B-001

モバイルエージェントシステム AgentSphere のための 透過的ファイルアクセス手法とエージェントスケジューラの試作
Design and Evaluation of a Transparent File Access Method and Agent Scheduler For Mobile Agent System AgentSphere

新倉　律† 高田　浪生‡ 甲斐　宗徳†
Norito Niikura Namiki Takata Munenori Kai

1. はじめに

近年 CPU の高速化は限界を迎えつつあり、それに伴って大規模な計算処理を行う際はマルチプロセス・マルチコアの計算資源による並列分散処理を行うことが主流になっている。ただしそれを行うためには並列分散処理に関する専門的な知識を持つ必要がある。

そこで筆者らは並列分散処理に対する専門知識を持たなくともその恩恵を受けられるようなシステムとして自律型並列分散処理プラットフォームである AgentSphere を開発している。

AgentSphere の開発コンセプトは以下の通りである。AgentSphere 同士がネットワークを構築し、そのネットワーク上でエージェントと呼ばれるプログラムが、必要に応じて移動しながら計算処理をすることで自律的な並列分散処理を行う。またエージェントは計算状態を記録した自己のバックアップも生成することができ、移動先のコンピュータが停止した際もバックアップの段階から復帰することによる高い耐故障性も兼ね備えている。さらに自律的に各マシンの性能を把握してそれに自動的に対応するように並列分散を行うことができ、ユーザはマシンの性能差を意識することなく並列分散処理を実行させることができる。

筆者らは自律的な並列分散処理をサポートするため、AgentSphere に対して様々な機能の研究開発を行ってきた。その中で、ファイルの取り扱いやエージェントに自律性を持たせるためのエージェント用スケジューラに関しても研究が行われてきた。

本論文では新たに実装されたエージェント向けの透過的なファイルアクセスインタフェースとメモリの使用率を考慮したエージェントスケジューラについての評価結果を報告する。

2. AgentSphere について

AgentSphere の説明にあたってその基本要素であるモバイルエージェントシステムについて述べる。

2.1 モバイルエージェントシステムについて

モバイルエージェントシステムとはネットワークを用いた並列分散処理を行うシステムの一つで、複数のコンピュータ間を状況に応じて自律的に移動しながら、与えられた計算処理を行うエージェントと呼ばれるプログラムを用いたシステムである。このシステムを使うことで処理させたい内容だけプログラムとしてエージェントに与え、それを実行するだけでエージェントが自律的に処理を行うため、ユーザはエージェントの移動やシステムの状況の変化を考慮することから解放される。よってアプリケーション開発において分散処理のためのプログラミングコストを減らすことができ、他の部分にそのコストを使うことができる。

このエージェントが実行されるコンピュータでは、モバイルエージェントシステムが実行可能な環境が整えられている必要がある。逆にその条件さえ満たしていれば、パソコンに限らずあらゆる媒体でこのシステムを利用できると考えている。

2.2 モビリティについて

モビリティ（コードモビリティ）とは、ネットワーク上でプログラムを異なるコンピュータに移動させ実行させる際に、どの程度情報を移動させるかという観点に基づいた指標である。モバイルエージェントシステムにおいてはどのモビリティを採用しているかによって、実際の実行における効率やプログラミングにおける制約が大きく変化する。

モビリティには遠隔実行、弱マイグレーション、強マイグレーションの三種類があり、遠隔実行はプログラムコードのみが移動可能なモビリティである。移動するデータが少なく、そのデータも加工することなく送信できるので実行が高速である。しかし移動先のコンピュータに移動前の実行状態を表すデータは送信されないため、初期状態からの実行しか行えない制約も存在する。

弱マイグレーションは、プログラムコードと大域的な変数が移動可能なモビリティである。こちらもプログラムの実行状態は最初からの開始となるが、変数の移動も可能であるためプログラムによってはメソッド単位での継続を行うこともできる。

