝播特性に関する一考察”，http://ktlabo.cm.kyushu-u.ac.jp/j/theme/complex/tanimoto20010823.pdf.

7) 根岸祥人，加賀屋誠一，内田賢悦，萩原亨：“マルチエージェントシミュレーションを用いた震災時避難の交通行動に関する研究”，第 28 回土木計画学研究発表会・講演集 Vol: 28 (2003)

8) 横澤和也：“居住地の凝集化と社会的サービス施設の拠点化の空間シミュレーション”，第 12 回 MAS コンペティション”, 部門 1，No.6 (2012)

9) 森下仙一，蟻川浩，村田忠彦：“MPI と GridRPC の併用によるマルチエージェントシミュレーションプログラムの実装”，情報処理学会研究報告，2007-HPC-111(24) (2007)

10) Nvidia Corpolation，“CUDA C PROGRAMMING GUIDE”，http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (2014)

11) 株式会社フィックスターズ，“改訂新版　OpenCL 入門　1.2 対応マルチコア CPU・GPU のための並列プログラミング”，インプレスジャパン (2012)

12) Khronos OpenCL Working Group，“The OpenCL Specification Version 1.1”，hhttp://www.khronos.org/registry/cl/specs/opencl-1.1.pdf (2011)

13) Kalyan S.Perumalla，Brandon G. Aaby：“Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs”，SpringSim '08 *Proceedings of the 2008 Spring simulation multiconference*，116/123 (2008)

14) Thomas C. Schelling, Dynamic models of Segregation, *Journal of Mathematical Sociology*, vol.1, 143/186 (1971)

15) Paul Richmond，Dr Simon Coakley，Dr Daniela Romano，“A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA”，*Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*，10/15 (2009)

16) NVIDIA Developer Zone，https://developer.nvidia.com/category/zone/cuda-zone

17) 伊庭斉志，“複雑系のシミュレーション–Swarm によるマルチエージェント・システム-”，コロナ社 (2007)

18) NetLogo Home，http://ccl.northwestern.edu/netlogo/

19) 平成 24 年 12 月 16 日執行衆議院議員総選挙・最高裁判所裁判官国民審査速報結果，http://www.soumu.go.jp/senkyo/senkyo_s/data/shugiin46/

20) http://www.nikkei.com/article/DGXDASFS17001_X11C12A2EB1000/

21) http://www.yomiuri.co.jp/adv/chuo/opinion/20121225.html