成蹊大学理工学研究報告
J. Fac. Sci.Tech., Seikei Univ.
Vol.49 No.1 (2012)pp.13-18

AgentSphere における AgentPool の実現と Master/Slave 型並列 API の作成

山口　大祐[*1]，甲斐　宗徳[*2]

Implementing AgentPool and API for Master/Slave Parallel Processing on AgentSphere

Daisuke YAMAGUCHI[*1], Munenori KAI[*2]

ABSTRACT : AgentSphere has been developed as an autonomous application framework for the heterogeneous parallel and distributed processing environment. AgentSphere runs on each machine, and provides a work field to mobile agents. As both AgentSpere and mobile agent are written in Java, there is a multithreading overhead problem in the case that application with very high parallelism is distributed onto fewer physical processing elements. So JavaVM has introduced Concurrent Utilities and ThreadPool for this problem, but they can work on only one JavaVM. In our research, we have developed AgentPool, which is similar to ThreadPool but available on multiple JavaVMs. We also report results of parallel processing of solving NQueens problem to show the performance of Agentpool.

Keywords : Multithreading overhead, Java Concurrent Utilities, parallel distributed processing

(Received March 31, 2012)

1．はじめに

過去数年にわたり，著者らは並列分散環境のための自律型アプリケーションフレームワーク AgentSphere[1][2][3][4][5]を開発してきた。AgentSphere は，モバイルエージェントに活動の場を提供するもので，1台の PC につき1つ動かすことを前提としている。AgentSphere が動いている PC が複数接続されたネットワーク上では，各 PC が対等な関係にあり，その台数は随時増加減少してよいものとしている。どのマシンの AgentSphere に行ってもモバイルエージェントを動作可能にするため，ヘテロジニアスなマシン環境に有効な Java を用いて開発している。

AgentSphere の利用目的として，複数台のマシンがつながれた環境を並列処理による高速化に利用することが考えられる。特に，並列性の高い対象である大量の並列タスクが生成される場合，これらのタスクの処理をエージェントがどのように担当するかについて考える必要がある。同様の議論は単一 JavaVM（以降 JVM）におけるマルチスレッドでも存在する。そのため，Java では単一 JVM 用に Java 1.5 より JSR166 である Concurrency Utilities [6][7][8]が取り込まれている。その中の ThreadPool の機能は，起動時にスレッドを少数生成し，投入された大量のタスクを少数のスレッドが実行していく方法である。

本稿では，単一 JVM で使うことができる ThreadPool と同様の働きを持つ AgentPool を実現する。ただし，AgentSphere では複数の AgentSphere にエージェントが移動して処理できるので単一 JVM で使える ThreadPool よりも多くのコアを利用することができるため，並列処理性能を上げることが期待できる。そして，この AgentPool を利用して大量の並列タスクを生成するアプリケーションの例として，NQueens 問題の並列探索の結果も報告する。

2．並列処理における問題

並列処理を行うにあたり，並列実行可能な箇所を見出すことは非常に重要である。しかし，並列性が高い対象を物理実行環境すなわちコア数に合わせず高並列実行すると，逐次実行時よりも時間がかかってしまう（4.1.節で

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

検証する）ことがある。それは，多数のスレッドを利用することによるスレッド生成コストや，スレッドの切り替えコストが増加するためである。

2.1. Concurrency Utilities

Concurrency Utilities とは，Java 1.5 より取り込まれた java.util.concurrent パッケージのことである。これは，ThreadPool などを備えたハイパフォーマンススレッド利用の拡張フレームワークである。この中の ThreadPool 機能は，多数の並列実行タスクに対してスレッド生成コストを減らし，ユーザの並列実行を助ける。以降はパッケージ中の，Callable，Future，ExecutorService という３つのインタフェースに注目して説明する。

2.1.1. Callable

Concurrency Utilities では，Callable<V>インタフェースが存在する。これを継承したオブジェクトに，ユーザは V call()というメソッドを定義するだけで，関数の返り値である V を並列実行後に取り出すことが可能になる。

