

# 分散オブジェクトフレームワークにおける スケールアウト機能の実現

鳴原 颯矢<sup>1</sup> 杉山 安洋<sup>2</sup>

**概要：**分散オブジェクトシステムは大きな計算処理を複数の処理に分割し、それらを複数の計算機で並列に実行することで負荷分散を図る。しかし、たとえ複数の計算機を使用する分散オブジェクトシステムであっても、ユーザの増加や処理するデータが増加すると、負荷が過剰となりスループットが低下する。そのため、システムの負荷に応じたスケラブルなシステム構成にすることが望まれる。我々の研究室では、分散オブジェクトフレームワークである TRMI (Transparent Remote Method Invocation) を開発している。TRMI を用いることで、スタンドアロンシステム上で処理を行うアプリケーションを何も書き替えることなく自動的に分散型アプリケーション化することが可能となる。今回、TRMI にスケラブルな分散型アプリケーションを開発できる機能を追加した。TRMI を用いて分散化されたアプリケーションは、アプリケーションの負荷に応じてオブジェクトを割り当てる計算機を自動的に追加可能とするようなスケールアウト機能を持つ。本稿では、今回実現したスケールアウト機能の実現手法とその評価について述べる。

**キーワード：**分散処理、分散オブジェクトフレームワーク、スケールアウト

## A Scale-out Mechanism for a Distributed Object Framework

RYUYA SHIGIHARA<sup>1</sup> YASUHIRO SUGIYAMA<sup>2</sup>

### 1. はじめに

分散オブジェクトシステムは大きな計算処理を複数の処理に分割し、それらを複数の計算機で並列に実行することで負荷分散を図る。しかし、たとえ複数の計算機を使用する分散オブジェクトシステムであっても、ユーザの増加や処理するデータが増加すると、負荷が過剰となりスループットが低下する。そのため、システムの負荷が増加した場合でも、スループットが低下しないスケラブルなシステム構成にすることが望まれる。

我々の研究室では、分散オブジェクトシステム（以後、分散型アプリケーションと呼ぶ）の開発効率向上のため、Java 言語の RMI (Remote Method Invocation) [1] を拡張した分

散オブジェクトフレームワークである TRMI (Transparent Remote Method Invocaton) [2] を開発している。TRMI を用いることで、スタンドアロンシステム上で処理を行うアプリケーション（以後、スタンドアロン型アプリケーションと呼ぶ）を自動的に分散型アプリケーション化することが可能となり、分散型アプリケーションの開発効率向上が期待できる。

今回、TRMI にスケラブルな分散型アプリケーションを開発できる機能を追加した。TRMI を用いて分散化されたアプリケーションは、アプリケーションの負荷に応じてオブジェクトを割り当てる計算機を自動的に追加可能とするようなスケールアウト機能を持つ。本稿では、今回実現したスケールアウト機能の実現手法とその評価について述べる。

### 2. TRMI

RMI は、クライアントオブジェクトが自身の Java 仮想

<sup>1</sup> 日本大学大学院工学研究科情報工学専攻  
Graduate School of Engineering, Nihon University

<sup>2</sup> 日本大学工学部情報工学科  
Department of Computer Science, College of Engineering,  
Nihon University

マシンとは異なる Java 仮想マシンの上に存在するサーバオブジェクトのメソッド呼び出しを行う仕組みである。RMIを用いることにより、クライアントオブジェクトは通常のオブジェクトのメソッド呼び出しに近い方法で、サーバオブジェクトのメソッド呼び出しを行うことが可能となる。しかし、RMI を用いて分散システムを構成する際に、スタンドアロン型アプリケーション本来の機能であるアプリケーションロジックに加え、分散処理を行うために特有のネットワークに関連した処理をプログラム内に記述しなければならない。そのため、アプリケーションロジックとネットワーク処理を合わせて開発することは処理の混合を招き、アプリケーションの開発作業を必要以上に複雑化する問題が存在する。

## 2.1 TRMI の概要

TRMI の特徴は、スタンドアロン型アプリケーションへ分散処理のための変更を記述することなく、分散型アプリケーション化することを可能とすることである。スタンドアロン型アプリケーションは、TRMI ランタイムシステムを用いて、ネットワーク処理を意識せずに分散処理を行うことができるようになるため、アプリケーションロジックとネットワーク処理を分離して開発することが可能となる。

スタンドアロン型の Java プログラムは図1左のように、複数のオブジェクトから構成され、通常はすべてのオブジェクトが同一の計算機上で実行される。TRMI を用いることにより、図1右の分散システムのようにそれらのオブジェクトを別々の計算機で実行可能となる。なお、分散型アプリケーションの処理起点となるオブジェクトをクライアントオブジェクト、クライアントオブジェクトから呼び出されるオブジェクトをサーバオブジェクトと呼ぶ。また、クライアントオブジェクトを実行する計算機をクライアントマシン、サーバオブジェクトを実行する計算機をサーバマシンと呼ぶ。

サーバオブジェクトを実行するサーバマシンは、図2に示すように、アプリケーションの実行開始時にサーバマシンの情報を記述した conf ファイルを読み込み決定する。ユーザは conf ファイルに address と port という属性を記述する。address 属性はサーバオブジェクトを実行す

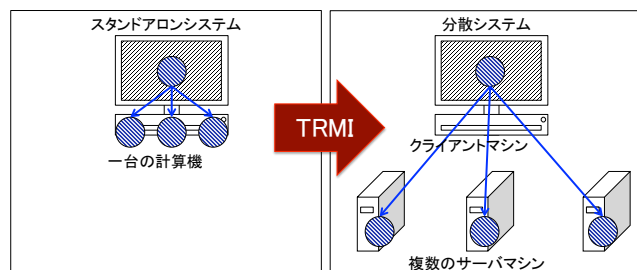


図1 TRMI の概要

Fig. 1 An overview of TRMI.

