自己移動型スレッドに基づく並列マルチエージェントシミュレータM++の構築

鈴木有也[†] 福田宗弘[††] 和田耕一[†††]

マルチエージェントに基づく複雑系モデルは，多種多様なエージェントの相互作用によってシミュレートされるため，シミュレーションアルゴリズムの良好な記述性がつねに求められる．そこで本研究では，スレッドで実装されたエージェントがクラスタ上を移動し，相互に作用しながらシミュレーションを進める並列マルチエージェントシミュレータM++を提案する．M++は，M++言語によって記述性を追求し，アルゴリズムに沿ったシミュレーションプログラムのコーディングを容易にする．また独自のSスレッドライブラリ，およびゼロコピー通信を用いたスレッド移動によって性能の低下を軽減し，台数効果を得ることができる．本論文は，人工社会シミュレーションをM++とMPIの双方で記述することによって，M++の記述性を検証し，その性能について考察することを目的とする．

M++: Parallel Multi-Agent Simulator Based on Self-Migrating Threads

Naoya Suzuki,[†] Munehiro Fukuda[††] and Koichi Wada[†††]

The multi-agent approach requires a large number of various agents and objects, where programmability are of essence in such simulations. To improve this feature of the approach, we propose the M++ parallel multi-agent simulator, in which threads represent agents, autonomously navigate through, and get access to a simulation space constructed over a cluster system. We have not only realized a narrow semantics gap between simulation models and their code with our *M++ language*, but also retained performance using our original *sthread* library and zero-copy thread migration scheme. In this paper we demonstrate programmability of M++ using artificial societies simulation by comparing with MPI and inspect its performance.

1. はじめに

近年，複雑系をモデリングする1つの手法としてマルチエージェントシミュレーションが注目されている[6]．マルチエージェントシミュレーションでは，自らの動作を自らが決定する実体（エージェント）が，規定された動作ルールに従って，エージェントの動作空間（シミュレーション空間）を構成する要素を参照・更新しながら他のエージェントと相互に作用する．これら微視的なエージェント間の相互作用は，シミュレーションの系全体に影響を与えて，エージェント集団に様々な群行動，いわゆる複雑系の自己組織化をもたらす．そのため，マルチエージェントシミュレーションは，複雑系における多様な自己組織化を予測・解明する手法として多岐にわたる研究分野に応用されている．代表例として，R.J. Collinsらによる Ant-farm シミュレーションを用いた生態系の解明[18]，ならびにJ.M. Epsteinらによる人工社会シミュレーションを用いた社会科学への応用[15]，ニューラルネットワークと遺伝的アルゴリズムを用いた工学のボトムアップ的な設計手法への応用[24]，ビジネスにおける航空会社の最適な貨物運搬経路の求解[1]等があげられる．これらのシミュレーションを処理するためには，自己組織化を観察できるだけの十分なエージェント数とそれらを収容するのに十分な広さを持ったシミュレーション空間を必要とする．したがって，第1に良好な**シミュレーション性能**，第2に多種多様なエージェントの動作を簡潔に定義するための**記述性**が必要不可欠といえる．

これら2つの課題のうち，前者のシミュレーション

† 筑波大学大学院工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† Computing and Software Systems, University of Washington, Bothell
††† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

性能を向上させる方法として，シミュレーションの並列化があげられる．近年では，コストパフォーマンスとシステムの拡張性に優れたPCクラスタによる並列計算プラットフォームが精力的に構築されている．PCクラスタ上でマルチエージェントシミュレーションを並列化する方式としては，(1) MPI等における，メッセージパッシング方式によるセルを主体としたシミュレーション，および (2) StarLogo[4]，AweSime[11]等における，共有メモリ方式によるエージェントを主体としたシミュレーションの2つがあげられる[14]．しかしながら，これらの手法では以下の特徴により記述性の向上に支障が存在する．

前者の (1) セルを主体としたシミュレーションでは，多次元配列で表現されたシミュレーション空間とその空間に属するエージェントを分割して個々のプロセッサに割り当て，各プロセッサが，それぞれのシミュレーションサイクルで，隣接するプロセッサと通信を行って，配列要素およびエージェントを送受信する．この方式は，並列処理に必要な通信・同期等，シミュレーションアルゴリズム以外の処理コードが必要となり，記述性を著しく低下させる．

また，後者の (2) エージェントを主体としたシミュレーションでは，エージェント集団を分割して個々のプロセッサに固定的に割り当て，共有メモリシステム上でシミュレーション空間全体を共有することによって，シミュレーション空間の更新が他のプロセッサに反映される．この方式は，PCクラスタ上でシミュレーション空間を共有するために，分散共有メモリを構築するか，HLA[21]・DIS[10]のように，シミュレーション空間の更新情報を逐一マルチキャストする下位システムを記述しなければならない．

そこで本研究では，記述性の向上を追求するために，上述の2手法とは異なる新たな手法を用いた**並列マルチエージェントシミュレータ M++**を提案する．M++の最大の特長は，エージェントの視点から並列マルチエージェントシミュレーションを記述できることである．これを実現するために，本研究では，C++言語をベースとしたエージェント指向言語 **M++言語**を設計した．M++言語では，エージェントの視点からシミュレーション空間を構築してクラスタ上に分散することができる．また，M++言語の簡潔なプリミティブによって，クラスタ上に分散されたシミュレーション空間をエージェントが移動し相互に作用する動作を的確に記述することができる．以降，本論文では，M++を提案シミュレータとして，M++言語を，M++上で使用するシミュレーション記述言語として定義する．

さらに本研究では，記述性の向上にともなうシミュレーション性能の低下を軽減し台数効果を得るために，独自に開発したユーザレベルスレッドであるSスレッドを用いてエージェントをスレッド化し，それにプロセッサ間の移動機能を付加した．本論文ではこれを**自己移動型スレッド**と呼ぶ．自己移動型スレッドは，これから参照しようとするシミュレーション空間を持つプロセッサに移動して，シミュレーション空間の参照，および他の自己移動型スレッドとの通信を同一プロセッサ上で行う．本研究では，空間的局所性を利用したこの方式によって性能低下の軽減を目指す．これらに加えて，物理層がMyrinet[22]の場合は，ゼロコピー通信によるスレッド移動を用いていっそうの性能向上を図る．

本論文では，J.M. Epsteinらによる人工社会の公害拡散シミュレーション[15]をMPIとM++の双方で記述し，それらを比較することによって，M++の記述性を評価するとともに，性能について考察することを目的とする．以降，2章でM++の設計方針，ならびにシステム構成，M++言語仕様，M++の実装技術について述べる．3章で，人工社会シミュレーションを用いてM++言語とMPIの記述性を比較・評価する．4章でM++の実際のシミュレーション性能をMPIと比較・評価する．5章で関連研究について述べ，6章で本論文をまとめる．

2. システム設計

2.1 基本構成

M++は，**図1**に示すように3層のネットワーク階層から構成される．最下層に位置するのが**物理ネットワーク層**であり，クラスタ上でプロセッサ間を接続す

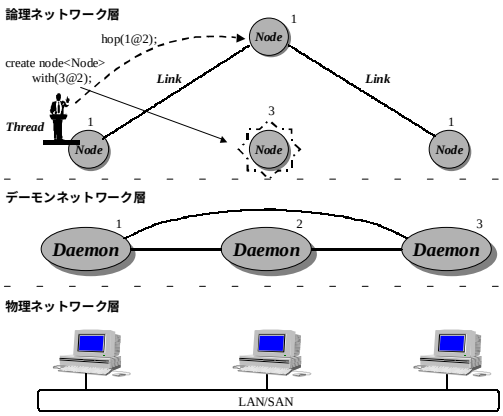

図1 M++ネットワーク階層
Fig. 1 M++ network layers.

