成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.51 No.1 (2014)pp.21-26

AgentSphereにおける適正負荷分散のためのエージェント生成制御機構

黒崎　信清*1，甲斐　宗徳*2

The Agent Generation Control Mechanism for the Proper Load Distribution in AgentSphere Network

Nobukiyo KUROSAKI*1, Munenori KAI*

ABSTRACT：In the AgentSphere network where a network situation is always changed, it becomes an important subject to perform parallel distributed processing using a number suitable for efficient behavior of the whole system of agents. In order that AgentSphere might make the cooperative behavior by agents efficient, a set of agents were generated by using AgentPool. In AgentPool, the user had to specify the number of agents to be generated, and it is fixed under their execution period. So, in this paper, the control mechanism of AgentPool was improved to change the number of agents dynamically, such that as the resource with which agents can work increased, the number of agents increased, and as the resource decreased, the number of agents decreased. Our proposed method not only makes resource requirements reasonable, but is useful for power consumption reduction of the whole system.

Keywords：mobile agent system, parallel distributed processing, load balancing, dynamic control

(Received March 31, 2014)

1．はじめに

1．1　研究背景

分散処理システムは構成しているノードに障害が発生しても処理を続けられるような耐故障性，構成するノードを柔軟に変更できるスケーラビリティ，複数マシンで処理を行うことによる処理速度の向上等様々な利点を見込むことができる。しかしこの利点を十分に発揮するためには，複数マシンを適切に扱うためのアルゴリズム，並列分散処理技術，ネットワークに対する高度な知識や各ノードの状態の把握等が要求される。そこで我々は，インフラとしてのネットワークを準備するだけでユーザが分散処理システムに関する知識，経験が十分に無くてもその恩恵を受けられるようなプラットフォーム，AgentSphere[1)]の開発を行ってきた。

AgentSphereは自律性を持ったモバイルエージェントシステムを採用している。その理由は，プログラムが稼働しているマシンがAgentSphereで構成されるネットワークから離脱した場合でも，他のマシンにエージェントが移動することで処理を続けられるという利点が存在しているからである。

1．2　AgentSphereの現状

我々のシステムは現在，エージェントの生成機能，エージェントの移動機能，エージェントのバックアップ機能は完成している。それに加えて，2012 年度の研究でMercuryアルゴリズムを使用したAgentSphereネットワークにおける情報共有機構 [2)]が構築された。以前まではAgentSphereネットワークへのマシンの参入・離脱の自動検知や性能情報の収集，マシン情報の更新をAgentSphere間のブロードキャストによって行っていたが，情報共有機構の実現によってほしい情報を効率的に検索することが出来るようになった。これによって，自律的に移動するエージェントの存在情報を正しく把握することが出来るようになった。

1．3　課題と目標

1.2 で挙げたように現在我々のシステムは基本となる動作はほぼ完成している。しかし，動作実験では数をカ

*1：理工学研究科理工学専攻博士前期課程学生
*2：理工学研究科理工学専攻教授(kai@st.seikei.ac.jp)

ウントするエージェントや，移動を繰り返すエージェントなどの簡単なエージェントの動作実験しか行われていない。過去の研究では*N*-Queens問題を複数のエージェントを用いてマスター，スレーブ方式で解くといった協調動作を行う実験も行われたが，問題を解くために生成されるエージェントの数はユーザが初期設定するしかなかった。ネットワークやマシン内の状況が常に変化していく環境においてエージェントの数が初期設定時のまま一定だと様々な問題が考えられる。

ネットワークに新しいマシンが参入した場合やネットワークからマシンが離脱した場合を考えてみる。新しいマシンが参入した場合，単純にエージェントの活動領域が増加する。その際に問題を解いているエージェントは空いているリソースがあるにもかかわらず初期設定したエージェントでのみ問題を解こうとするため，環境に応じて柔軟な動きをすることができないと考えられる。マシンが離脱した場合はリソースが減ることになるので，初期設定したエージェントを動かそうとした場合動作が遅くなる可能性がある。最悪の場合JVMの容量を超えてしまいAgentSphereそのものがダウンしてしまう可能性も考えられる。