2.1.2. Future

Future インタフェースを用いたオブジェクトは，並列実行後の結果を取り出すためのオブジェクトである。具体的には，Callable<V>を継承したクラスを実行するときに作成され，このインタフェースを継承したクラスオブジェクトの，V get()を呼び出すことで V call()の結果 V を取り出すことができる。この V get()の中で，対象の計算が終了するまで待つ処理が行われることにより，ユーザは同期処理を意識することなく結果Vの取得が可能になる。結果としてユーザは，Future が継承されたクラスから V get()を呼び出すだけで，並列実行後の結果を得ることができる。

2.1.3. ExecutorService

ExecutorService インタフェースを用いたオブジェクトは，渡された Runnable や Callable タスクを処理するオブジェクトである。具体的には，Future<T> submit(…)を呼び出すことで実行したいオブジェクトを渡し，実行結果を取り出すための Future<T>を生成して返す。ユーザは，結果が必要になった段階で，Future を通して結果へアクセスする。この時 ExecutorService では，submit(…)メソッドを通してタスクを受け取った段階でキューイングを行い，生成済みの WorkerThread がタスクを取得するか，新しい WorkerThread を生成してタスクの実行を行う。また shutdown()により，新しい Callable の受け取りを禁止し終了処理を行う。

ユーザは，Callable を継承した並列実行オブジェクトを作成し，ExecutorService へ投入するだけで並列実行可能になる。

2.2. NQueens 問題

NQueens 問題とは，NxN 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，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[10]が考案されている探索問題である。

本稿では，並列性の高い問題として NQueens 問題を選択し，Master で幅優先探索による多数の並列タスクの生成を行った後に，タスクごとに深さ優先探索で解の探索を行うことで，多数の並列タスクを実行するプログラムを作成した。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）によって表 1.のように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。また，このとき各タスクのサイズを不均一にするため，配置途中の段階で配置できないことがわかる箇所については探索を中断し，処理の削減を行っている。

表 1. NQueens 問題の分割レベルとタスク数の変化

分割レベル	N=15	N=16
4	13980	19688
5	89428	141812
6	463038	838816
7	1897702	3998456

図 1.は ThreadPool を利用して作成した NQueens 問題のプログラム[9]の一部である。Executors は ExecutorService などを生成するファクトリメソッドが定義されているクラスである。

```
LinkedList<NQueenSlaveTask> taskList;
ExecutorService exeserv =
  Executors.newFixedThreadPool(nThreads);
LinkedList<Future<Integer>> futureList;
// start tasks
for(NQueenSlaveTask task : taskList){
        futureList.add(exeserv.submit(task));
}
// result to get from futures
for(Future<Integer> future : futureList){
        ans += future.get();
}

// service shutdown
exeserv.shutdown();
```

図 1. ThreadPool の利用例(NQueens 問題)

3. AgentPool

AgentSphere というネットワークでつながれた複数台の JVM を利用できる環境で,多数の並列タスクに対して,性能低下を引き起こすこと無く実行できるようにする。ThreadPool 記述方法として Concurrency Utilities を参考に記述方法をそろえることで,性能低下への対策アプローチになると共にユーザの習得コストを減らすことを目的にする。

AgentSphere では,AgentSphere を起動しネットワークに参加するマシンで実行されるプログラムを,Agent という形で記述することを求めている。一方,Concurrency Utilities では 2 章で述べたように,Callable<T>を継承したオブジェクトを受け取り,各 WorkerThread が実行を行う。そこで,Concurrency Utilities の WorkerThread を Agent へ置き換え,AgentSphere ネットワーク上で多数並列タスクの並列実行をオーバヘッド少なく行うのが AgentPool である。

AgentPool では,タスクを実行するにあたってエージェントがネットワークを移動し,その先でタスクを実行した後に,起動マシンへ戻り実行結果を届ける必要がある。この機構を表したものが図 2.であり,以降で動作を説明する。