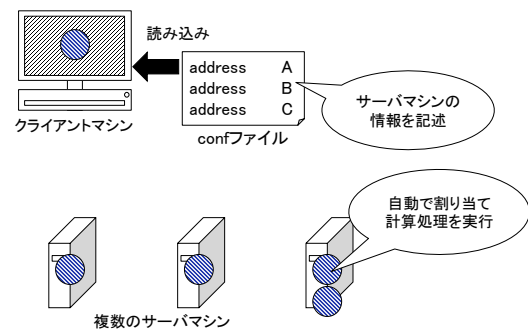


図2 TRMI の実行方式

Fig. 2 A distributed object system generated by TRMI.

るサーバマシンのホスト名や IP アドレスを表す。port 属性はサーバオブジェクトと通信する際に使用するポート番号を表す。また、サーバマシンへ割り当てるサーバオブジェクトの数を指定する属性である multiplexity 属性がある。さらに、ユーザが自由に属性を追加記述することも可能である。

## 2.2 解決すべき課題

分散オブジェクトシステムでは、ユーザの増加や処理するデータが増加すると、負荷が過剰になりスループットが低下する問題がある。例えば、図2では、右のサーバに2つのオブジェクトが割り当てられているため、負荷が過剰となっている可能性がある。このような場合は、図3のように使用するサーバマシンを追加するスケールアウトを行うことにより負荷分散を図ることが望まれる。

しかし、これまでの TRMI ではスケールアウトを行うことはできない。TRMI を用いて開発した分散型アプリケーションは、実行を開始する際に conf ファイルを読み込み使用するサーバマシンを決定する。そのため、使用するサーバマシンを決定することが実行開始時のみとなり、アプリケーションが使用するサーバマシン数は固定され、負荷が増加する場合、スループットの低下に繋がる。

また、実行中にサーバマシンが不足する問題を解決するためには、あらかじめ多くのサーバマシンを用意しておく方式も考えられる。しかし、conf ファイルへ必要な台数以

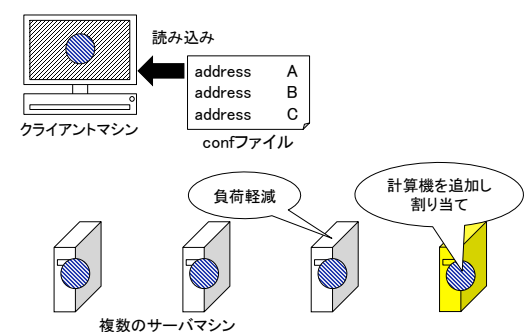


図3 TRMI へのスケールアウト機能の追加

Fig. 3 Implementing a scale-out function for TRMI.

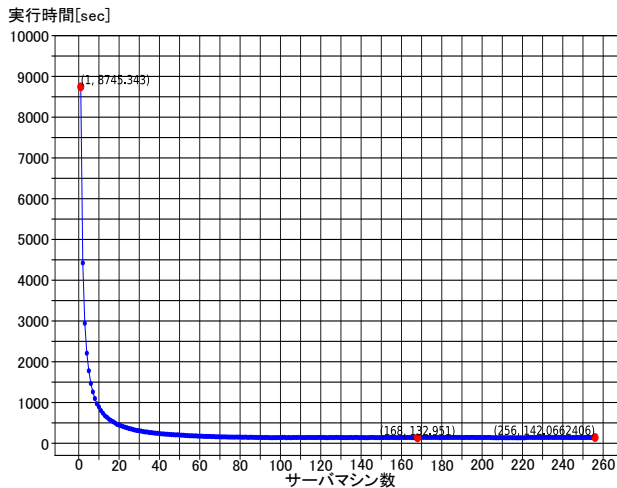


図 4 サーバマシン数と実行時間の関係

Fig. 4 Relationship between the number of servers and program execution time.