強マイグレーションは弱マイグレーションで移動するものに加えてローカル変数やプログラムカウンタも移動可能なモビリティである。これらの情報を保持していることから移動先でも移動前の処理の途中から再開することが可能になっている。欠点として上記のデータをすべて持ち運ぶため、移動におけるコストが大きくなってしまう点、また Java における欠点としてプログラムカウンタの取得は通常の JVM では行えないことからそれを取得する手段を用意する必要がある。また中断した状態におけるローカル変数やオペランドスタックの保持を行うためにバイトコードの解析や JVM の改造を行う必要がある。AgentSphere においては Javaflow[1]というライブラリを使用することでこの問題を解決した。このライブラリはバイトコードの解析によって各種データの保持を可能としている。

2.3 AgentSphere とは

AgentSphere は、成蹊大学ソフトウェア研究室で開発されている自律型並列分散処理システムのプラットフォームであり、モバイルエージェント技術を用いている。特徴として強・弱マイグレーションの両方に対応しており、JVM

† 成蹊大学理工学研究科理工学専攻 Graduate School of Science and Technology, Seikei University
‡ 成蹊大学理工学部情報科学科 Department of Computer Information Science, Seikei University

には手を加えていない為、JVM が対応するすべての OS で動作することができる。

AgentSphere の開発コンセプトとしてバックアップによる耐故障性、ソフトウェアによるスペック差のあるコンピュータの柔軟な活用、ユーザに専門的な知識を求めない並列分散環境の提供がある。

AgentSphere そのものはモバイルエージェントを動作させるための仮想的な空間という位置づけであり、それ自身がモバイルエージェントというわけではない。

2.4 AgentSphere の現状

AgentSphere はモバイルエージェントを保持し、実行をさせるためのシステムであり、そのエージェントの活動を支援するためのいくつかのモジュールが内包されている。この論文で報告する機能はこのモジュールと連携して機能することから、今回の報告に関わるモジュールの概略をここで述べる。

●ネットワークモジュール

AgentSphere が稼働しているコンピュータのネットワーク機能を管理するためのモジュール。

各エージェントが各々勝手にネットワークを使用してしまうと処理が複雑になる為、このモジュールに一括管理を行わせている。また他の AgentSphere の生存確認や情報共有のための分散ハッシュテーブル(DHT)の管理も行っている。

●メッセージングモジュール

エージェントや各モジュール間でのメッセージングを総括するモジュール。

このモジュールはその AgentSphere 内でのメッセージングはもちろん、他の AgentSphere に存在する対象へのメッセージングも行うことができる。

3. 透過的ファイルアクセスインタフェース

現在の AgentSphere はファイル操作を行う際、エージェントが各々独自に行うような形となっており、ファイルアクセスを行う際は Java の標準ライブラリに存在するファイル API を使用する必要がある。しかし、JavaAPI はリモートアクセスの手段を提供していないため、そのファイルの存在を認識できるコンピュータでしかファイルのアクセスができないので、エージェントがそのコンピュータの AgentSphere に移動することを迫られる。またユーザはアクセスするファイルがどの AgentSphere のコンピュータに存在するか把握して、それをコード上に記述する必要があり、その点に関して透過性が低くなる。そして後述のアクセス集中という点において分散処理の効率が落ちてしまう。また、そのファイルを持つ AgentSphere に集中的な負荷がかかる他、その AgentSphere がダウンすることでそのファイルを利用するすべてのエージェントが活動不可能となってしまう。そして複数のエージェントから同時に同じファイルにアクセスされた際、それらを排他的に取り扱うことができない。

つまり現状の方法ではファイルの一貫性や並行透過性が低く、分散処理の効率や耐故障性がファイルに依存して低下してしまうことを意味する。

これらの原因により、ユーザが AgentSphere のエージェントに処理を記述する際、ファイルに関する透過性の低さがネックとなる。またそれを補うためユーザは並列分散的なコーディングを強いられてしまう問題につながる。これは AgentSphere のコンセプトにおいて避けるべき事柄である。

よってこれらの問題を解決する為に Java 単体で動作しエージェントに透過的なファイルアクセスを提供することができ、ファイルアクセスを分散させられるようなインタフェース及び制御機構が求められる。