そこで，本研究では探索解法プログラムに焦点を当て，現在のAgentSphere上で探索問題を複数のエージェントによる協調動作によって解決する手法を提案し，その間に生成されるエージェント数を動的に最適化することを目的としている。

2．探索問題

探索問題は問題サイズ*N*が増加するにつれて探索パターンが指数関数的に増加するので，*N*が一定以上の問題では逐次処理で解くことが非常に困難になる。

そこで，本研究ではエージェントを用いて並列分散処理させることによってこれらの問題の解決を試みる。今回扱う問題は次節で説明する。

2．1　*N*-Queens問題

N-Queens問題とは，*N*×*N*のチェス盤上に*N*個の縦，横，斜めにいくつでも進むことのできるクイーンの駒を，図1 に示すような互いに効き筋に入らない形で配置パターン数を求めるパズルである。現在，*N* =26 まで解の数が求められており，*N* =27 以降の解を得るため近年でも計算量削減手法[3]が考案されている探索問題である。

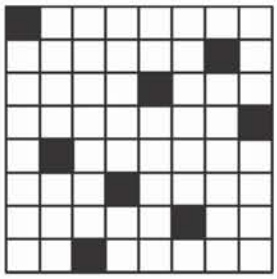

図 1　*N*=8 の配置例

3．解決手法

2 章で挙げた問題を複数のエージェントを用いて解決するために，探索の初期段階でこれらの問題を幅優先探索することによって図 2 のような部分問題に分割する。その後，分割された部分問題を図 3 のように各エージェントに深さ優先探索で解かせることによって並列処理させる。これは，幅優先探索で並列タスクを生成するにあたり，探索する深さ（分割レベルと呼ぶ）によって並列タスク数が大幅に変化することを利用するためである。

探索問題をいくつのエージェントで実行するかについては，マシンのコア数分のエージェントで実行を開始し，状況に応じてエージェントの数を変化させていく。エージェントに問題を割り当てた後，部分問題の数がエージェント数以上の場合，エージェントは自分に割り当てられた問題を終わらせた後に残りの問題を解く。

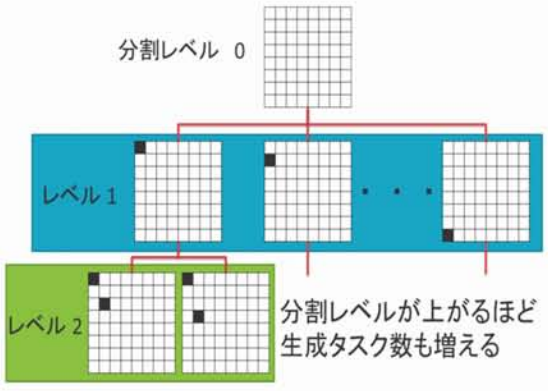

図 2　*N*-Queens問題　幅優先探索による問題の分割

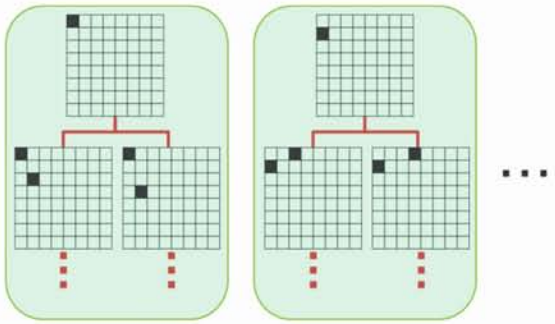

図 3　*N*-Queens問題　各エージェントが処理する部分問題

エージェント同士が協調動作をするために，互いに通信する必要がある。本システムではエージェント間通信をサポートするためのメッセージング機構が備わっているが，情報共有機構実装前は自律的に移動するエージェントを補足することが出来なかったのでAgentSphere間のブロードキャストによって実現していた。そのため，無駄な通信が発生しやすく，マシンの台数が増えるとネットワークの負荷が大きくなってしまうという点が問題になっていた。