る物理層，およびその上で動作する TCP/IP とゼロコピー通信プロトコルが該当する．その上位には**デーモンネットワーク層**が形成される．デーモンは，クラスタ上の各プロセッサに 1 個ずつ存在し，M++を分散管理する．それらは整数型の ID で識別され，互いに他のデーモンと接続を行って，完全結合のデーモンネットワークを形成する．各デーモンに対応する ID は，参加するプロセッサの IP ネームと割り当てたい ID の組をプロファイルに記述することによって任意に決定できる．最上位に位置するのは**論理ネットワーク層**であり，整数型の ID で識別されるノードとリンクによって自己移動型スレッドの移動領域が各プロセッサに分散配置される．本論文では，これを M++におけるシミュレーション空間と定義する．シミュレーション空間は異なるプロセッサをまたいで構築することができる．そして，M++におけるシミュレーションは，シミュレーション空間上において 1 個以上の自己移動型スレッドが規定された動作を行うことを指す．以下，自己移動型スレッド・ノード・リンクの各要素について特徴を述べた後，それらを管理するデーモンと M++におけるシミュレーション過程について説明する．

（1）自己移動型スレッド

自己移動型スレッド（以下，スレッド）は，シミュレーション空間を移動し，後述するノードとリンクを参照・更新しながら他のスレッドと相互に作用する．スレッドは，それぞれ独立した実行の流れと内部変数を持ち，ネイティブ実行において強いマイグレーション[5]が可能である．すなわち，関数・メソッドの途中で他のプロセッサに移動でき，移動後にその直後から実行を再開できる．移動方法には，移動先のノードを直接指定する絶対移動と，リンクに沿って移動する相対移動の 2 種類が提供される．移動したスレッドは，移動先のノード，またはそのノードに接続されたリンクの外部インタフェースにアクセスして，それらの内部状態を参照・更新できる．同一ノードに存在するスレッドに対しても同様，外部インタフェースを通して内部状態の参照・更新を行うことができる．

スレッドは，その実行時にプロセッサをまたいでノードとリンクを生成・削除してシミュレーション空間を構築することができる．その他，自らを複製する機能，ならびに新たなスレッドを投入する機能，スレッド間の同期機能を有する．

（2）ノードとリンク

ノードはスレッドの作業領域として定義され，リンクはスレッドの移動経路としてその両端にノードを接合し，ノード間を連結する．ノードとリンクを総称して本研究では**オブジェクト**と呼ぶ．本研究のオブジェクトは，オブジェクト指向で定義されるオブジェクトと同等の概念を持ち，さらに以下の特徴を兼ね備える．

第 1 に，オブジェクトはすべてモニタ機構を持ち，情報の一貫性が保たれる．よって，スレッドがしかるべきノードに移動し実行を再開すると，このスレッドが実行権を譲るか別のノードに移動するまで他のスレッドは待機させられる．リンクに関しても同様である．モニタ制御は M++デーモンが行うため，ユーザがそれを意識することはない．このことは，ユーザのモニタ誤操作によるデッドロックを防止する．

第 2 に，ノードはリンクを介してデータを交換することができる．これは，同一リンクの両端に接続されたノードが，そのリンクへデータの読み書き操作を行うことによって実現される．M++では，ノード内に `clock` メソッドを定義すると，スレッドは `clocking` というキーワードを用いて，M++に存在するすべてのノードの `clock` メソッドをいっせいに起動することができる．この機構を**クロッキング機構**と呼ぶ．`clock` メソッドにノード間のデータ交換を記述しておくことによって，全ノードがリンクを介していっせいに通信を行い，セルオートマトンの動作を実現することができる．

（3）デーモン

デーモンは，プロセッサ上に存在するスレッドを管理し，それらの要求を実行する．たとえばノード生成の要求があれば，ノードの生成先のデーモンに，生成すべきノード情報が格納されたデータパケットを送信する．スレッドの移動・複製・新規スレッドの生成の要求も同様にデーモンが引き受け，必要なスレッドパケットの送受信を行う．その他，シミュレーション空間の管理，オブジェクトのモニタ制御，プロセッサ間をまたぐリンクの制御，シミュレーション時間の制御等，シミュレーションに必要不可欠な処理を担う．これらのサポートによって，ユーザはシミュレーションアルゴリズムの記述のみに専念できる．

（4）シミュレーション過程

スレッドは，ユーザプロンプトから，デフォルトで M++が生成した INIT ノードに投入される．最初に，投入されたスレッドは，プロセッサ間を移動しながらノードとリンクの生成を繰り返し，初期状態のシミュレーション空間を分散配置する．このシミュレーション空間は，シミュレーション実行時にスレッドがノードとリンクの生成・削除を行うことによって，トポロジを任意に変化させることができる．その後，シミュレーションに必要な個数のスレッドが投入され，それ

```
class Node { public: ... };
class Link { public: ... };
thread Thread {
public:
  Thread( int argc, const char** argv ) :
    nid( 0 ), did( 1 ) {}
  void main() {
    create node<Node> with( nid@did );
    hop( nid@did );
  }
private:
  int nid, did;
};
```

図 2 M++ フレームワーク
Fig. 2 M++ framework.

ぞれがグローバルバーチャルタイム（以下，GVT）[8] にしたがって動作を進める．各スレッドは，ノードを移動しながらノードの参照・更新を行い，他のスレッドと相互に作用しつつ必要な計算を行う．そうすることによってシミュレーション全体が進行していく．

2.2 M++ 言語

M++言語は，クラスタ上におけるマルチエージェントシミュレーションを記述する言語として以下の 6 つの特徴を持つ．M++言語で記述されたプログラムは M++トランスレータによって C++言語に変換され，C++コンパイラによって完全にバイナリ化される．以下，本節で述べる言語構文に関しては，M++マニュアル[9] に記載されてあるもののうち，3 章で使用しているものに限定している．

（1）エージェント指向パラダイムに基づく言語体系

M++言語では，スレッドをエージェント指向パラダイムに基づいて記述することができる．すなわち，**図 2** に示すように，スレッドは，`thread` キーワードを用いて，スレッドの内部変数とメソッドとともに，`main` 関数に記述されたスレッドの実行の流れをカプセル化することができる．スレッドには，クラスと同様に継承・多態の概念を適用でき，`public` 等のアクセス指定子を使用して，他のスレッドからのアクセスを制限することができる．