上のサーバマシンを記述した場合、サーバリソースが無駄になってしまう。図 4 は、ある分散型アプリケーションを実行した際の、サーバマシン数と実行時間の関係を表すグラフである。縦軸が実行時間であり、横軸はサーバマシン数となっている。conf ファイルへサーバマシンの台数を増やしても、図 4 のように、実行時間が台数に応じて短くなるわけではない。必要以上の台数を使用することは、リソースの有効利用の観点から望ましくない。

### 3. TRMI のスケールアウト機能

本稿では、TRMI を用いて分散型アプリケーションを開発する際に、必要な台数だけサーバマシンを自動的に追加してアプリケーションの実行時間を短くするような、スケールアウト機能の実現手法を提案する。

今回実現したスケールアウト機能には、アプリケーションの管理者が手動で使用するサーバマシンを追加できる静的スケールアウト機能と、アプリケーションの実行状況に応じて自動的にサーバマシンを追加する動的スケールアウト機能の 2 つがある。

#### 3.1 これまでの割り当て手法

本節では、まず、比較のためスケールアウト機能を用いない、これまでの TRMI のオブジェクト割り当て手法を説明する。なお、TRMI のオブジェクト割当手法は、何種類かの割り当てアルゴリズムが用意されており、それらを開発者であるユーザが選択できる方式となっている。本稿では、それらのアルゴリズムの中で、最も標準的なアルゴリズムを例として説明する。

TRMI の標準のオブジェクト割り当て手法では、図 5 に示すように、conf ファイルへ使用するサーバマシンを address 属性でユーザが予め記述しておく。また、TRMI

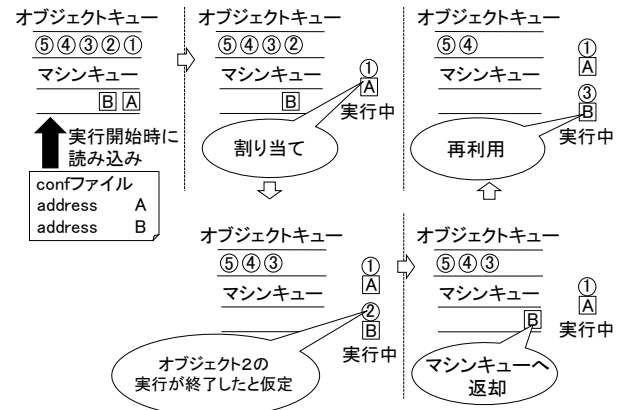


図 5 これまでの割り当て手法

Fig. 5 The current object allocation method of TRMI .

のランタイムシステムは、実行対象のサーバオブジェクトを管理するオブジェクトキューと、使用可能なサーバマシンを管理するマシンキューを持つ。これら 2 つのキューを用いてサーバオブジェクトをサーバマシンへ割り当てる。

まず、アプリケーション実行開始時に conf ファイルを読み込み、記述されている address 属性のサーバマシンをマシンキューへ追加する。そのため、使用可能なサーバマシン数はアプリケーション実行開始時に、conf ファイルに記述されたサーバマシン数となる。アプリケーションの実行が開始され、実行対象となるサーバオブジェクトが生成されると、それらがオブジェクトキューに順次追加される。オブジェクトキュー中のサーバオブジェクトは、FIFO の順でマシンキューのサーバマシンへ割り当てられ実行が開始される。割り当てられたサーバオブジェクトとサーバマシンは、それぞれオブジェクトキューとマシンキューから削除される。マシンキューが空になるとサーバマシンが不足し、サーバオブジェクトの割り当てが不可能となる。すると、サーバオブジェクトは実行を開始できず待機状態となる。

実行中のサーバオブジェクトの計算処理が終了すると、サーバマシンは再度マシンキューの最後尾へ返却され、再び割り当ての対象となり待機状態のオブジェクトが割り当てられ実行が開始される。図 5 では、オブジェクト 2 の計算処理が終了した際に、サーバマシン B はマシンキューに返却され、オブジェクト 3 の実行に再利用されている状況を表している。同様に、オブジェクト 4, 5 も実行の終了したサーバマシンに順次割り当てられて実行される。

なお、サーバマシンの台数が不足する場合には、conf ファイル中のサーバマシンを巡回的に使用するアルゴリズムを選択することもできる。この場合は一台のマシンで複数のサーバオブジェクトが実行されることになる。しかし、今回実現したスケールアウト機能は、サーバマシンが不足した場合にはサーバオブジェクトが待機状態になるアルゴリズムを前提にしている。そのため、本稿でもサーバマシン

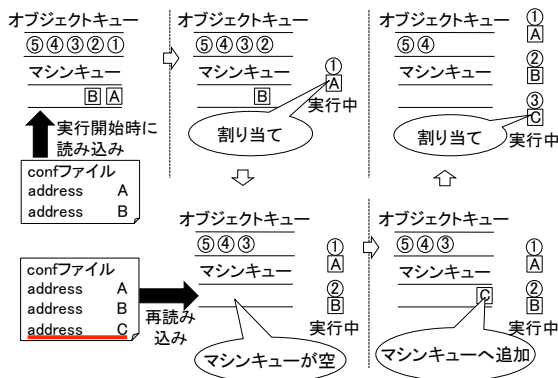


図 6 静的スケールアウト機能  
Fig. 6 The static scale-out.