3．1　メッセージング機構の改良

情報共有機構実装後はエージェントの情報を各AgentSphereで分散共有できる仕組みになったので，通信したいときにエージェントの検索 [4]を行うことでエージェントの位置を把握できるようになった。これを利用して，エージェント間で通信が発生するタイミングでエージェントの検索を行うことにより，ブロードキャストせずにエージェント間の通信が行えるように改良した。

また，あるエージェントが同じグループで処理をしている複数のエージェントに通信しようとした場合，1 対 1 通信をグループ内のエージェント数分記述しなければならなかったので，グループ間で通信が出来るように改良を加えた。

3．2　複数エージェントでの協調動作

探索問題を複数のエージェントで解決するための手法として，エージェントをマスターとスレーブに分けて処理させる方法を採用した。

マスターは問題の分割，スレーブは分割された部分問題を解く処理をそれぞれ行う。マスター，スレーブ間で行われる解の収集，および解の更新は 3.1 で説明したメッセージング機構を用いて行う。

3．3　エージェント生成の制御機構

今回は以前までの研究で用いられていたAgentPool[5]とは別に新たなPoolを加えた。この新たなPoolは従来のAgentPoolと同じコンセプトで設計されているため，単一JVMで使うことが出来るThreadPoolと同様の働きを持つ。これにより，性能低下への対策アプローチになると共に，記述方法をそろえユーザの習得コストを減らすことができる。また，従来のThreadPoolとは異なり，AgentSphereでは複数のAgentSphere間をエージェントが自由に移動して処理を実行できるので，単一JVMで使えるThreadPoolよりも多くのコアを利用することが出来る。そのため，並列処理性能を上げることが期待できる。

次節以降で説明する新たなAgentPoolと従来のものとの大きな違いはエージェントの数が状況によって変動するということである。

3．3．1　freeNewCachedAgentPool

Pool生成後はタスクが投入されるまで待機する。タスクが投入されるたびにエージェントを生成し，生成されたエージェントはそれぞれ各タスクを実行する。ここで生成されるエージェント数に制限はないため，タスクの量が多く，タスク一つ一つが時間のかからない小さなものに対して使うことを想定している。

3．3．2　NewCachedAgentPool

NewCachedAgentPoolもプール生成後に 3.3.1 のプールと同様にタスクが投入されるまでは何もせずに待機する。タスクが投入されるとエージェントの追加，削除を行うエージェント（以下コントロールエージェントと呼ぶ）を 1 つ生成する。そして，コントロールエージェントが現在のネットワークの状況を取得し，今生成できるワーカエージェントの数（制限値）を計算しその数分ワーカエージェントを生成する。その後はタスクがすべて終了するまでコントロールエージェントが定期的にネットワークの状況を取得し，制限値と残りのタスク数を参照して新たにエージェントを生成する必要があるならばエージェントを生成（増加）させる。今タスクを解いているエージェントが制限値を上回っている場合はエージェントが持っているタスクを終了させ次第処理を停止（減少）させる。制限値はネットワークの状況に応じて動的に変化する。実験をするにあたってエージェントを生成できる数を設定しなければならなかったため，制限値の計算式は暫定的に以下のように定めた。

制限値 =

$$\sum_{i=1}^{n} \{(\text{マシン}i\text{のコア数} \times 2) - \text{マシン}i\text{で稼働しているエージェント数}\}$$

この制限値は各マシンで収容できるエージェント数を足したものである。これは各コアが各エージェントに対して 1 対 1 で対応することで各エージェントの処理を円滑にできると考えたからである。実際には各マシンで想定以上のエージェントが稼働していた場合の評価は 4.で検証する。