そのため本研究におけるファイルアクセスインタフェースの開発では、AgentSphere におけるファイルアクセスに関する問題の解決とユーザへの透過性の提供を目的として、以下のような条件を制定した。

① エージェントがどの AgentSphere に移動しても、同一のファイルには同様にアクセス可能である。
② エージェントがファイルをオープンした状態でも AgentSphere 間を移動である
③ ユーザはエージェントに対して透過的にファイルのアクセス命令を記述できる。
④ 複数のエージェントのアクセスに対してそれを調停し、並行的に効率よいファイルアクセスを提供できる
⑤ JavaAPI と使用方法を似せることで独自の知識を要求しない。

これらの内①、③、④はそれぞれ分散システムにおける位置透過性、アクセス透過性、並行透過性と関連している。

3.1 実装方法

今回新たに開発したファイルアクセスインタフェースでは、まずファイルのオープンとクローズ、内容の読み書きを透過的に行うための仮想的なファイル操作の仕組みと API を作成した。また、エージェントが移動しながらの読み込みも達成するため、ファイル操作のインタフェースと実際のファイル操作部分の分離も行う。つまりこのファイルアクセスインタフェースはインタフェースと実ファイル操作部の２つによって構成される。またインタフェースとファイルシステムの連携に必要な通信は AgentSphere が提供しているメッセージングモジュールを、各 AgentSphere にどのファイルが存在するかの情報の共有はネットワークモジュールの DHT を使用する。また、ユーザに独特の知識を要求しない為に、今回のファイルアクセスインタフェースの API は JavaAPI のファイル操作メソッドと類似させる。

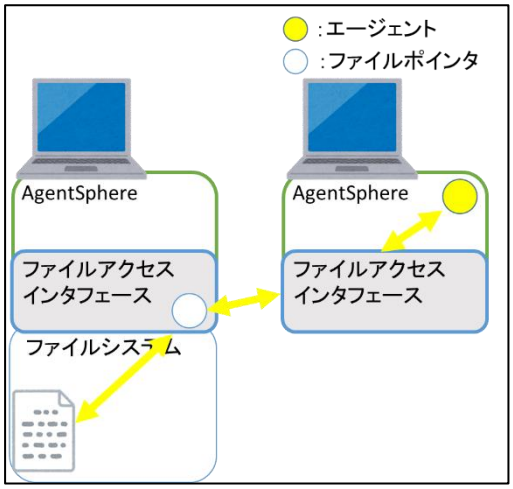

図１　ファイル操作の大まかな流れ

ファイル操作の流れを簡単に述べると、まずエージェントはファイルのオープンをその時活動している AgentSphere のファイルアクセスインタフェースに委譲する。その際にファイルアクセスインタフェースによってインタフェースオブジェクトが生成される。

そしてファイル読み書きの際はインタフェースオブジェクトが持つ読み書きのためのメソッドを使用することで、そのエージェントが活動している AgentSphere のファイルアクセスインタフェースに対してファイル操作の要求を送信する。

その要求を受けたファイルアクセスインタフェースは、その要求がローカルのファイルに対するものであれば、そのままファイルアクセスインタフェースが保持するファイルポインタを介して JavaAPI を使用しファイルの操作を行う。

対象が別のマシンのファイルであればその AgentSphere のファイルアクセスインタフェースに要求を送信し、そのマシンで行われた操作の結果を受け取る。

最後にこうして得られた結果をエージェントが持つインタフェースオブジェクトに送信することでエージェントは透過的なファイル操作を達成できる。

3.2 動作実験

動作実験として実装されたファイルアクセスインタフェースを使用しローカルファイルと遠隔ファイルの読み書きを行った。

まず使用した PC の詳細を示す。

	PC-A	PC-B
OS	Windows10	Windows10
CPU	Intel(R)Core(TM)m5-6Y57 CPU @ 1.10GHz 1.51 GHz	Intel(R)Core(TM)i5CPU M580@2.67GHz 2.6 GHz
RAM	8GB	8GB