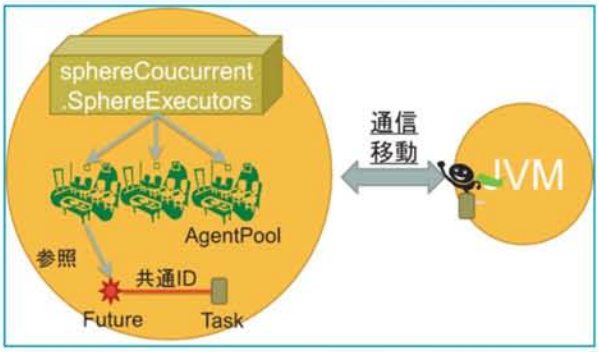

図 2. AgentPool の設計

まず,AgentPool をワーカエージェントの数を設定して起動することで,ワーカエージェントも同時に生成する。

次に,AgentPool はタスクが投入されたらそれらをキューイングしていく。ワーカエージェントはタスクが投入されたらそれを取り出し,ネットワークでつながれたマシンへ移動しそこでタスクの実行を行う。また,ネットワーク通信回数を減らすために,多数タスクが存在する場合には複数タスクを取得しそれらのタスクをまとめて実行してくる。

実行後戻ったのちに結果を設定するには,起動したマシン,ワーカエージェントの所属 AgentPool,実行したタスク,それぞれの識別が行えなければならない。AgentSphere では AgentSphere 毎に識別可能な固有の ID を用意しているので,起動したマシンの識別についてはその機構を利用する。所属 AgentPool の識別には,各起動したマシンにおいて AgentPool ごとに java.util.UUID を用いて固有識別子を生成し,固有識別子と AgentPool のオブジェクトをマッピングしたデータをマシンごとに一つ生成する。他のマシンで実行完了し帰ってきたエージェントはこれを用いて所属 AgentPool のオブジェクトを見つけ出す。実行したタスクの結果を届ける方法としては,所属 AgentPool を選別したのと同様に,AgentPool 毎にタスクと Future へ共通の固有識別子を生成し,AgentPool 内で固有識別子と Future へのマッピングを行い,それを利用して届ける。

また,各プールの終了である shutdown()が呼ばれた段階で,各 AgentPool 内の Future と Task へのマッピングを破棄することで,メモリリークを防ぐ。

記述方式を Concurrency Utilities と揃えることで,図 1 のプログラムからは図 3.のように,変更点は利用クラスの 1 点だけである。

```
SphereExecutorService exeserv =
        SphereExecutors.newAgentPool(nAgents);
```

図 3.　Concurrency Utilities からの変更点
(NQueens 問題)

4．実行時間比較

AgentPool の機能を確認するとともに，性能の評価を行う。

4.1.　実行形態による実行時間の変化

並列実行の実行時間の比較を行うにあたって，実行するエージェントの数とそれに伴う実行形態の変化によってどのように実行時間が変化するのかを確認した。確認したのは以下の 4 パターンである。

4.1.1.　SingleAgentSphere-ManyAgent (SingleAS-MA)

1 つの AgentSphere 内で 1 タスク 1 エージェント生成をすることで，莫大な並列実行を同時に行う方式である。

4.1.2.　SingleAgentSphere-AgentPool (SingleAS-AP)

1 つの AgentSphere 内で AgentPool を 1 つ生成し，本実験では AgentSphere から見えるマシンのコア数分のエージェントを生成して並列実行を行う。これは，動作的にも ThreadPool とほぼ同様な動きである。

4.1.3.　MultiAgentSphere-ManyAgent (MultiAS-MA)

複数の AgentSphere を用いて 1 タスク 1 エージェント生成をすることで，莫大な並列実行を行う方式である。

4.1.4.　MultiAgentSphere-AgentPool (MultiAS-AP)

複数の AgentSphere を用いる実行で 1 つの AgentPool を生成して実行を行う。本実験では「AgentSphere から見えるマシンのコア数×マシン数」のエージェントを生成して並列実行を行う。

これらのテストを行う環境として，Intel Core i3 (4 コア) メモリ 16GB のマシン 2 台で実行を行い，各 5 回の平均値をプロットした結果が図 4.である。