4．性能評価

性能の異なる複数のマシンでネットワークを構築し，

それらをネットワークに参加，離脱させることによってエージェント数の変化と実行時間を測定した。
測定に使用したマシンの性能は以下の表のようになっている。

表 1　各マシンの性能

マシン名	メモリ	プロセッサ
PC1～PC3	8GB	Intel®Core™i5-2400 CPU @3.10GHz(4コア)
PC4	4GB	Intel®Core™2Duo CPU E8400 @3.00GHz(2コア)
PC5	4GB	Intel®Core™i5-2500 CPU @3.30GHz(4コア)
PC6	8GB	Intel®Core™i7 CPU 870 @2.93GHz(8コア)

4．1　ネットワークの変化によるエージェント数の変化

今回作成した改良型AgentPoolはネットワークに参加しているマシン台数に応じて生成されるエージェント数を動的に制御し，エージェントの追加や削除を行うことができる。この実験ではワーカエージェントの数は動的に変動することになるが，ワーカエージェントの数を減少させる方法はワーカエージェントが持っているタスクを終了させ次第プールから削除するという方法 1 とワーカエージェントを減らさなければならなくなった時にワーカエージェントから 1 つ選択し，未処理のタスクをプールに戻すという方法 2 の 2 つあった。そこで 2 つの方法で実験を行い，結果を比較した。図 4，図 5 のグラフは 5 秒ごとのネットワークに参加しているマシン台数とエージェント数の推移グラフである。このグラフを見るとマシン台数が変化するとエージェント数も変化していることがわかる。

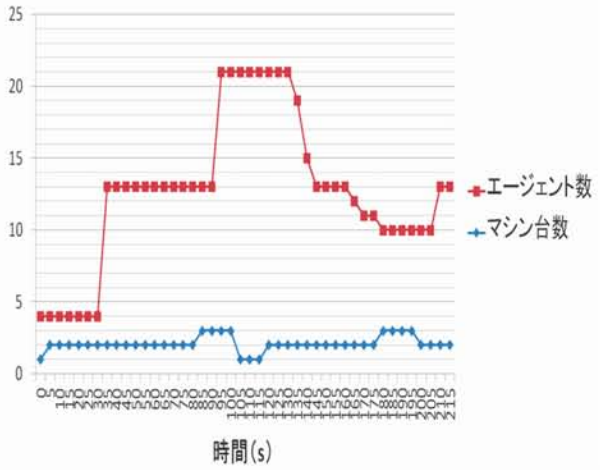

図 4　マシン台数とエージェント数の推移(方法 1)

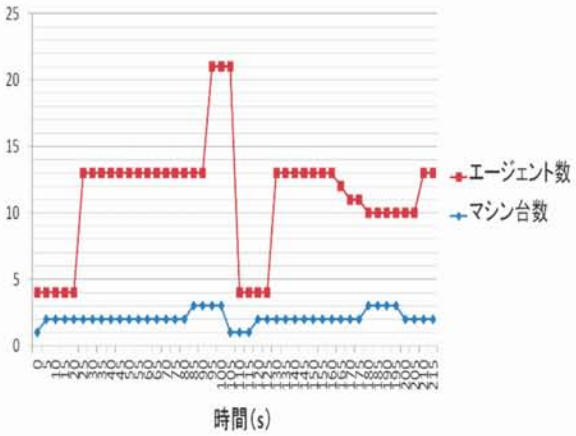

図 5　マシン台数とエージェント数の推移(方法 2)