スレッドは以下の手順で実行される．最初に，

ThreadName`( int` *argc*`, const char**` *argv* `)`

という型のコンストラクタが，UNIX シェル，または他のスレッドからのスレッド生成時に引数を受け取ってスレッド変数の初期化を行う．その後，`main` 関数へと実行の流れが移り，そこに記述されているシミュレーション動作を行う．`main` 関数の動作がすべて終了するとスレッドは消滅する．

（2）シミュレーション空間の生成・分散配置

M++言語は，シミュレーション空間の要素であるノードとリンクの生成・分散配置を `create` キーワードで実現する．

- `create node<`*ClassName*`>(` *args...* `)`
  `with(` *NodeID*[`@`*DaemonID*] `)`
- `create link<`*ClassName*`>(` *args...* `)`
  `with(` *SrcLinkID* `)`
  `to(` *NodeID*[`@`*DaemonID*] `)`
  `with(` *DestLinkID* `)`

図 2 に示すように，ノードとリンクは，C++言語のクラス定義と同様，`class` キーワードを用いて定義される．定義したクラスはノードとしてもリンクとしても使用できる．ノードを生成するには，`create node<`*ClassName*`>`によってクラス名を指定し，`with` 以下でデーモン ID を指定することによって，M++に参加しているどのプロセッサに対しても *ClassName* 型のノードを動的に生成することができる．同様に，`create link<`*ClassName*`>`キーワードによって，M++に参加しているどのプロセッサのノードに対しても，当該スレッドが存在する現在のノード（以下，現ノード）と *ClassName* 型のリンクで接続できる．

（3）スレッドの振舞いの記述

M++言語では，スレッドの移動に `hop` キーワードを，スレッドの複製に `fork` キーワードを用いる．これらには振舞いの目標点となるノード（以下，目標ノード）を指定する方法として絶対指定と相対指定の 2 種類が使用でき，これらを必要に応じて使い分けることによってスレッドの動作を柔軟に記述できる．

絶対指定は以下のような記述法で実現する．

- `hop(` [*NodeID* [`@`*DaemonID*]] `)`
- `fork(` [*NodeID* [`@`*DaemonID*]] `)`

これらは，`@`演算子を用いて，デーモン ID とノード ID の組で目標ノードの絶対位置を指定する．この記述の利点は，ノードがどのプロセッサにどの ID で割り当てられているかを把握している場合に，スレッドの振舞いの目標点を簡潔に記述できることである．それに対して，相対指定は以下のような記述法を用いる．ここでは `hopalong` キーワードについて述べる．

- `hopalong(` *SrcLinkID* `)`

これは，現ノードに指定されているリンクを `along` 以下で指定し，そのリンクの接続先を目標ノードとして指定する．この記述の利点は，目標ノードがどのプロセッサに割り当てられているのかをいっさい意識することなくスレッドの振舞いを記述できることである．

（4）シミュレーション空間の参照と更新

M++言語は，スレッドが，現ノード，および当該スレッドが存在する現在のデーモン（以下，現デーモン）

の属性を参照するために，それぞれ以下のキーワードを提供する．

- `node`
- `daemon`

属性には，たとえばノードであれば，ID，そのノードに存在するスレッド数等があげられる．これらは，それぞれ `node.id()`，`node.thr_num()` と記述することによって得られる．ノードに対して，それらの型がすでに判明している場合はキャストが可能となり，以下の記述を使用する．

- `node<`*ClassName*`>`

性能低下を防ぐため，C++言語の `dynamic_cast` とは異なり，キャストに失敗してもエラーを返さない．処理はそのまま続行される．

(5) セルオートマトン

M++言語は，セルオートマトンのルールの記述とその起動をサポートする．セルオートマトンの動作ルールは，ノードの `clock` メソッドに記述する．近隣のノードの情報はリンクを介して得ることができる．セルオートマトンは，スレッドが `clocking` キーワードを実行したときに起動される．M++に存在するすべてのノードの `clock` メソッドがそのときにいっせいに起動し，シミュレーション空間の状態が更新される．

(6) GVT

カリフォルニア大学アーバイン校で開発された Messengers は，エージェントの視点からのシミュレーション記述において GVT を実現している[8]．M++言語では言語レベルで GVT をサポートしている．

現 GVT の値は `gvt_time` キーワードによって得られる．GVT に基づく計算に参加するスレッドは，現 GVT で行うべき処理を終えた後，`gvt_delta` キーワードによって実行を再開すべき時間を現 GVT に対して相対的に指定し，スリープ状態へと移行する．M++は，現 GVT で実行すべきスレッドがすべてスリープ状態に移行したかを分散終了検知アルゴリズムによって検出する[8]．その後，GVT を進め，上記の動作を繰り返す．GVT の管理はデーモンが行うので，ユーザは，シミュレーションの記述に集中することができる．

2.3 性能維持のための実装技術

M++は，記述性の向上に主眼をおいて設計されていると同時に，台数効果を得るための性能の維持にも重点を置いている．本節では，性能の維持を目的とした実装技術のうち，(1) 自己移動型スレッド，ならびに (2) S スレッドライブラリ，(3) ゼロコピー通信について，それぞれの特長を述べる．これら M++の実装には C++言語を使用している．

(1) 自己移動型スレッド

M++が扱うエージェントは，シミュレーション空間の変化に即座に反応しながら情報交換を行うリアクティブエージェントである[2]．そこで M++では，多数のエージェントの高速実行と高速なプロセッサ間移動を実現するために，エージェントを自己移動型スレッドとして実装し，さらに以下の方針に基づき送受信すべき実行状態を可能な限り軽減する．

スレッドが所有する実行状態には，(1) 実行コード，ならびに (2) スタック，(3) レジスタ，(4) ファイルや標準入出力等への I/O ディスクリプタの 4 点があげられる．(1) 実行コードに関しては，NFS で接続されているクラスタを対象とすることにより，明示的にコード転送を行う必要がない．(2) スタックに関しては，局所変数，および関数の戻り番地が格納されているが，スレッドが使用できる変数を `thread` スコープに定義した変数（以下，スレッド変数）に限定することによって，局所変数部分を実行状態から削除できる．また，`main` 関数以外のメソッドでスレッド移動を行う場合には，一般的にスタック内の関数の戻り番地の補正が必要となるが，本研究では，そのメソッドをコンパイル時にインライン定義して関数の実体を `main` 関数に埋め込んでいる．よって，移動時には戻り番地の情報を必要とせず，スタックに関してはスレッド変数のみを移動すればよい．(3) レジスタについては，実装対象としている Pentium，および Sparc プロセッサでは，スレッドの再開時に復元すべきレジスタがプログラムカウンタ（以下，PC）のみとなる．これは，再開時に新たなスレッドを生成することによって，再開に必要な PC 以外のレジスタ値はすべて移動先の環境に適切なように再設定されるからである．その後，PC のみを復元することによって任意の場所から実行を再開できる．PC の獲得と復元にはそれぞれ `setjmp` 関数と `longjmp` 関数を用いる．(4) ファイルや標準入出力への I/O ディスクリプタについては，それらを獲得したプロセッサでのみ使用可能とすることによって実行状態から削除する．以上に示した方針により，送受信すべきスレッドの実行状態は PC とスレッド変数のみに軽減される．