が不足した場合にはサーバオブジェクトが待機状態になるアルゴリズムをもとに説明する。

### 3.2 静的スケールアウト機能

静的スケールアウト機能は、アプリケーション実行中であっても、サーバオブジェクトを割り当てるサーバマシンが不足するような場合には、使用可能なサーバマシンをアプリケーションの実行管理者が追加できる機能である。これにより、待機状態となるサーバオブジェクトを減らすことが可能となる。具体的には、図6に示すように、処理を行う。

まず、先ほどと同じくアプリケーション実行開始時に conf ファイルを読み込み、記述されているサーバマシンをマシンキューへ追加する。マシンキューのサーバマシンへサーバオブジェクトを割り当て実行が開始され、アプリケーション実行中にマシンキューが空になるとこれまでだと待機状態となってしまう。

そこで、conf ファイルをアプリケーション実行中であっても読み込む機能を追加する。conf ファイルを読み込むタイミングは、マシンキューが空となりサーバマシンが不足する際に、ユーザが conf ファイルへ追加記述を行った場合となる。読み込んだ conf ファイルとそれ以前に読み込んだ conf ファイルの差分を求め、新たに追加されたサーバマシンをマシンキューに追加する機能を追加する。なお、conf ファイルにサーバマシンを追加する際には、同名のサーバを追加した場合や conf ファイルの行間に追加記述を行った場合であっても差分として認識される。

読み込んだ conf ファイルに変更がなければ、キューには何も追加されないが、一定時間後に再度 conf ファイルを読み込む処理を conf ファイルが変更されるまで繰り返す。そこで、新たなサーバマシンが検出された場合にはマシンキューに追加する。すると、待機状態のオブジェクトがマシンキューへ追加されたサーバマシン上で実行され、これまでの割り当て手法と同様に実行を継続する。

以上の処理により、実行中にサーバマシンの追加ができ

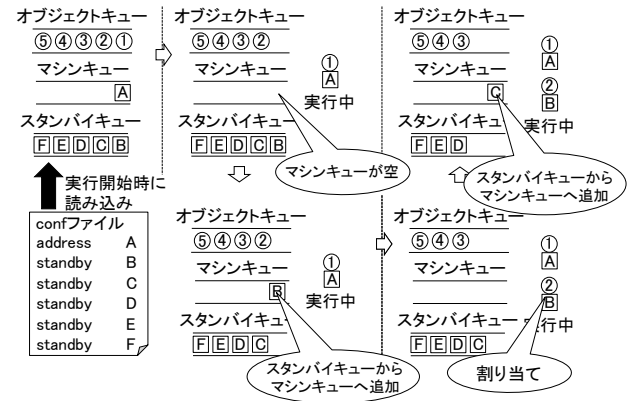


図 7 動的スケールアウト機能  
Fig. 7 The dynamic scale-out.

るようになり、待機状態のサーバオブジェクトを減らすことが可能となる。

### 3.3 動的スケールアウト機能

次に動的スケールアウト機能について説明する。動的スケールアウト機能は、アプリケーション実行中に使用するサーバマシンを自動的に追加可能とするオートスケール機能である。

動的スケールアウト機能では、図7に示すように、conf ファイルへ使用するサーバマシンを address 属性あるいは standby 属性で記述する。address 属性で指定されるマシンは実行の開始時から使用するマシンである。一方、standby 属性で指定されるマシンは実行の開始時には使用されないが、スケールアウトが必要になった時に追加されるユーザが保有している予備のサーバマシンである。

今回、TRMI のランタイムシステムに、オブジェクトキューとマシンキュー以外に、ユーザが保有している予備のサーバマシンをリソースプールとして管理するためのキューであるスタンバイキューを追加した。まず、先ほどと同じくアプリケーション実行開始時に conf ファイルを読み込み、記述されている属性が address のサーバマシンをマシンキューへ、standby のサーバマシンをスタンバイキューへ追加する。マシンキューのサーバマシンへサーバオブジェクトを割り当て実行が開始され、実行中にマシンキューが空になりサーバマシンが不足すると、スタンバイキューからサーバマシンをマシンキューへ追加される。追加するタイミングは、マシンキューが空でサーバマシンが不足する場合となる。スタンバイキューからマシンキューへサーバマシンを追加後、これまでの割り当て手法と同様に実行を継続する。以上の処理により、自動的にサーバマシンの追加が可能となり、ユーザがアプリケーションの実行状況を監視する負荷を軽減することが可能となる。また、ユーザはアプリケーションの実行時間が最適になるように、サーバマシン数を考慮する必要がなくなる。

表 1 静的スケールアウト機能の評価結果

Table 1 An evaluation result of the static scale-out.

| ケース番号 | 使用したマシン数 | スケールアウト有無 | 実行時間 [sec] |
|-------|----------|-----------|------------|
| 1     | 1        | 無         | 9197       |
| 2     | 8        | 無         | 1147       |
| 3     | 16       | 有         | 739        |
| 4     | 24       | 有         | 577        |

## 4. スケールアウト機能の評価

今回開発したスケールアウト機能が有効に動作するか評価を行った。

評価は 5 次の魔方陣の解の総数を求めるプログラムを用いて行った。魔方陣の計算は 256 個のサーバオブジェクトで並列に行うものである。このアプリケーションは SIMD 型のアプリケーションで、256 個のサーバオブジェクトは同一クラスのインスタンスである。各サーバオブジェクトで異なる範囲の解の個数を求め、それらをクライアント側で合算して解の総数を求めている。また、256 個のサーバオブジェクトの計算粒度は様々である。それぞれのサーバオブジェクト一つの最大と最小の実行時間は約 100 秒と 0.001 秒である。また、使用したマシンの CPU は Intel Xeon X5570 2.93GHz である。