以降の実験を行うに当たってこの 2 つの方法のどちらを採用するか決定するために，2 つの方法で実行時間の比較を行った。グラフは図 6 のようになった。

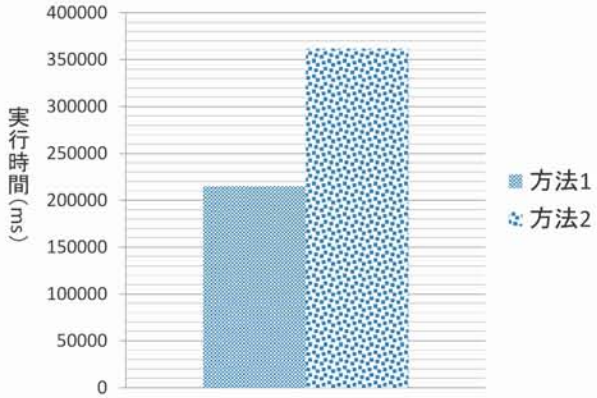

図 6　実行時間比較

これらの実験結果のグラフから方法 2 は方法 1 に比べてネットワークの状況の変化に対するエージェント数の変化がよりダイレクトに反映されているということがわかった。しかし，実行時間は方法 1 の約 1.7 倍の長さになった。これは，方法 2 がネットワークの変化をよりダイレクトにエージェント数に反映させるためにエージェントがタスクを実行中であった場合，実行中のタスクを中断させて未処理のタスクとしてタスクを返すことによってタスクの再計算が発生するということが原因だと考えられる。また，方法 1 はエージェント数を減らすとき，エージェントが所持しているタスクをすべて終わらせてからエージェントの処理を終了させ，プールから削除するので方法 2 のようにタスクの再計算が発生しないということも 2 つの実行時間の差に影響していると考えられる。

方法 1 ではネットワークからマシンが脱退した場合，タスクが終了するまでエージェントを減らすことができない。つまり，制限値を超えた数のエージェントが処理を実行する可能性がある。そのことからエージェント 1 つ 1 つの処理時間が増加するかもしれない。そこで，以降の実験では 1 台のマシンで*N*-Queens問題を解くこととした。

4．2　実行時間比較

今までのエージェント数を固定させたまま問題を解く方法とエージェント数をネットワークの状態に応じて変動させる方法とで比較した。

最初に，実験に使用したPC1~6 の各マシンが独立して *N*-Queens問題を*N*=16（分割レベル 3）で解いた。問題が解き終わったらエージェントの数を増やして同じ問題を解かせるということを各マシンで行い，単一マシン上でエージェントの数を増やした場合の実行時間の平均値を測定した。

次に，最大 6 台まで変動するPC環境でネットワークの状況を変化させることでエージェント数を変動させて実行時間を計測した。この実験ではマシン台数を徐々に増やしていくことでネットワークの状況を変化させた。これらの実行時間を比較したグラフは図 7 のようになった。この実験の結果，エージェント数を固定させた場合，エージェントの数を増やしていくほど実行時間は増えていくことが確認できた。また，マシン台数を増やしてエージェントを動的に増加させた場合，エージェントの数が増えるほど実行時間が短くなることが確認できた。

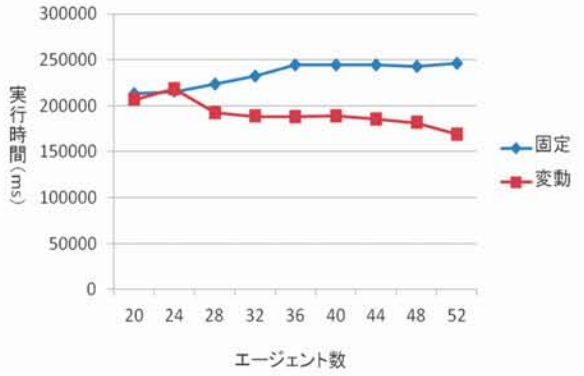

図 7　エージェント数固定と変動での実行時間の比較

4．3　1 台のマシンでのエージェント稼働数による影響

図 7 で示したように 1 台のマシンでエージェント数を増やしていく方法とマシン台数が増えるのに応じてエージェント数を増やす方法では，後者の方が実行時間が短くなるということが分かった。これはエージェントが活動できる領域が物理的に増えたことによりエージェントが分散され，大量のエージェントを取り扱えるようになったからだと考えられる。1 台のマシンで大量のエージェントを実行させる場合，どのような影響があるのかを調べるためにPC 1，PC4，PC6 を独立で稼働させ，実行するエージェント数を 4，8，…と増やしていきながら*N*-Queens問題を解き，その時の実行時間を比較することでエージェント数が増えた時の実行時間の変化を測定した。