実行状態を軽減することによって，記述性に以下の 3 つの制限が生まれる．第 1 の制限は，インライン展開でメソッドを `main` 関数に埋め込むために再帰を使用できないことである．第 2 の制限は，送受信するスタック領域をスレッド変数のみに限定するためにメソッド内で局所変数が使用できないことである．第 3 の制限は，I/O ディスクリプタの使用を制限すること

によって遠隔プロセッサからの入出力を他のプロセッサへリダイレクトできないことである．これらに関しては，記述性と性能のトレードオフとなる．

(2) S スレッドライブラリ

本研究では，マルチエージェントシミュレーションに必要となる多数のスレッドを効率良く動作させるために **S スレッドライブラリ**を開発した．S スレッドライブラリの最大の特徴は，マルチエージェントシミュレーションに特化したスケジューリングを行うことである．スケジューリング方式のデフォルトは，生成したスレッドすべてに CPU 使用権を渡し，その後，デーモンに CPU 使用権を移す．この方式により，多数のスレッドを生成したときにも，すべてのスレッドに CPU 使用権が与えられた後にデーモンが実行されるため，スレッドの実行を円滑に進めることができる．また，個々のシミュレーションによって，デーモンとスレッドの実行比率を変えることも可能である．S スレッドライブラリでは，シグナル処理・イベント処理等，マルチエージェントシミュレーションに必須ではない機能を削除し，それらに費やされる処理を削減してスレッド実行の高速化を図っている．S スレッドライブラリは，現在 Intel 版 Solaris7，および Solaris8，Red Hat Linux 7.2，Sparc 版 Solaris2.5 に対応しており，その上での M++の動作も確認済みである．

(3) ゼロコピー通信

M++は，TCP/IP 通信の他に，Myrinet によるゼロコピー通信[22] によって，より高速なパケット送受信を実現する．ゼロコピー通信ライブラリでは，TCP/IP におけるカーネル内部へのパケットコピー操作を省略し，ピンダウン領域に存在するパケットを Myrinet を介して直接送受信する．この方式は，TCP/IP 通信におけるカーネル内部のコピーのオーバヘッドを低減し，パケット送受信を高速化する．

3. 記 述 性

本章では，実際のシミュレーションの記述例として，人工社会シミュレーションを取り上げ，M++言語と MPI による記述性の評価を行う．

人工社会シミュレーションは，J.M. Epstein らが考案した社会プロセスのエージェントベースモデルである[15]．このシミュレーションは，Sugarscape と呼ばれる，砂糖が分布したセルオートマトンのシミュレーション空間上で，様々な行動ルールを持った多数のエージェントの動作をシミュレートし，そこから生じる，自己組織化したエージェントの社会構造を観察することを目的とする．本節では，Sugarscape において，砂糖の分布と再生に加えて公害の発生と拡散をモデル化し，その公害をできるだけ避けながら砂糖を摂取するためのエージェントの動作ルールを記述する．

Sugarscape の構成は以下のとおりである．Sugarscape のトポロジは 2 次元トーラス空間であり，すべての (x, y) 座標は［砂糖の最大容量，砂糖の現在量，公害量］という 3 つの要素を持つ．初期状態における Sugarscape には，右上と左下に砂糖の山が生成されている．山を降りるにしたがって砂糖の量は減少し，最終的に 0 となる．各 (x, y) 座標の砂糖は，1 単位時間ごとに 1 単位ずつ最大容量になるまで砂糖が再生される．砂糖が 1 単位再生されると，生産公害として 1 単位の公害がその座標に蓄積される．蓄積された公害は 1 単位時間ごとに拡散していく．

エージェントは［視力，代謝率，砂糖貯蓄量］という 3 つのパラメータを持つ．これらの初期値は一定範囲でランダムに与えられる．シミュレーションの初期状態において，エージェントは Sugarscape 上にランダムに配置される．エージェントの行動ルールは以下のとおりである．

(i) 視力 v，代謝率 m を持つエージェントは，現在の場所から直交 4 方向を v 単位まで見渡すことができる．
(ii) それらの中で砂糖対公害比が最大の場所を見つけて移動する．
(iii) その場所に蓄積されているすべての砂糖を摂取する．
(iv) 移動時に自分の砂糖を m 単位消費する．同時に m 単位の消費公害が移動先に蓄積される．
(v) 自分の砂糖の貯蓄がなくなるまで，(i)〜(iv) の動作を繰り返す．

(1) MPI による記述例

MPI による記述例を**図 3** に示す．最初に各プロセッサにエージェントの受信キュー `incomingAg`，ならびに配置キュー `residingAg`，送信キュー `outgoingAg` が作成される（第 2 行）．受信キューは他のプロセッサから受信したエージェントを，配置キューは現プロセッサに配置済みのエージェントを，送信キューは他のプロセッサへ送信すべきエージェントを格納する．`Agent` クラスにはエージェントの各動作メソッドと状態変数が定義されている（第 7〜45 行）．2 次元トーラスの Sugarscape は 1 次元配列で表現され，各プロセッサに `arraySize` 個ずつ割り当てられる（第 5 行）．そこに `Agent` オブジェクトがランダムに配置される．

`main` 関数から実行が始まり，GVT が偶数のときはエージェント群の動作を（第 50 行），奇数のときは

```
1  int        gvt;
2  AgQueue incomingAg, residingAg, outgoingAg[CPUNum];
3  const int simSize( 200 );
4  const int arraySize( simSize * simSize / CPUNum );
5  Node  node[arraySize];
6
7  class Agent {
8  public:
9    enum { searching, movingToMax };
10   Agent() : dir( Agent::north ), i( 1 ),
11             status( searching ) {}
12   int getCPUid()
13     { return ( ix + iy * simSize ) / arraySize; }
14   void calcIxIy( int dir ) {
15     switch( dir ) {
16       case east:  ix=(ix+1) % simSize; break;
17       case south: iy=(iy+1) % simSize; break;
18       case west:  ix=(ix>0)?ix-1:simSize-1; break;
19       case north: iy=(iy>0)?iy-1:simSize-1; break;
20   } }
21   void pointNextElem() {
22     if( dir == Agent::north && i == 1 )
23       { orgIx = ix; orgIy = iy; }
24     if( dir <= Agent::west )
25       if( i <= visible ) {
26         if( maxRatio >= curNode->getRatio() ) {
27           maxRatio = curNode->getRatio();
28           maxIx = curNode->getIx();
29           maxIy = curNode->getIy();
30           i++;
31         }
32         calcIxIy( dir );
33       }
34       else { i=1; dir++; ix=orgIx; iy=orgIy; }
35     else { status = movingToMax;
36            dir = Agent::north; i = 1; }
37   }
38   void effectToSimSpace()  { ... }
39   ...
40 private:
41   enum { north, east, south, west };
42   int visible, metabolism, sugar, ix, iy,
43       orgIx, orgIy, maxIx, maxIy, dir, i, status;
44   float maxRatio;
45 };
46
47 int main( int argc, const char** argv ) {
48   for( gvt = 0 ; gvt < maxSimTime ; gvt++ ) {
49     switch( gvt % 2 ) {
50       case 0: agentSimulation();          break;
51       case 1: simSpace->clocking( gvt ); break;
52 } } }
53
54 void agentSimulation() {
55   do {
56     incomingAg.enqueue( residingAg );
57     while( !incomingAg.isEmpty() ) {
58       ag = incomingAg.dequeue();
59       ag->pointNextElem();
60       if( ag->getStatus()==Agent::searching )
61         outgoingAg[ag->getCPUid()].enqueue( ag );
62       else {
63         elem = simSpace->getElement( ag );
64         ag->effectToSimSpace( elem );
65         if( ag->getSugar() )
66           ag->terminate();
67         residingAg.enqueue( ag );
68   } } } while( exchangeAgents() > 0 );
69 }
70
71 int exchangeAgents() {
72   packedAgent = packAgent( outgoingAg );
73   sendAgent( packedAgent );
74   recvedAgent = recvAgent();
75   incomingAg.enqueue( unpackAgent(recvedAgent) );
76   countIncomingAg = incomingAg.count();
77   MPI_Allreduce( countIncomingAg,
78                  totalExchangedAg, ... );
79   return totalExchangedAg;
80 }
```