### 4.1 静的スケールアウト機能の評価手法

静的スケールアウト機能の評価では、これまでのスケールアウトを用いない割り当て手法を用いて実行した結果と、静的スケールアウト機能を用いて実行した結果を比較し、サーバマシン数を実行時に追加することにより実行時間の短縮が図れることを確認する。

比較には 4 つのケースを用いて評価を行った。ケース 1 は、実行開始時のサーバマシン数を 1 台、スケールアウト機能を用いずに、256 個のサーバオブジェクトを逐次実行し実行時間を計測した。ケース 2 は、実行開始時のサーバマシン数を 8 台、スケールアウト機能を用いずに、256 個のサーバオブジェクトを並列実行し実行時間を計測した。同時に実行できるオブジェクト数は 8 である。ケース 3 は、実行開始時のサーバマシン数を 8 台、実行開始から 300 秒後に静的スケールアウト機能を用いて新たに 8 台のサーバマシンを追加し、合計 16 台で並列実行し実行時間を計測した。ケース 4 は、実行開始時のサーバマシン数を 8 台、実行開始から 180 秒後に静的スケールアウト機能を用いて新たに 8 台のサーバマシンを追加し、さらに、360 秒後に 8 台のサーバマシンを追加し、合計 24 台で並列実行し実行時間を計測した。

### 4.2 静的スケールアウト機能の評価結果

実行時間を計測した結果を表 1 に示す。また、図 8 と図

表 2 動的スケールアウト機能の評価結果

Table 2 An evaluation result of the dynamic scale-out.

| 割り当て手法        | 当初指定したマシン数 [台] | 実際に使用したマシン数 [台] | 実行時間 [sec] |
|---------------|----------------|-----------------|------------|
| スケールアウト機能無し   | 256            | 256             | 144        |
| 動的スケールアウト機能有り | 1              | 126             | 135        |

9 は、それぞれ左からケース 1 とケース 2、ケース 3 とケース 4 のサーバマシンへ割り当てたサーバオブジェクト数と実行時間の関係を表すグラフである。縦軸が実行時間、横軸がサーバマシンの通し番号である。さらに、グラフの縞模様は、1 つ 1 つのオブジェクトの実行時間を区別しており、1 つのオブジェクトの実行時間を表している。ケース 3 では、サーバマシン 1 番から 8 番の実行すべきジョブが、追加されたサーバマシン 9 番から 16 番へ移動できたことが確認できた。同様にケース 4 では、サーバマシン 1 番から 8 番の実行すべきジョブがサーバマシン 9 番から 16 番とサーバマシン 17 番から 24 番へ移動できたことが確認できた。

実行結果の表とグラフから、静的スケールアウト機能を用いることで、プログラムの実行時間が予測通り短くなることを確認できた。

### 4.3 動的スケールアウト機能の評価手法

動的スケールアウト機能の評価では、実行時間の短縮化だけではなく、必要な台数のマシンを自動的に追加することで使用するマシン台数の最適化が図れることを確認する。スケールアウト機能を用いないこれまでの割り当て手法を用いて実行した結果と、動的スケールアウト機能を用いて実行した結果を比較する。

動的スケールアウト機能を用いないこれまでの割り当て手法では、実行開始時に conf ファイルへサーバマシン 256 台を address 属性で記述し、256 個のサーバオブジェクトを並列で実行し、実行時間と使用したサーバマシンの台数を計測した。

動的スケールアウト機能を用いた場合では、実行開始時に conf ファイルへサーバマシン 1 台のみを address 属性で記述し、サーバマシンを 255 台を standby 属性で記述した状態で、256 個のサーバオブジェクトを並列実行し、実行時間と使用したサーバマシンの台数を計測した。

### 4.4 動的スケールアウト機能の評価結果

計測した結果を表 2 に示す。スケールアウト機能を用いないこれまでの割り当て手法より動的スケールアウト機能を用いた割り当て手法の方が実行時間が短くなっていることが確認できる。また、オブジェクトを割り当てたサーバ