図 8，図 9，図 10 のグラフから，PC1 ではエージェント数が 8 のときまで，PC4 ではエージェント数が 4 のときまで，PC6 ではエージェント数が 16 のときまではそれぞれの実行時間に大きな変化は見られなかった。しかし，この数以上のエージェントが生成されたときに実行時間が増えていることが各マシンで確認できた。これらのマシンのコア数は表 1 で記載されているようにそれぞれ 4 コア，2 コア，8 コアとなっている。このことから，エージェント数がコア数×2 を超えて稼働していた時に処理時間が増加していることがわかる。

つまり，1 台のマシンでのエージェント稼働数の限界はコア数の 2 倍（ハイパースレッディングによるもの）を基準として考えることができると考えられる。

また，1 台のマシンで大量のエージェントを実行させた場合，実行時間以外にもマシンのCPU使用率が増加していることが確認できた。このとき，文書作成や動画再生などの全く関係ない処理をすると動作が重くなったりアプリケーションの動作が停止してしまったりするといったことが発生した。マシン台数が増えたときも 1 台当

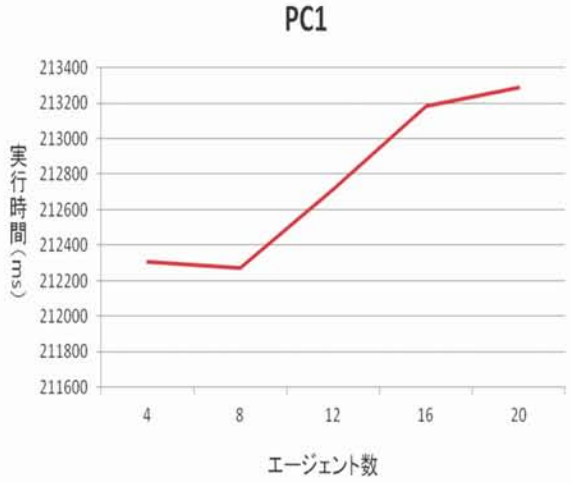

図 8　PC1 でのエージェント数と実行時間の推移グラフ

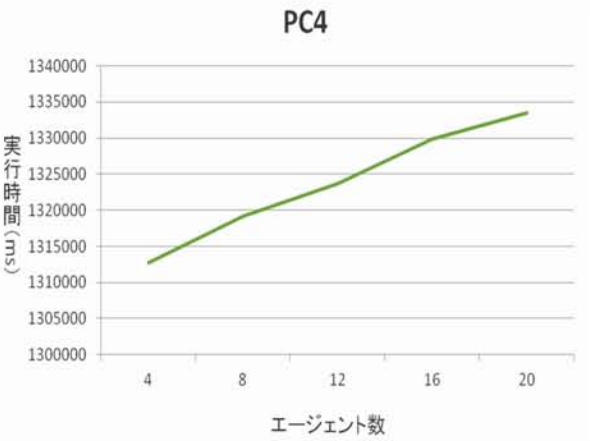

図 9　PC4 でのエージェント数と実行時間の推移グラフ

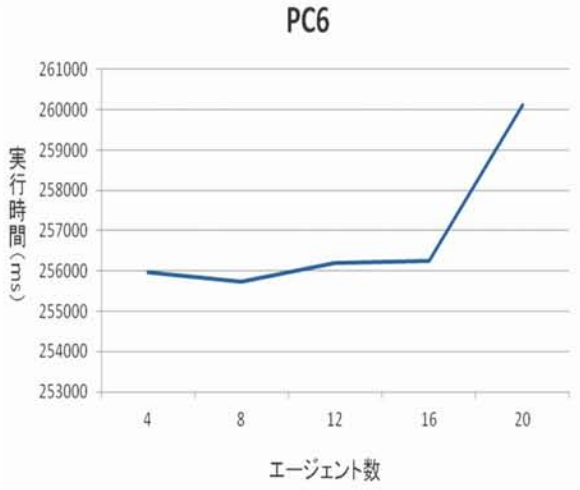

図 10　PC6 でのエージェント数と実行時間の推移グラフ

たりのエージェント数が分散されているので 1 台の時ほど影響はなかったがCPU使用率は増えていった。

5．まとめ

今回行った実験結果からマシンの増加，減少に応じてエージェント数も増加,減少することができたといえる。また，1 台のマシンで大量のエージェントを生成するとエージェント 1 つ 1 つの処理時間が長くなってしまって逆に実行時間が延びるということがわかった。しかし，4.1 で行った実験では常にマシン台数に合わせてエージェント数を抑える方法をとった場合逆に実行時間が延びるという結果が得られた。これは，エージェント単体のパフォーマンスが多少悪くなってもエージェントがより多くのタスクの処理を続けるのと，パフォーマンスを最大限に引き出すために手を付けるタスク数を絞って行うという違いから起こったことだと考えられる。

実行時間の比較から，マシン台数を増やしてエージェント数が増えた場合とマシン台数を増やしてエージェント数を固定させた場合，前者のほうが物理的に利用できるリソースが増えた分エージェントを大量に処理できるようになり実行時間が早くなったことがわかった。