図 3 MPI を用いた人工社会シミュレーション
Fig. 3 Artificial societies simulation using MPI.

Sugarscape において砂糖の再生，および公害の発生と拡散のルールを実行する（第 51 行）．エージェント群の動作を実行する agentSimulation 関数（第 54〜69 行）では，配置キューに存在するエージェントを受信キューに移動し（第 56 行），その受信キューからエージェントを 1 個ずつ取り出してそのメソッドを呼び出す（第 57〜67 行）．エージェントの次の移動先を決定する pointNextElem メソッドを呼び出すと（第 59 行），前述のエージェントの行動ルール (i)〜(ii) が実行される．その詳細を以下に示す．

エージェントは現在見渡している方向を示す変数 dir，その方向への移動距離を示す変数 i，視力範囲を見渡したことを示す変数 status を持つ（第 43 行）．pointNextElem メソッド（第 21〜37 行）は，現在のエージェントの位置が，それまで調べた場所よりも砂糖対公害比の大きな場所かを調べ（第 26 行），そうであればその値と座標を保存する（第 27〜29 行）．そして，次の移動場所を計算する calcIxIy メソッドを呼び出す（第 32 行）．calcIxIy メソッド（第 14〜20 行）では，2 次元トーラスの特徴にしたがって次の移動場所を計算する．実際の移動は，後述の exchangeAgents 関数で行われ，Agent クラス内では行われない．

pointNextElem メソッドの実行を終えた後，次の移動先が他のプロセッサへの移動をともなうならば，送信キューに格納される（第 61 行）．また，砂糖対公害比が最大の場所を見つけたなら，その場所に対して行動ルール (iii)〜(iv) を実行した後，そのエージェントは配置キューに格納され，その場所に落ち着く（第 63〜67 行）．これらの動作は，すべてのプロセッサにおいて，移動するエージェントが存在する間繰り返される（第 68 行）．移動中のエージェント数の計算と送信キューに入っているエージェントの送信は，exchangeAgents 関数（第 71〜80 行）によって行われる．exchangeAgents 関数では，送信キューに存在するエージェントを連続領域にまとめて送信した後（第 72〜73 行），他のプロセッサからエージェントを受信して受信キューに格納する（第 74〜76 行）．その後，MPI_Allreduce 関数を用いて明示的に通信を行い，各プロセッサで移動したエージェント数を合計して（第 77 行），その値を返値とする（第 79 行）．その返値が 0 になれば agentSimulation 関数が終了する．

（2）M++言語による記述例

次に M++言語による記述を**図 4** に示す．M++言語では，Node 型のノードと Link 型のリンクから構成される 2 次元トーラスによって Sugarscape が表現される．Node クラスにはセルオートマトンのための clock メソッドが定義されている（第 3 行）．シミュレーションは，CreateSpace スレッド（第 6〜29 行），Agent スレッド（第 31〜64 行），および ClockingAgent スレッド（第 66〜73 行）の 3 個のスレッドから構成される．

CreateSpace スレッドは，クラスタの各プロセッサに自スレッドを分散し（第 9〜10 行），並列に 2 次

```
1  const int simSize( 200 );
2
3  class Node { public: void clock(int t){ ... } ... };
4  class Link { ... };
5
6  thread CreateSpace {
7  public:
8    void main() {
9      for( i = 1 ; i < daemon.total() ; i++ )
10       if( fork(INIT@i)==Errno::forkChild ) break;
11     startNodeId = simSize*simSize / daemon.total()
12                   * daemon.id();
13     // create horizontal torus
14     for( j=0 ; j < simSize/daemon.total() ; j++ )
15       for( i = 0 ; i < simSize ; i++ ) {
16         nodeId = startNodeId + j * simSize + i;
17         create node<Node> with( nodeId );
18         if( i % simSize != 0 )
19           create link<Link> with( Agent::east )
20             to( nodeId ) with( Agent::west );
21         else
22           if( node<Node>.id() != INIT )
23             create link<Link> with( Agent::east )
24               to( nodeId-size ) with(Agent::west);
25         hop( nodeId );
26       }
27     // create vertical torus
28     ...
29 } };
30
31 thread Agent {
32 public:
33   enum { north, east, south, west };
34   Agent( int argc, const char** argv ) :
35     visible( atoi( argv[4] ) ),
36     metabolism( atoi( argv[5] ) ),
37     sugar( atoi( argv[6] ) ) {}
38   void hopToMaxRatioNode() {
39     orgNodeId = node.id();
40     orgDaemonId = daemon.id();
41     for(dir=Agent::north;dir<=Agent::west;dir++) {
42       hop( orgNodeId, orgDaemonId );
43       for( i = 1 ; i <= visible ; i++ ) {
44         hopalong( dir );
45         if( maxRatio >= node<Node>.getRatio() ) {
46           maxRatio    = node<Node>.getRatio();
47           maxNodeId   = node.id();
48           maxDaemonId = daemon.id();
49     } } }
50     hop( maxNodeId, maxDaemonId );
51   }
52   void effectToSimSpace()  { ... }
53   void main() {
54     while( gvt_time() < maxSimTime ) {
55       hopToMaxRatioNode();
56       effectToSimSpace();
57       if( sugar < 0 ) terminate;
58       gvt_delta( 2 );
59   } }
60 private:
61   int visible, metabolism, sugar, orgNodeId,
62       orgDaemonId, maxNodeId, maxDaemonId, dir, i;
63   float maxRatio;
64 };
65
66 thread ClockingAgent {
67 public:
68   void main() {
69     gvt_delta( 1 );
70     while( gvt_time() < maxSimTime ) {
71       clocking( gvt_time() );
72       gvt_delta( 2 );
73 } } };
```

図 4 M++言語を用いた人工社会シミュレーション
Fig. 4 Artificial societies simulation using M++.