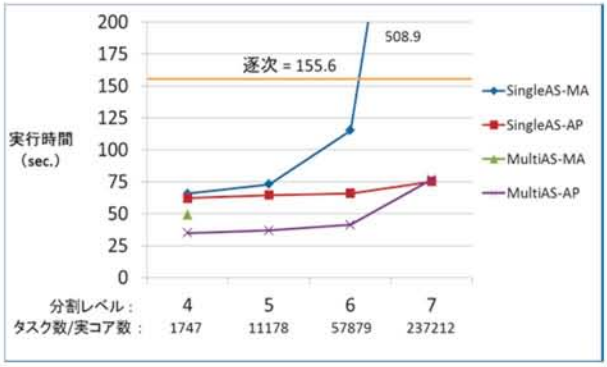

図 4.　実行形態による実行時間の変化

まず，SingleAS-MA の場合は分割レベルが上がるにつれて実行時間が延び，最終的には逐次での実行時間に対して 3 倍超の時間になってしまっている。これは，スレッドの生成コスト及びスレッドの切り替えコストの増加により実行時間が増えてしまったためである。

SingleAS-AP の場合には，タスク分割増によるオーバヘッドで実行時間の増加がみられるものの，SM に比べればほぼ一定の実行時間で実行が可能になっている。これは，ThreadPool とほぼ同様な動作になるため ThreadPool が目指す実行時間の削減効果が得られた結果だと考察する。

MultiAS-MA の場合においては，多数の細かなエージェントの移動による通信，およびそれに伴うメモリオーバフローにより分割レベル 4 以降のデータを取得することができなかった。

MultiAS-AP の場合においては，2 台のマシンで並列実行することによる SingleAS-AP からさらに実行時間が削減することができた。さらにタスク数が増えたとしても SingleAS-MA のような実行時間の増加にはつながらずに実行できた。分割レベル 7 の実行時間が SingleAS-AP とほぼ等しい実行時間になったことについては，今回のプログラムがタスク数に比例してデータサイズが増えるプログラムであり，データ通信の増加によって実行時間が増えてしまった結果である。

これらにより，適切にタスクサイズを設定した上で AgentPool を利用することにより，処理の高速化が期待できる。

4.2.　実行マシン数による変化

AgentSphere では，接続マシン数は特に制限されず随時増加減少させることができる。そのため，並列実行を行うマシン台数に応じて，どのように実行時間がスケールするか確認した。図 5. は同一ハードウェア構成の実行マシン台数が 1, 2, 3, 4, 8, 16, 64 の時，マシン 1 台の時の実

行時間を 1 とした時の高速化率(1 台の実行時間／n 台の実行時間)が示してある。実行環境は Intel Core2 Duo メモリ 2GB のマシンである。実行したプログラムは NQueenss 問題の N=16，分割レベル 4 のプログラムである。

図 5. の結果から，マシン台数に比例して並列化効果が出ていることがわかる。またその効果は，マシン台数 2 台では 95%，64 台でも 85%と高い効果が出ていることを確認した。

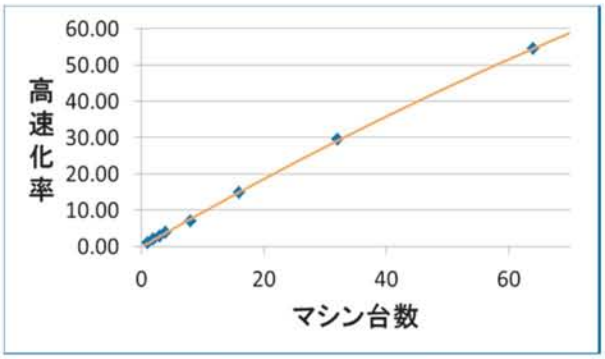

図 5. 実行マシン台数と相対速度の変化

4. 3. 実行マシン種類による実行時間の変化

AgentSphere では，マシン性能にばらつきのあるヘテロジニアスな環境での動作を想定している。そのため，4.1. 同様 NQueens 問題のプログラム(N=15，分割レベル 4)を用いて以下のような実験を行った。

① 1 台のマシンにおいて N=15，分割レベル 4 の問題を AgentPool での処理性能に差のあるマシンを 3 種類用意する。各マシン単体での実行時間は表 2 のとおりである。