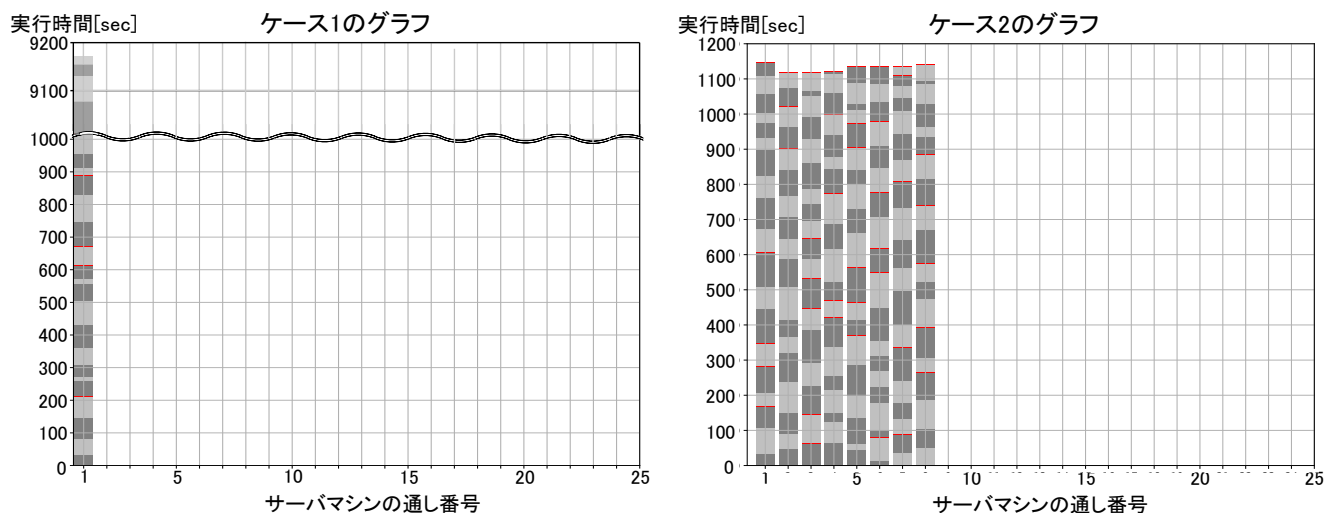


図 8 静的スケールアウト機能を用いない場合の評価結果を表すグラフ

Fig. 8 Evaluation graphs without static scale-out.

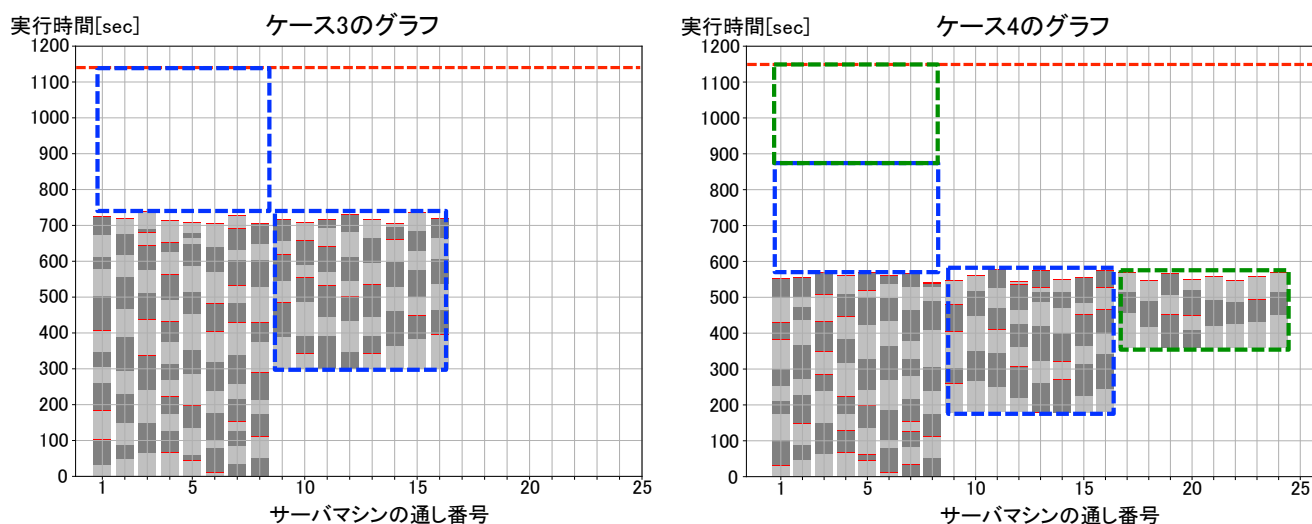


図 9 静的スケールアウト機能を用いた場合の評価結果を表すグラフ

Fig. 9 Evaluation graphs with static scale-out.

マシン数も少なくなっていることが確認できる。

また、図 10 は各サーバマシンにおけるサーバオブジェクトの実行状況を表すグラフである。左側がスケールアウト機能を用いない場合で、右側が動的スケールアウト機能を用いた場合である。縦軸が実行時間、横軸がサーバマシンの通し番号となっている。

図 10 左のグラフより、スケールアウト機能を用いない場合はサーバマシン 1 番から 256 番のサーバマシンへオブジェクトを割り当て実行していることが確認できる。1 台のマシンでは一つのサーバオブジェクトのみが実行されている。また、何も実行されていないように見えるマシン、例えば 200 番以降のマシンなどは処理時間が非常に短いマシンである。また、点線枠で示す実行開始時間の遅れは、オブジェクトのバイトコードをサーバマシンへアップロードするための時間である。

一方、図 10 右の動的スケールアウト機能を用いた場合のグラフでは、縞模様でオブジェクトを一つ一つの実行時間を区別している。このグラフからは、動的スケールアウト機能を用いた場合には、サーバマシン 1 番から 126 番のサーバマシンのみへオブジェクトを割り当て実行していることが確認できる。また、複数のサーバオブジェクトが 1 台のサーバマシンへ割り当てられ逐次に行われていることが確認できる。すなわち、図 10 左では、各サーバマシンが一つのオブジェクトの実行終了後に再利用されていなかったが、動的スケールアウトを用いたことで図 10 右のように早めに実行を終了したサーバマシンが再利用され、サーバマシンの有効利用が図れたことが確認できる。