元トーラス空間を生成する．本スレッドは，ノードを生成して（第 17 行），そのノードを現ノードとリンクで連結し（第 19 行），接続先のノードに移動する（第 25 行）ことを繰り返し行う．トーラスの X 座標が `simSize` まで到達したときは，現ノードとその行の一番左のノードをリンクで連結する（第 23 行）．これらの作業で 1 行のトーラスが生成され，以上を繰り返し行うことによって各プロセッサの割当て分のトーラスを生成する．その後，垂直方向を同様にリンクで連結することによって，各プロセッサに分散された 2 次元トーラスが完成する．

`Agent` スレッドは，Sugarscape 上にランダムに多数配置される．投入時にコンストラクタによってスレッド変数を初期化した後（第 35〜37 行），`main` 関数の実行を開始する．`Agent` スレッドは GVT の偶数時に動作する．この偶数時にエージェントの行動ルール (i)〜(v) の動作を行う（第 55〜57 行）．特に `hopToMaxRatioNode` メソッド（第 38〜51 行）には，砂糖対公害比の大きな場所を探索する手続きがメソッド内で完結した形で記述されている．すなわち直交 4 方向を視力の範囲で移動し（第 44 行），もし現ノードが，それまで調べたノードよりも砂糖対公害比が大きければ（第 45 行），その値と位置を保存する（第 46〜48 行）．そして，すべて調べ終えた後，砂糖対公害比の最大の場所へ移動する（第 50 行）．一連の行動ルールを終えた後は，GVT が 2 単位時間進むまでスリープする（第 58 行）．

一方，`ClockingAgent` スレッドは，GVT が 0 の時に 1 単位時間スリープすることによって，GVT の奇数時に動作するようにしている（第 69 行）．`main` 関数では，`clocking` キーワードを用いてクロッキング機構を起動する（第 71 行）．クロッキング機構が起動されると，すべてのノードは，それぞれ砂糖の再生，および公害の発生と拡散のルールを実行する．クロッキング機構の動作が完了すると，`ClockingAgent` スレッドの GVT が 2 単位時間進み（第 72 行），次の GVT 奇数時までスリープする．

(3) M++言語と MPI の比較

以上の記述例から，M++言語と MPI の記述性に関して以下の 4 点に着目して比較・評価を行う．

(I) エージェントの記述

M++言語では，図 4 の `hopToMaxRatioNode` メソッドのように，エージェントの状態変数と動作の流れを `thread` スコープの中に完全にカプセル化できるので，シミュレーションアルゴリズムに合致した記述を行うことができ，記述性・保守性の向上がもたらされる．それに対して，MPI では，エージェントは単にオブジェクトとして表現され，図 3 の第 57〜67 行のように，`main` 関数がオブジェクトのメソッドを順番に呼び，エージェントを動作させる．エージェントの動作の流れをオブジェクト中にカプセル化することはできず，かつエージェントの処理の中に移動を記述できないため，シミュレーションアルゴリズムからのパラダイム転換が必要となり記述性が低下する．これによりプログラムの行数も極端に増加してしまうため保守性も低下する．

(II) 通信の記述の有無

M++言語は，スレッドの移動に関して，デーモンが通信の必要性を判断して実際の通信を行うので，ユーザは通信をいっさい記述する必要がない．すなわち，図 4 の第 44 行のように，ユーザはリンクに沿ってスレッドを移動することを単純に記述するだけでよい．もし，リンク先のノードが他のデーモンに存在する場合は，デーモンがスレッド移動のための通信を担う．しかしながら，MPI では，図 3 の第 61 行のように，ユーザが，エージェントの位置から，次に移動するシミュレーション空間の要素がどのプロセッサに存在するかを計算し，`exchangeAgents` 関数に示されるような，スレッドのキューへの出し入れ・パック・アンパック等，通信に必要な前処理を行った後，実際の通信関数を明示的に記述しなければならない．このことは，きめ細かい通信が可能である反面，その記述に多大な労力を必要とする．

(III) シミュレーション空間の分散配置

M++では，図 4 の `CreateSpace` スレッドに記述しているように，各プロセッサにスレッドがノードを配置し，それらを相対的にリンクで接続することによって 2 次元トーラスを表現している．この手法の利点は，シミュレーション空間の分散トポロジを直観的に記述できることである．また，同図の第 44 行のようにスレッドがリンクに沿って移動することによって，シミュレーション空間の分散トポロジにまったく依存しないシミュレーション記述が可能となる．一方，MPI では，図 3 の第 5 行に示すように，各プロセッサに配列で表現されたシミュレーション空間を割り当てる．この方法を用いた場合，エージェントの座標計算（`Agent::calcIxIy` メソッド），およびその座標がどのプロセッサに存在するかの計算（`Agent::getCPUid` メソッド）をユーザが明示的に記述しなければならない．これらの記述は，静的な分散トポロジであれば比較的容易である．しかしながら，動的にシミュレーション空間の大きさが変化し，分散トポロジもそれにともなって変化する場合，これらを記述することは工夫を要し，プログラム記述に労力を費やす．

(IV) エージェント間の同期

M++言語では，GVT キーワードを用いて，エージェント間の同期を簡潔に記述できる．しかしながら，MPI の場合は，図 3 の第 68 行に示されるように，すべてのプロセッサにおいて，エージェントの移動が終わるのを検知するまで互いに通信を繰り返して，エージェント間の同期を実現する必要がある．この分散終了検知の記述は，同期にかかわるエージェントがどのようなタイミングで同期をとるかによって変化するため，シミュレーションに大きく依存する．よって，これらの記述をシミュレーションごとに記述するのは，ユーザに大きな負担を強いることとなる．

以上を総合して評価を行った場合，MPI では，ユーザがシミュレーション内部のデータ構造を決定し，かつ適切な通信の制御を行うことが可能である一方，その作業に多大な労力を必要とするといえる．また，実行の流れがつねに 1 つであるため，多数のエージェントをすべてメインプログラムと各メソッドで処理しなければならず，シミュレーションアルゴリズムに沿った記述を実現するとはいえない．さらにユーザは，シミュレーションのアルゴリズムに通信処理を挿入しなければならないため，記述性が低下し，デバッグも煩雑になる．これに対して，M++は，シミュレーションアルゴリズムと合致したエージェント指向のプログラミングパラダイムによって，簡潔にシミュレーションを記述することができる．さらにそれはシミュレーション空間の分散トポロジにまったく依存しない．

4. 性　　能

本章では，人工社会シミュレーションにおける M++と MPI の性能を測定し，比較・評価する．性能測定には**表 1** に示す測定環境を用い，使用台数を変化させて測定を行う．

200 × 200 の Sugarscape 上に 5000 エージェントをランダムに配置して，シミュレーションを 300 単位時間行ったときの実行時間を M++と MPI において，それぞれ TCP/IP とゼロコピー通信を用いて測定する．その結果を**図 5** に示す．

MPI に関しては，TCP/IP を用いた場合，台数効果が観測されない．これは GVT を進めるにあたって，エージェントの移動があるたびに同期通信が発生するためである．使用台数が増加するにつれて，この同期通信による通信負荷は顕著となり，性能が低下していくと推察する．一方，ゼロコピー通信を使用した場合は，通信性能の向上によって同期通信による通信負荷

表 1 測定環境
Table 1 Environment for measurement.