今回は PC-A と PC-B を使用し、それらを LAN ケーブルで接続した。ローカルファイルの読み書きでは PC-A で活動するエージェントが PC-A のファイルにアクセス可能であるかの確認を、遠隔ファイルの読み書きは PC-B で活動するエージェントが PC-A のファイルにアクセス可能であるかの確認をそれぞれ行った。またこの動作がどちらの場合でもエージェントにとっては同等であることを確認するため、実験で使用したエージェントのソースコードはどちらも共通である。

読み込みを行うコードの実行内容はテスト用ファイルのロケーションを IP アドレスで示し、それを開いてシーケンシャルに 2 バイト読み込んだ後、オフセット 2 から 2 バイト読み込む。しかし読み込みテスト用ファイルは 3 バイトしか内容がないためオフセット 2 からの読み込みは 1 バイトだけの読み込みとなることが期待される。それぞれ読み込んだ後、読み込んだバイト数と内容が出力される。

書き込みを行うコードの実行内容はあらかじめコードに定められたバイナリデータ 01~08 を書き込みテスト用ファイルのオフセット 0 及びオフセット 8192 にそれぞれ書き込む。書き込みが完了した際に書き込んだバイト数を出力する。

各コンピュータには固定 IP アドレスを振り、PC-A は 192.168.0.1、PC-B には 192.168.0.2 がそれぞれ割り当てられている。ローカルでのアクセスの結果を図 2、図 3 に、遠隔アクセスの結果を図 4、図 5 に示す。ここで赤枠がエージェントの出力である。