評価結果の表とグラフから、動的スケールアウト機能を用いることでアプリケーションの負荷に応じたオブジェクトの割り当てが実現でき、アプリケーションの実行時間が

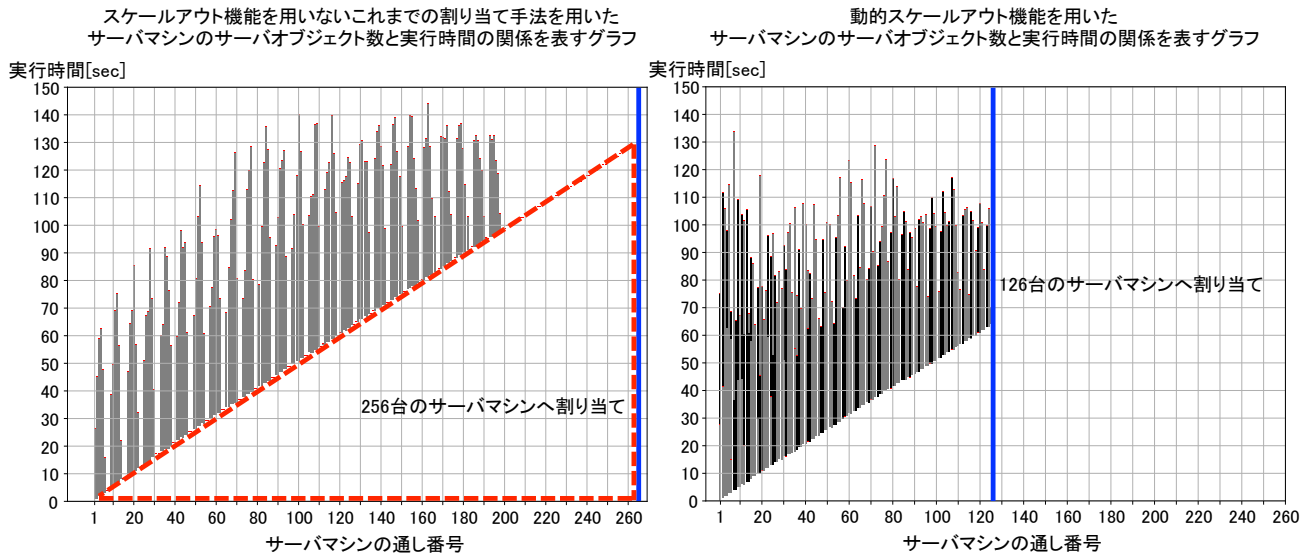


図 10 動的スケールアウトの評価結果を表すグラフ  
Fig. 10 Evaluation graphs of dynamic scale-out.

予想通り短くなることが確認できた。

しかし、図 10 からは、各サーバマシンでの実行開始時のオーバーヘッドが無視できない大きな値であることも確認できた。これは前述したようにサーバオブジェクトのバイトコードをサーバマシンへアップロードするための時間である。また、評価結果の図には含まれていないが、今回はサーバマシン 1 台で同時実行するオブジェクトは 1 つであるという制約を設けたために、各サーバマシンの CPU 利用率が低く、有効活用されていないという結果も確認できた。

## 5. 考察

サーバオブジェクトの実行開始時のオーバーヘッドを減らし、さらに CPU 使用率の向上を図るために、次のような実験を行った。今回評価に用いたサーバマシンは、4 コアで 8 スレッドを並列実行可能な CPU を 2 つずつ搭載している。そこで、各サーバマシンで同時実行するサーバオブジェクトの個数を変化させることにより、実行時間や使用されるサーバマシンの台数がどう変化するかを検証を行った。conf ファイルに multiplexity 属性を記述することで、同時実行されるサーバオブジェクト数を調整できる。各サーバマシンの総コア数が 8 であることから、1 台のサーバマシンへ 8 個のサーバオブジェクトを割り当てると、最も効率が良いことが推測される。また、すべてのサーバオブジェクトのクラスは同一であることから、1 台のマシンで複数のオブジェクトを実行しても各サーバへのバイトコードのアップロードは一度しか行われないうために、実行開始時のオーバーヘッドを減らすことが期待できる。

比較には 3 つのケースを用いて評価を行った。ケース 1 では、各サーバマシンに割り当てるサーバオブジェクト数

の上限を 8 とし、動的スケールアウト機能を用いて実行する。割り当てられた 8 個のサーバオブジェクトは並列実行される。ケース 2 では、各サーバマシンに割り当てるサーバオブジェクト数を 16 とし、動的スケールアウト機能を用いて実行する。この場合も割り当てられた 16 個のサーバオブジェクトは並列実行される。ケース 3 は、各サーバマシンに割り当てるサーバオブジェクト数を 16 とするが、それらのうち同時実行するものは 8 個に限定し、動的スケールアウト機能を用いて実行する。一台のサーバマシンに割り当てられた 16 個のサーバオブジェクトの内の 8 個は待機状態となり、先行して実行されているサーバオブジェクトの実行が終了すると、待機状態のオブジェクトの実行が開始される。CPU の負荷を抑えながら、利用効率を向上させることを狙っている。

計測した結果を表 3 に示す。また、図 11 に各サーバマシンの実行状況を示す。ケース 1 では、19 台のサーバマシンが使用され、実行時間は約 111 秒であった。表 2 と比べて、実行時間、実行開始時のオーバーヘッドが短くなっていることが確認できる。使用するマシンの台数も大幅に減少しており、サーバ資源の有効活用を図りながら、実行時間の短縮が可能であることが分かる。