CPU	Athlon 1 GHz (FSB 266 MHz)
メモリ	DDR-SDRAM 256 MB
NIC	Myrinet M2MPCI32 B
スイッチ	M2M-DUAL-SW8
プロトコル	TCP/IP，GM (zero-copy)
OS	Solaris8
コンパイラ	gcc-2.95.3
クラスタ台数	8

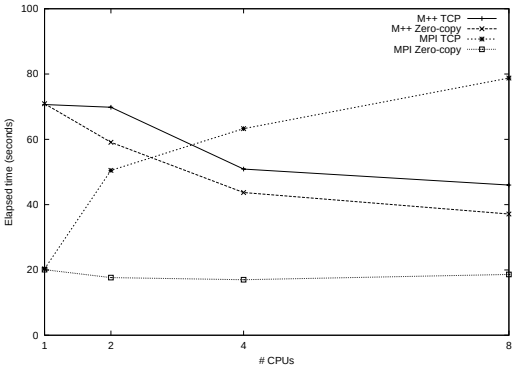

図 5 人工社会シミュレーションの性能
Fig. 5 Performance of artificial societies.

が軽減され，性能低下を回避することができる．しかしながら，台数効果は 4 台時の 1.18 倍が最大で，8 台時は 4 台時よりも性能が低下する．

M++に関しては，TCP/IP を使用した場合，8 台時で 1.53 倍の台数効果が得られた．シミュレーションの空間的局所性を利用する M++の特徴がもたらした結果と推察する．

2 台時に性能の向上がそれほど観測されないにもかかわらず，4 台時において性能が大幅に向上する理由については，ホスト間をまたぐリンクデータの共有を効率よく行うことができるためと推察する．すなわち，2 次元トーラスを中心から上下に帯状に分割した場合，2 台時においては，プロセッサをまたぐリンクデータの獲得要求がすべてもう一方のプロセッサに対して送信されるのに対し，4 台時の場合は上下 2 台のプロセッサに分散されて送信される．このことによって，一方の通信の待ち時間を他方の通信に割り当てることができ，全体性能の向上がもたらされたと推察する．一方，ゼロコピー通信を用いた場合は，8 台時で 1.90 倍の台数効果を得ることができた．2 台時においても TCP/IP と比較して台数効果が 1.01 倍から 1.20 倍へと向上しており，通信性能の向上が全体性能の向上につながったと考察する．

両者を比較した場合，M++は，MPI と比較して台数効果が顕著に現れている．この理由は，MPI が，GVT の分散終了検知アルゴリズムにおいて，すべてのエージェントが移動を停止するまで隣接するプロセッサと何度も同期通信を行うのに対し，M++は，非同期通信によって，現プロセッサでスレッドが移動していない場合に限り，隣接するプロセッサと通信を行うよう GVT 機構が最適化されているからである．MPI において，M++と同等の非同期通信を用いた GVT 機構を用いようとするならば，メッセージを受信しながらエージェントを動作させる仕組みを実現する必要があり，シミュレーションアルゴリズムからのさらなるパラダイム転換が要求される．

実行時間で比較した場合，1 台時では，M++は MPI の 3.50 倍の時間を必要としているが，TCP/IP の 8 台時では，M++は MPI の 0.58 倍まで実行時間が短縮されている．ゼロコピー通信においては MPI の 1.99 倍に実行時間を抑えることができている．1 台時における性能のオーバヘッドの要因としては，スレッドの生成・削除とそれらのコンテキストスイッチがあげられる．また，複数台時におけるオーバヘッドには，スレッド移動にともなう通信遅延があげられる．これらのオーバヘッドについては，実行時間の内訳を綿密に検証する必要があるため，追って詳細を報告する．

5. 関 連 研 究

M++を（1）モバイルエージェント，ならびに（2）スレッドマイグレーション，(3）マルチエージェントシミュレータと比較する．また，これらのシステムを用いてマルチエージェントシミュレーションを実行した場合において，M++との相違点を記述性と性能の面から論じる．

（1）モバイルエージェント

IBM-Aglets [13] 等に代表されるモバイルエージェントは，主にインターネット上での電子商取引，情報検索等に用いられ，遠隔のプロセッサに移動して作業を行うための多くの機能を備える．これらの多くはスクリプト言語で記述され，性能面よりもむしろ機能の豊富さ・移植性・セキュリティに重点が置かれて設計されている．並列マルチエージェントシミュレーションでは多数のエージェントが高速に移動しながら計算を行う必要があるため，以上の特徴を持つモバイルエージェントではシミュレーション性能に限界が生じる．このことはスレッド移動の性能を他の通信システムと比較した著者らの先行研究[17] においても明白である．

（2）スレッドマイグレーション

スレッドマイグレーションは分散メモリ環境において，リモートメモリアクセスの軽減，ならびに負荷分散，RPC，リモートエグゼキューションの有効な実装方法として応用されている．これらを用いてマルチエージェントシミュレーションを行うことは可能であるが，以下のような制限を受ける．たとえば，Nexus [7] は，強いマイグレーションをサポートしていないため記述性に劣る．Olden [20] は，コンパイラが移動命令を挿入するため，ユーザがスレッド中に明示的に移動命令を記述することができない．同様に，UPVM [3]

に関しても，スケジューラによるスレッド移動のみをサポートしているため，スレッド移動を明示的に記述できない．PM2[19]は，スレッド間通信にメッセージパッシング方式のみを提供している．以上のことから，これらのスレッドマイグレーションシステムを並列マルチエージェントシミュレーションに使用するためには，多大な労力と工夫を要するといえる．

(3) マルチエージェントシミュレータ

マルチエージェントシミュレータにも，数々のシミュレータが提案されている．SWARM[16]は，Objective-Cによるフレームワークによってエージェントを記述し，それらエージェントの群れである swarm に対してイベント列を与え，シミュレーションを進行させるシミュレータである．SWARM のツールキットである MAML[12]は，シミュレーションの記述を Objective-C のフレームワークからエージェント指向によるプログラミングへと発展させている．また，MAS[23]は教育用のマルチエージェントシミュレータであり，GUI を用いてエージェントのルール，ならびにシミュレーション空間の設定，シミュレーションの実行を簡潔に行うことができる．しかしながら，これらは並列処理に焦点を置いておらず，ユーザ自身が，セルを主体としたモデル，またはエージェントを主体としたモデルによって並列化を実現する必要がある．

一方，StarLogo[4]では，Logo 言語で記述されたエージェントがスレッドで実行されるため，SMP 上においてスレッドによる並列マルチエージェントシミュレーションを行うことができる．しかしながら，StarLogo は分散共有メモリの有無にかかわらず，クラスタ上における並列処理には対応していない．また，シミュレーション空間は 2 次元格子空間に限定される．さらにリンクの概念がないため，シミュレーション空間を構成する要素を任意のトポロジで結合することができない．

M++は，記述性を重視した並列マルチエージェントシミュレータである．M++は，M++言語を提供し良好な記述性を実現すると同時に，自己移動型スレッド，ならびに S スレッドライブラリ，ゼロコピー通信によって性能低下を軽減し台数効果を維持する．