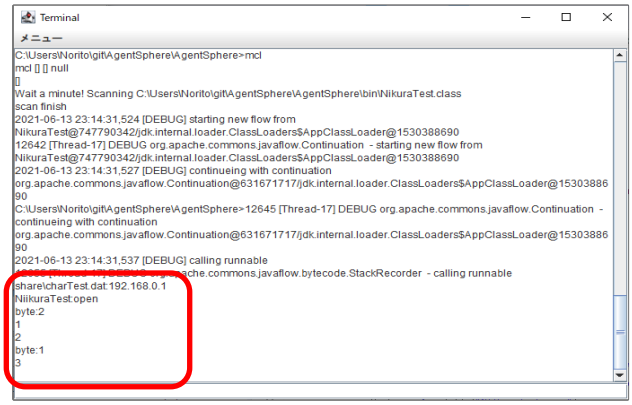

図 2　ローカルファイルの読み込み結果

図 3　ローカルファイルの書き込み結果

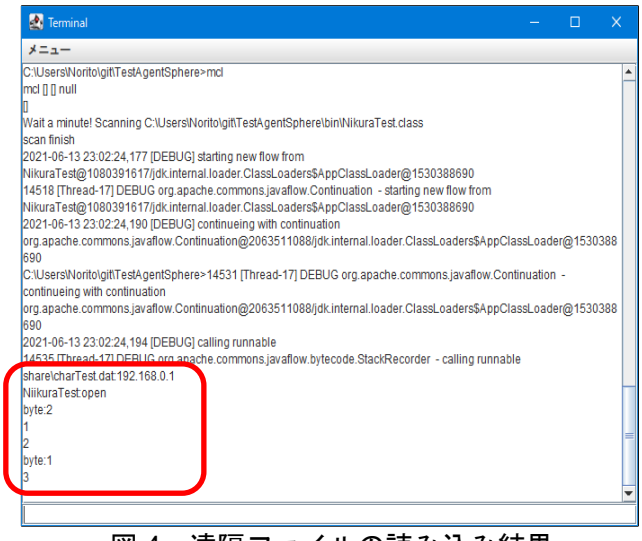

図 4　遠隔ファイルの読み込み結果

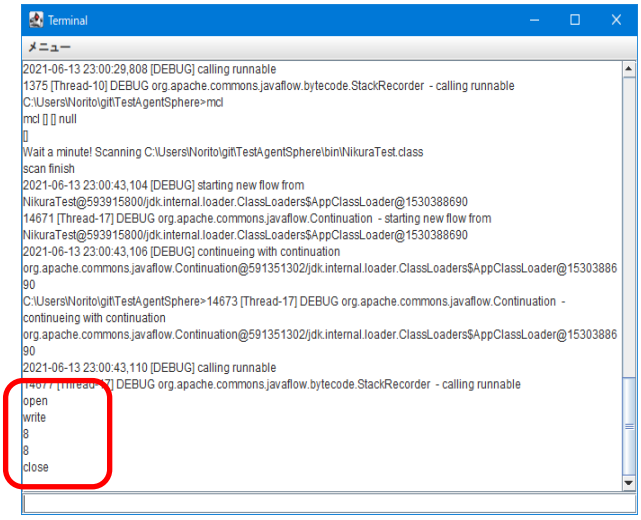

図 5　遠隔ファイルの書き込み結果

実行結果からローカル、遠隔問わず前述のプログラムの解説で述べた通りの結果が得られていることが分かる。

3.3 性能評価

これらの結果から今回開発した透過的ファイルアクセスインタフェースは今回期待したファイルに関する透過性の提供の役割を備えていると言える。

ファイルアクセスインタフェースに対する性能評価として、今回開発したファイルアクセスインタフェースを使用したファイルアクセスと、それをせずにファイルアクセスを行う場合で、ユーザにかかるコーディング上で考慮する必要のある要素を比較することで定性的に評価する。

今回開発したファイルアクセスインタフェースでユーザが意識しなければならない点については、アクセスインタフェースのクラス構成及びメソッドを JavaAPI のそれと同じにしたことから、通常のファイルアクセスと同じであり、実ファイルの場所は今回のシステムが抽象化しているため、Java のコーディングが行えるのであれば懸念事項はないと考えられる。

逆に今回開発したファイルアクセスインタフェースを使用せずに既存の AgentSphere に存在する機能だけでファイルアクセスを行う際にユーザが考慮しないといけない点として以下の物が挙げられる。

まずファイルがどのコンピュータに存在するか知らなければならず、そのコンピュータが動作する AgentSphere に対して明示的にエージェントを移動させなければならない。

さらに移動した際にファイル内容を扱うのが 1 回だけであれば JavaAPI のファイル操作用クラスで読み込み、書き込みすればよいが、エージェントの特色である移動を行いながらファイルを読み込むことを考える場合、その AgentSphere にユーザがそのファイル専用のサーバのようなオブジェクトを設置する必要が出てしまう。

このサーバのようなシステムをユーザがファイル操作のためだけに管理しなければならないことは大きな負担でもあり、システム的な観点からもファイルに対して様々なエージェントがファイルのためだけにスレッドを作成していくことは非効率である。

またファイルアクセス時の IO 例外やファイルの終了などを通達する仕組みもユーザが考えなければならない。

今回のファイルアクセスインタフェースはこれらの懸念事項を委譲されユーザから見えなくし、AgentSphere 全体でファイルを透過的に扱えるようにしたことから、AgentSphere における開発をより有用なものにしたと言える。

4. エージェントスケジューラ

AgentSphere はコンセプトとして「そのネットワーク上でエージェントと呼ばれるプログラムが、必要に応じて移動しながら計算処理をすることで自律的な並列分散処理を行う」ことの実装を目的として研究が進められている。

しかし現状の AgentSphere ではエージェントを移動（強マイグレーション）させる際には「指定した PC に指定したエージェントを起動させるコマンド入力」、「エージェントのプログラム内に IP アドレスで指定した PC へと移動させる命令処理」を記述しておく必要があり、自動で移動させる機能は存在しない。

そのため、AgentSphere がより目的に沿った発展をするにはエージェントを自動で振り分けるなどといったエージェントスケジューラ機能が必須であると考えられる。

また振り分け時にエージェントが実行できない、または実行時間が大幅に遅くなる等といった問題を避けるためにある程度の PC 状態を加味したエージェントスケジューラ機能が必要である。

今回の研究では AgentSphere が動作するコンピュータの CPU 使用率やメモリの使用量に着目して、エージェントの起動時にエージェントを適した AgentSphere に振り分けるスケジューラの試作を行った。