ケース 2 では、使用されるサーバマシンの台数は 10 台

表 3 オブジェクトの割り当て数による実行時間の変化  
Table 3 The dynamic scale-out with various object allocation.

| ケース番号 | 使用したマシン数〔台〕 | オブジェクト割り当て数(同時実行数) | 実行時間〔sec〕 |
|-------|-------------|--------------------|-----------|
| ケース 1 | 19          | 8(8)               | 111.03    |
| ケース 2 | 10          | 16(16)             | 129.64    |
| ケース 3 | 16          | 16(8)              | 108.70    |

と減少したが、実行時間が約 130 秒とケース 1 に比べて長くなっていることが分かる。これは、8 つのコアで 16 スレッドを同時実行したことによる影響である。

ケース 3 では、16 台のサーバマシンが使用され、実行時間は約 108 秒であった。使用するマシン数は若干増加したが、実行時間がケース 1 より若干短くなったことが確認できた。これは、処理が早く終わったマシンをうまく再利用できたことを示している。

これにより、動的スケールアウト機能だけではなく、サーバマシンで同時実行するサーバオブジェクトの数を調整することにより実行時間の短縮や、サーバリソースの有効活用ができることが確認できた。

## 6. まとめと今後の課題

本稿では、TRMI で分散化されたアプリケーションに追加した静的スケールアウト機能、動的スケールアウト機能の実現方式について述べた。また、それぞれのスケールアウト機能の有効性を確立するため、計算処理の粒度が均一でないプログラムを用いた実行時間の評価結果について述べた。静的スケールアウト機能の評価結果では、必要な台数のマシンを手動で追加することで実行時間の短縮化が実現できた。また、動的スケールアウト機能の評価結果では、必要な台数のマシンを自動的に追加することで使用するマシン台数の最適化、実行時間の短縮化が実現できた。さらに、各サーバマシンで同時実行するサーバオブジェクトの数を調整することにより、さらなる使用するマシン台数の最適化や実行時間の短縮化が実現できた。

今後の課題としては、サーバマシンの台数を自動調整するスケールアウト機能のみならず、各サーバマシンで実行するサーバオブジェクトの数を調整する機能を実現したい。5 章では、各サーバマシンで実行するサーバオブジェクト数を手動で調整して実験を行ったが、これを実行中に自動的に調整できるようにしたいと考えている。

また、今回は関連研究の調査が済んでいないため、関連研究を調査し、本研究の改善に役立てたいと考えている。

### 参考文献

- [1] Oracle, Remote Method Invocation Home, <http://www.oracle.com>
- [2] 杉山安洋: TRMI によるオブジェクトの分散化, コンピュータソフトウェア, Vol.19, No.5, pp.40-50, 2002

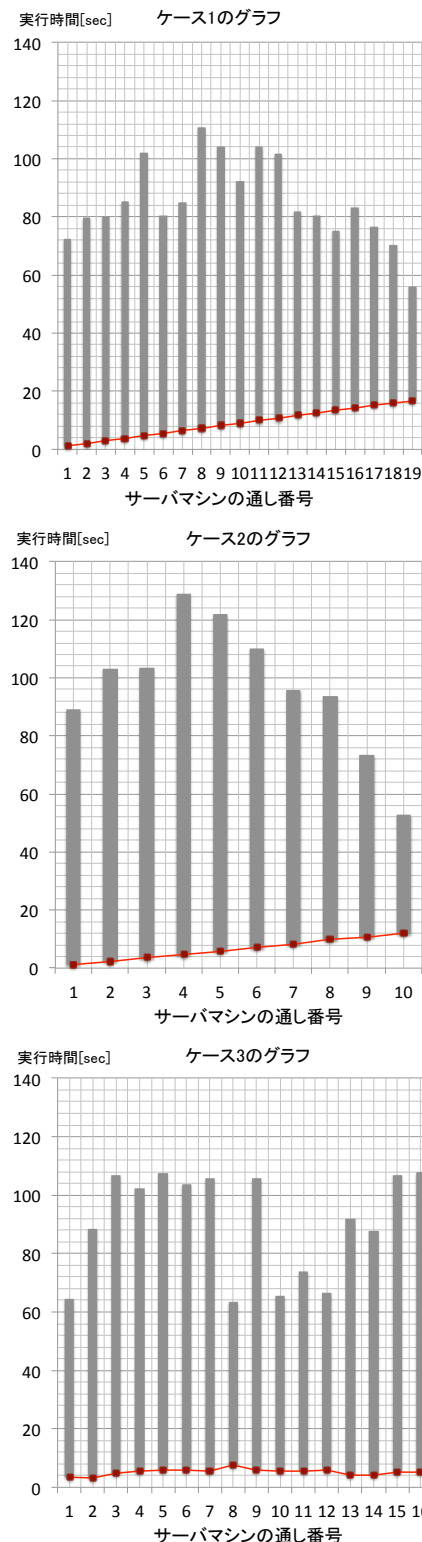


図 11 各サーバマシンで実行するオブジェクト数を変更した場合の実行状況

Fig. 11 Evaluation graphs of dynamic scale-out with different multiplexity.