6. おわりに

本論文では，スレッドがクラスタ上を移動し，相互作用を及ぼしながらシミュレーションを進める並列マルチエージェントシミュレータ M++の記述性と性能について論じた．比較対象としては，つねに性能を追求できるものの通信・並列処理にユーザの多大な労力を要する MPI を選択した．その結果，本研究で得られた知見は以下のとおりである．3 章に述べたように，記述性に関して M++言語は，MPI におけるマルチエージェントシミュレーションの記述と比較して，並列処理に必要な通信をまったく意識せずに，シミュレーション空間のクラスタ上への分散配置を直観的に記述でき，エージェントの視点からのシミュレーション記述をきわめて容易に実現できる．性能に関しては，実性能に着眼すれば MPI が優れているのはいうまでもない．しかしながら，4 章に述べたように，M++は，人工社会シミュレーションにおいて，MPI と比較して台数効果が顕著に表れた．以上の検証結果から，M++は，シミュレーションを簡潔に記述して，多台数で構成されるクラスタを用いて台数効果を得ることができる，記述性を重視した並列マルチエージェントシミュレータといえる．

本研究の展望としては，デバッギングシステムと統合環境を開発し，シミュレーション構築においてユーザをいっそう支援する環境を提供することがあげられる．

参考文献

1) *Scientific American* (Apr. 2002).
2) Kendall, E.A., Krishna, P.V.M., Pathak, C.V. and Suresh, C.B.: Patterns of intelligent and mobile agents, *Proc. 2nd International Conference on Autonomous agents*, pp.92–99 (May 1998).
3) Casas, J., Konuru, R., Otto, S., Prouty, R. and Walpole, J.: Adaptive load migration systems for PVM, *Proc. Supercomputing '94*, pp.390–399, Washington D.C. (Nov. 1994).
4) Colella, V.S., Klopfer, E. and Resnick, M.: *Adventures in Modeling: Exploring Complex, Dynamic Systems with StarLogo*, Teachers College Press (2001).
5) Cugola, G., Ghezzi, C., Picco, G.P. and Vigna, G.: Analyzing mobile code languages, *Mobile Object Systems: Towards the Programmable Internet*, pp.93–110, Springer-Verlag, Heidelberg, Germany (1997).
6) Ferber, J.: *Multi-Agent Systems, An Introduction to Distributed Artificial Intelligence*, Addison-Wesley (1999).
7) Foster, I., Kesselman, C. and Tuecke, S.: The Nexus approach to integrating multithreading and communication, *Journal of Parallel and Distributed Computing*, Vol.37, No.1, pp.70–82 (Aug. 1996).
8) Fukuda, M., Bic, L.F. and Dillencourt, M.B.:

Global virtual time support for individual-based simulations, *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, pp.9–16, Las Vegas, NV (July 1998).
9) Fukuda, M. and Suzuki, N.: M++ user's manual, Technical report available in http://dept.washington.edu/dslab/m++/user_man.ps, University of Washington, Bothell, WA 98011 (Sep. 2002).
10) Fullford, D.: Distributed interactive simulation: Its past, present, and future, *Proc. Winter Simulation Conference*, pp.179–185 (Dec. 1996).
11) Grunwald, D.: A users guide to AweSime: An object oriented parallel programming and simulation system, Technical report CU-CS-552-91, Department of Computer Science, University of Colorado (1991).
12) Gulyás, L., Kozsik, T. and Fazekas, S.: The multi-agent modeling language, *Proc. 4th International Conference on Applied Informatics*, Eger-Noszvaj, Hungary, August 30–September 3 (1999).
13) Lange, D. and Oshima, M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison Wesley Longman, Reading, MA (1998).
14) Logan, B. and Theodoropolous, G.: The distributed simulation of multi-agent systems, *Proc. IEEE*, Vol.89, No.2, pp.174–185 (2001).
15) Epstein, J.M. and Axtell, R.L.: *Growing Artificial Societies*, MIT Press, Cambridge, MA 02142-1493 (1996).
16) Minar, N., Burkhart, R., Langton, C. and Askenazi, M.: The SWARM simulation system: A toolkit for building multi-agent simulations, Technical report, Argonne National Laboratory, University of Chicago (June 1996).
17) Suzuki, N., Fukuda, M. and Bic, L.F.: Self-migrating threads for multi-agent applications, *Proc. IEEE Computer Society International Workshop on Cluster Computing*, pp.221–228, Melbourne, Australia (Dec. 1999).
18) Collins, R. and Jefferson, D.: *Artificial Life II*, chapter Antfarm, pp.579–601, Addison-Wesley (1992).
19) Namyst, R. and Mehaut, J.: PM2: parallel multithreaded machine, a computing environment for distributed architectures, *Proc. ParCo'95*, pp.279–285. Elsevier Science Publishers (Sep. 1995).
20) Rogers, A., Carlisle, M.C., Reppy, J.H. and Hendren, L.J.: Supporting dynamic data structures on distributed-memory machines, *ACM TOPLAS*, Vol.17, No.2, pp.233–263 (Mar. 1995).
21) Dahmann, J.S., Fujimoto, R.M. and Weatherly, R.M.: The department of defense high level architecture, *Proc. Winter Simulation Conference*, pp.142–149 (Dec. 1997).
22) Myricom website, http://www.myri.com/.
23) 山影　進，服部正太：コンピュータのなかの人工社会—マルチエージェントシミュレーションモデルと複雑系，共立出版 (2002).
24) 大内　東，山本雅人，川村秀憲：マルチエージェントの基礎と応用，コロナ社 (2002).

(平成 14 年 12 月 21 日受付)
(平成 15 年 4 月 9 日採録)

鈴木　有也（学生会員）

1976 年生．1999 年筑波大学第三学群工学システム学類卒業．2001 年同大学大学院工学研究科博士前期課程修了．修士（工学）．現在，同大学院工学研究科博士後期課程在学中．モバイルエージェント，分散シミュレーションの研究に従事．ACM 学生会員．

福田　宗弘（正会員）

1963 年生．1986 年筑波大学第三学群情報学類卒業．1988 年同大学大学院修士課程理工学研究科修了．工学修士．同年日本アイ・ビー・エム（株）東京基礎研究所勤務．1995 年カリフォルニア大学アーバイン校，MS 授与．1997 年同大学博士課程修了．Ph.D. 同大学助手．1998 年筑波大学電子・情報工学系講師．2001 年ワシントン大学バセル校アシスタントプロフェッサー．モバイルエージェント，分散シミュレーションの研究に従事．

和田　耕一（正会員）

1956 年生．1984 年神戸大学大学院システム科学専攻博士課程修了．同年，同大学大学院自然科学研究科助手．1987 年筑波大学電子・情報工学系講師．助教授を経て 1999 年教授．現在に至る．1992〜1993 年カナダ，ビクトリア大学客員研究員．並列・分散処理とコンピュータアーキテクチャ，マルチメディア情報処理に関する研究に従事．学術博士．IEEE，ACM 等会員．