4.1 実装方法

今回作成したスケジューラは、ネットワークモジュールの DHT と連携して、各 AgentSphere の動作している環境の情報やエージェントの実行時にかかるメモリ使用量を測定しながらエージェントの分散を行う。

そのため以下の機能を実装した

① エージェント実行時の使用メモリと計算時間の計測機能
② 自分が動作している環境の性能計測
③ エージェント実行時の最適な AgentSphere の選定

それぞれの機能の実装方法を述べる。

①ではエージェントの実行時に必要なメモリ量及び実行時間を計測するが、今回は一度エージェントを実行させ、そのデータを収集する手法をとった。こうして得られたデータはネットワークモジュールの DHT に格納することで他の AgentSphere と情報を共有する。

メモリ量を計測するための他の手法として構文解析などの解析プログラムによる方法が挙げられるが、エージェント解析ではエージェントごとに毎回起動させる必要がでてきてしまい場合によっては大きな負荷がかかってしまうこと、ユーザやファイルの入力によっては動的にメモリ使用量が変わってしまうことが考えられる。

次に②を実装するにあたって、まずその環境のどの性能を評価値とするかを決めた。基本的に処理速度は PC の CPU の性能によって決定されるため、まずは CPU のクロ

ックスピードを評価指標に加えた。クロックスピードであれば OS から情報を取得でき、かつ演算処理性能に密接にかかわっているためである。今回の研究における調査では Java において OS の情報を直接取得するようなライブラリ、関数は発見できなかったため今回は Runtime クラスを用いてコマンドプロンプトにコマンドを打ち込むことで、結果を取得する形とした。

またそのほかに処理性能に影響を及ぼすものとして主に CPU 使用率、空きメモリ容量が考えられたためこれらのパラメータがどの程度プログラムの処理時間に影響するかの簡単なテストを行った。テストプログラムに行列の足し算を行うだけのシンプルなものを用意し、「負荷をかけずに行った場合」「CPU のみに負荷をかけた場合」「メモリのみに負荷をかけた場合」の 3 パターンを比較した。この時の負荷のかけ方については CPU に負荷をかける場合には HeavyLoad[2]というアプリケーションを使用して行った。また JVM のメモリに負荷をかける手法として可変長配列にデータを追加し続けるようなスレッドをバックグラウンドで実行させることで負荷をかけた。

負荷テストの結果を以下に示す。

表 1.　負荷テスト結果

	平均CPU使用率	平均空きメモリ[Byte]	実行時間[ms]
負荷なし	0.411919558	98445824	9403
CPUに負荷	0.95	105804288	13094
メモリに負荷	0.444572836	19994161	8751

結果は表 1 のようになった。このデータにおいて、CPU 使用率、空きメモリ容量はともに計算開始から 0.5 秒間隔で計測したものである。

上記の結果からメモリに負荷をかけた場合、通常起動と遜色ない実行時間となり、CPU に負荷をかけた場合は明らかに実行時間が遅くなった。しかし、メモリに関しては負荷をかけすぎた際に OutOfMemory エラーを出してしまいプログラム自体が実行できなかった例があった。したがって CPU 使用率は実行時間に影響を及ぼすが、起動する際には影響しない。JVM 空きメモリ容量は実行時間に影響はないが、起動には影響することがわかる。このことから PC の性能評価する際にはまず空きメモリが充分であるかどうかの評価をし、次に CPU 使用率を判定することでエージェントを割り振る際により効率のいい振り分け方ができるのではないかと考えたため、今回のエージェントスケジューラでは主にメモリ状況を重視した。今回の情報の内メモリ量及び CPU 使用率は常に変動するため、リアルタイムで計測を行う。こうして得られたクロックスピード、使用メモリ量、CPU 使用率も①と同じく DHT で共有する。

最後に③の実装では①、②の仕組みで DHT に共有されたエージェント実行に関するデータを使用し、起動しようとするエージェントの実行に適する AgentSphere の選定を行う。