結果として，マシン台数に応じてエージェントを変動させることによって余剰のリソースを有効に活用することができたといえる。

今後の課題として，AgentSphereを利用するにあたって，様々なエージェントがマシン間を行き来することになるが，現状ではJVMのメモリ使用率を用いた簡単なエージェント収容処理 [6]を行っているだけでCPU使用率などは考慮していない。ユーザが所有しているマシンでAgentSphereを立ち上げ，エージェントの処理しかしないといった場合はJVMのメモリ使用率を用いた方法だけでもいいかもしれないが，エージェントにタスクを渡したユーザが同一マシン上で他の作業をすることは容易に考えられる。また，ネットワークでつながれた他のユーザもAgentSphereでエージェントを動かす以外の作業をするかもしれない。そこで，各AgentSphereの初期設定としてAgentSphereに備えつけられている独自のシェル [7]からユーザに何%までCPUの使用を許可するか設定させ，エージェントを移動させる際にCPU使用率を参照した確認を追加するなど，エージェントを受け入れる際に何らかのアプローチをすることが必要になってくると考えている。

引用文献

1) 赤井雄樹，横内　貴，若尾一晃，甲斐宗徳，「強マイグレーションモバイルエージェントシステム AgentSphere の開発」，情報科学技術フォーラム 2009，B-011，Sep.2009
2) 鈴木幸祐，「AgentSphere における分散ハッシュテーブルを用いた情報共有機構」，成蹊大学理工学研究科理工学専攻修士論文，2012
3) 萩野谷一二，「NQueens 問題への新しいアプローチ（部分解合成法）について」，情報処理学会研究報告ゲーム情報学(GI), 2011-GI-26 巻 11 号, pp.1-8, 2011
4) 長塩征幸，「DHT に基づく検索機能を用いた AgentMonitor の改良」，成蹊大学理工学部情報科学科卒業論文，　2012
5) 山口大祐，「AgentSphere における AgentPool の実現と Master/Slave 型並列 API の作成」，成蹊大学理工学研究科理工学専攻修士論文，2011
6) 黒崎信清，「モバイルエージェントシステム AgentSphere の開発 －エージェント移動プロトコルの設計と実装－」，成蹊大学理工学部情報科学科卒業論文，2011
7) 大久保秀，「モバイルエージェントシステム AgentSphere の開発 －デバッグ補助機能を持つオブジェクト操作可能なシェルの開発－」，成蹊大学理工学部情報科学科卒業論文，　2011