② 基準マシンを，1, 2, 3 台と増やしていき，実行時間の減少を確認する。

③ 基準マシンに比べ単一での実行時間の遅いマシンを追加し，実行時間の変化を確認する。

これらの実験結果のグラフが図 6.である。

マシン台数を増やしても，タスク数を均一に割り振る場合において，相対的に遅いマシンを増やした場合には実行時間が延びてしまった。これは，このプログラムでは実行マシンの性能差を比較に入れていないため，実行マシン選定の際に一律に同僚のタスク数を持った実行エージェントを送り込んでしまうためである。そのため，均一の場合の実行時間は「最低性能マシンでの実行時間/全マシン台数=全体の実行時間」とほぼ等しい時間になっている。これを改善するために，マシン性能によってタスクを割り振る数を変えたものが性能依存のデータである。低性能のマシンをマシン群に含めても実行時間の増加を抑え，実行時間の短縮につなげることができた場合もある。

表 2. AgentPool による単体実行時間

マシン	単体実行時間 (sec.)
基準マシン	62.25
低性能1	246.96
低性能2	161.84

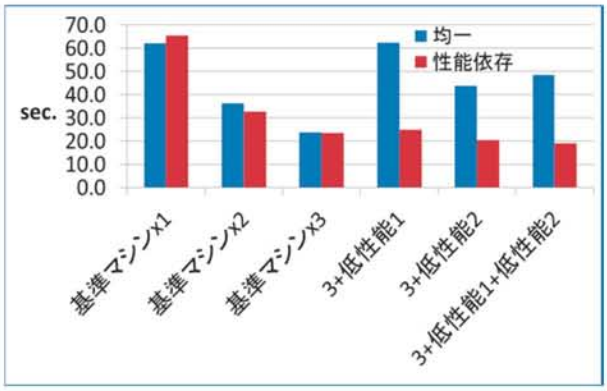

図 6. マシン群の構成による実行時間の変化

これらの実験結果から，AgentSphere による複数台つながった環境での並列実行の有用性，および更なる性能向上のためには，ネットワークの通信コストやマシン性能に応じた，タスクとエージェントの効率的な割り当て機構の必要性が確認された。

5．まとめ

本稿では，AgentSphere による並列処理を行うとき AgentPool を用いることで性能を引き出すと共に，容易に並列処理が行えることを確認した。

今後は，Concurrency Utilities にある別の実行方式や，タスクサイズおよびマシン割り当ての作成による性能向上が必要だと考える。また，AgentPool を用いた探索アプリケーションにおいて，他研究との性能比較をしていきたい。

謝辞

本研究は科研費（基盤研究(C)21500041）の助成を受けたものであることをここに記し，謝意を表します。

参考文献

[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」,FIT2009, B-011, 2009,9

[2] 山口 大祐・市川 顕・白谷 浩次郎・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 におけるエージェントの活動管理」,FIT2010,B-003,2010,9

[3] 山本 卓也・山口 大祐・甲斐 宗徳「モバイルエージェントシステム AgentSphere における通信プロトコルの開発」FIT2011,B-031,2011,9

[4] 山口 大祐・赤井 雄樹・甲斐 宗徳「JavaVM 上での非手続オブジェクト転送を可能とする直列化方式の構築」,
FIT2011,B-032,2011,9

[5] Yamaguchi,D. Akai,Y. Kai,M. :
"Construction of Serializing Method for Non-procedural Object Transfer between JavaVMs"
Pacific Rim Conference 2011, pp.262-267

[6] Lea, D.: 「The java.util.concurrent Synchronizer Framework」
http://gee.cs.oswego.edu/dl/papers/aqs.pdf

[7] Lea, D.: 「Concurrency JSR-166 Interest Site」
http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

[8] JDK6 並行処理ユーティリティー
http://java.sun.com/javase/ja/6/docs/ja/technotes/guides/concurrency/index.html

[9] Brian Goetz, Jshua Bloch, Doug Lea 岩谷 宏（訳）「Java 並行処理プログラミング ―その「基盤」と「最新 API」を究める」 ソフトバンククリエイティブ (2006/11/22)
ISBN-13: 978-4797337204

[10] 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」IPSJ-GI11026011