具体的な動作としては、エージェント起動の際にまずそのエージェントに関して①で計測できるデータが存在するかどうか DHT に問い合わせを行う。この時データが存在しない場合はそのコンピュータでそのエージェントを起動する。データが存在する場合はその使用メモリ量を確認し、エージェントを起動しようとする AgentSphere が動作している JVM が十分なメモリ量を確保していればそのエージェントをその場で実行する。その AgentSphere で起動できない場合、すなわち実行環境の JVM のメモリ量が足りない場合は DHT に他の AgentSphere の環境の情報を問い合わせ、メモリ量が十分にある AgentSphere の中から最も性能の高い環境で動作している AgentSphere にエージェントを移動させ、そこで実行させる。

4.2 動作実験

動作実験として PC-A と PC-B を使用し、開発したエージェントスケジューラが正しく動作しているか確認した。使用した PC の詳細を示す。

	PC-A	PC-B
OS	Windows10	Windows10
CPU	Intel ® Core ™ i3-8130U CPU@2.20GHz 2.21GHz	Intel ® Core ™ i7-6650 CPU@2.20GHz 2.21GHz
RAM	4GB	16GB

またプログラム内で取得した PC のクロック数はどちらも 2208 であった。

実行するプログラムはメモリを大きく取得しつつ計算の時間もかかるエージェントの表現のため 3000*3000 の行列の演算を 1000 回繰り返すエージェントを作成した。まず 1 度実行して各種データをスケジューラに記憶させた後それを複数回実行させることでメモリの枯渇が発生した際にエージェントの割り振りが発生するかどうか検証した。

動作結果を以下に示す。

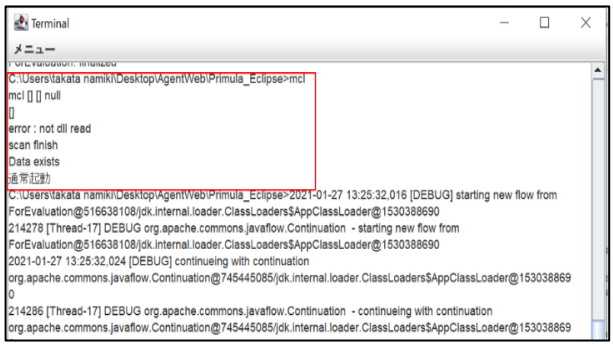

図 6　PC1 で実行できる際の結果

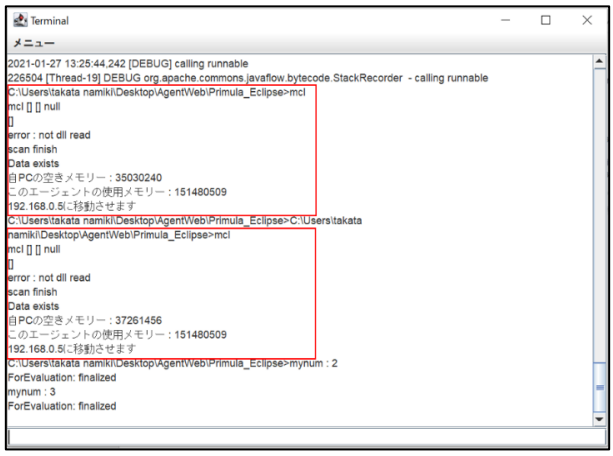

図 7　PC2 に移動させることを判定した結果

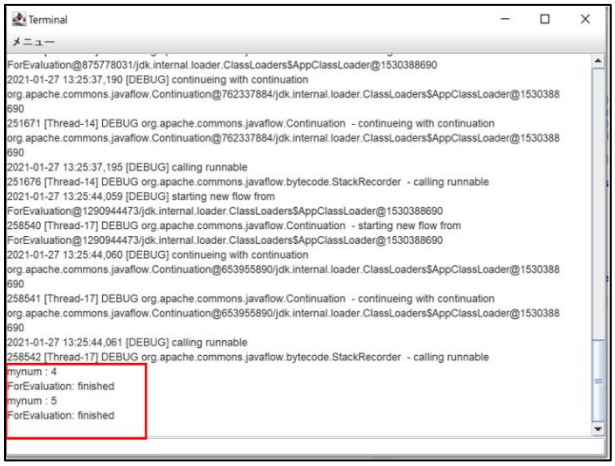

図 8　エージェントが送られて来た PC2

図 6 赤枠内より、「Data exists」と出力されており、これはそのエージェントの実行時データがプロファイルされていることを示している。

またその次行の出力で「通常起動」と出力されていることから自 PC で起動できると判断し待機処理や移動処理を挟まずに起動していることが確認できた。また図 6 中のその後の出力にエラーが見当たらないことから正常に機能していることが確認できた。

図 7 赤枠内より、「自 PC の空きメモリー：163780656」「このエージェントの使用メモリー：151480509」と出力されていることから自 PC にメモリが不足しそうな際に自 PC で起動していないことが確認できる。その後「192.168.0.5(PC-B)に移動させます」と出力された。

図 8 赤枠内より評価用プログラムの終了処理が呼び出されていることから自 PC で起動できず、他 PC で起動できる際に移動させる処理が正しく動作していることが確認できた。

4.3 性能評価

4.2 の実行結果から今回開発したスケジューラは起動するエージェントを各 AgentSphere の動作するそれぞれの JVM の状況に合わせて適した AgentSphere に移動させることができた。

スケジューラに対する性能評価として、ユーザが手動で適切な AgentSphere にエージェントを移動させる命令を記述する場合と今回のスケジューラを利用する場合でユーザが考慮する必要がある点を比較することで定性的な評価を行う。

まずスケジューラを使用する場合、ユーザはエージェントのコードに明示的にエージェントを移動させるコードを記述する必要はなく、それに伴いどの AgentSphere にエージェントを移動させるべきか、現在の AgentSphere で使用できるメモリ量は十分かどうかなど、実行時の AgentSphere の状況を考える必要は無い。

スケジューラを使用せずに同じ機能をユーザが実装する場合、エージェントの実行開始時にすべての AgentSphere に対してその AgentSphere の性能を計測し、生存信号と共にその情報を本体のエージェントに送信するような監視役のエージェントを作成する必要がある。またエージェントにもこの情報を受け取り、処理するための仕組みを実装しなければならない。さらに、この手法を採用するエージェントが増えるにつれ監視役エージェントも増加し、AgentSphere 全体のリソースを削ってしまうことにもつながる。

以上の点から今回のスケジューラはユーザが並列分散処理に対して考慮しなければならない点を請け負って、AgentSphere の自律並列分散処理能力を高めることができたと言える。

5. おわりに

本研究によって AgentSphere に透過的なファイルアクセス手段とメモリ量を考慮したスケジューラが実装された。

まずファイルアクセスインタフェースにより今までの AgentSphere における開発で発生していたファイルの場所を知らないといけない問題や、エージェントの活動場所によるファイルアクセスの制限、ファイルオープン中にエージェントが移動できないなどの制限を取り払うことができた。

次にメモリ量を考慮したエージェントスケジューラにより、メモリを原因とする実行が失敗したりするような状況を回避できるようになり、AgentSphere の自律性を高めることができた。

今後の課題としては、今回開発したファイルアクセスインタフェースもスケジューラもまだ開発の余地があるためそれをさらに深めていくことが挙げられる。

ファイルアクセスインタフェースであれば AgentSphere のユースケースに合った機能を追加することで、既存のネットワークファイルシステムにはない独自性を実現でき、AgentSphere での開発に貢献できると考えられる。

スケジューラは現在エージェントの実行時に移動を行わせているだけにとどまっているため、エージェントの実行中でも最適な PC があれば自律的にそちらに移動できるような仕組みが求められている。

参考文献

[1] “Javaflow”,https://commons.apache.org/sandbox/commons-javaflow/, 2021.6 現在参照可
[2] “HeavyLoad”,https://www.jam-software.com/heavyload, 2021.6 現在参照可