平成15年度学士論文

過負荷時の分散ソフトウェアの性能劣化を改善するスケジューリングの提案

東京工業大学 理学部 情報科学科
学籍番号 00-1966-6
日比野 秀章

指導教官
千葉 滋 助教授

平成16年2月27日

概要

近年、リモートメソッド呼び出しを基軸とする分散オブジェクト技術の延長として、EJB(Enterprise Java Beans) のような分散コンポーネント技術が広く利用されるようになった。本論文では、分散コンポーネントの処理性能を向上させる手法を提案する。コンポーネントとは、機能単位で分割したソフトウェアのことで、プログラムコードの再利用性が高い等の利点がある。

多くの EJB コンテナは、分散コンポーネント間の相互のリモートメソッド呼び出しを並列化し処理性能の向上を図っている。しかしながら、リモートメソッド呼び出しの処理は必ずしも並列化させた方が良いというわけではない。並列度を適切に制御しないと、コンポーネント同士が CPU やメモリなどの計算機リソースの競合を引き起こしサーバの性能を低下させることがある。例えば、巨大な XML ファイルから特定のデータを探索するような、大量のメモリ資源を消費するコンポーネントでは、無制限に呼び出しを受け入れた場合、サーバ全体の処理性能が低下してしまう。また、あるコンポーネントの処理が他のコンポーネントに干渉し、処理効率を悪化させるという場合もある。

EJB コンテナの中には、生成されるインスタンス数の上限を Bean 毎に決めることで、同時に受け入れる呼び出しを制限できるものもある。そのような EJB コンテナでは、インスタンス数の上限を設定ファイル (配置記述子) に記述することで固定的に接続数を制限することができる。しかし、最適な処理の並列度はその他の Bean や計算機上の他のプロセスの挙動に依存するため静的には決められない。すなわち、静的に決める制限値は、あくまで無制限に呼び出しを受け入れることで、陥る DoS (Denial of Service) を回避する手段でしかない。したがって、実行時に変動する余剰リソースを効率的に利用するような並列度の設定も行うことはできない。我々は、分散コンポーネント間の相互のリモートメソッド呼び出し処理の最適な並列度を、動的に算出、適用する Method-level Queue Scheduling を提案する。本手法では、リソース競合を起こさずに効率よく処理できる並列度を、メソッド単位で設定する。最適な並列度の決定には、Progress-based regulation を利用している。Progress-based regulation とは、進捗状況をみて資源の競合を判断し、資源の割り当てを動的に変更するスケ

ジューリング方法である。また、優先度を設定することにより、特定のホストからのリモートメソッド呼び出しを優先的に処理する仕組みを、本手法に取り入れた。この機能は、ホストの識別と、処理順序の変更から成る。

我々は本手法を、サーバアプリケーションの実装で広く使われている EJB の基軸とも言える Java RMI 用の API として実装した。さらに実験により、リソースを大量に消費する処理の並列度が制限され、リモートメソッドの処理のスループットが改善されることを確認した。また、高い優先度が設定されたホストからのリモートメソッド呼び出しが、優先度を設定していないホストからの呼び出しに比べ、短時間で処理されることを確認した。

謝辞

本研究を進めるに辺り、研究の方針や進め方について数々の助言をしてくださった指導教官の千葉滋助教授に感謝致します。

また、研究の方針について様々な助言をしていただいた光来健一助手に大変感謝致します。同研究室所属の佐藤芳樹氏、東京工業大学大学院の西澤無我氏、松沼正浩氏、柳澤佳里氏には、多くの疑問を聞いて頂き、様々な助言を頂きました。感謝致します。

また本研究に必要な多くの知識を与えてくださった東京工業大学大学院の中川清志氏、栗田亮氏、宇崎央泰氏、須永豊氏に感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

現在、情報システムはますます肥大化し複雑化する方向にある。そして、ただでさえ複雑なシステムが結び付き、さらに巨大な複合システムを構成している。このような複雑なシステムの開発コストを最小化し、しかも後々の保守を容易にするため、分散コンポーネント技術が広く利用されている。分散コンポーネントとは、コンポーネントと呼ばれる特定の機能を持つ再利用可能なコードを組み合わせることで必要な機能を実現し、異なるマシンに分散させて配置することで負荷や機能を分散させる技術である。代表的なものでは CORBA、EJB、DCOM などがある。

これら分散コンポーネント間では、通信を行うためにリモートメソッド呼び出し (RMI、Remote Method Invocation) が一般的に使用されている。RMI は、分散コンポーネントでお互いの物理的な位置を意識せずに通信するための技術であり、EJB コンテナを始めとする多くのコンテナでは、分散コンポーネント間の相互のリモートメソッド呼び出しを並列に処理することで性能の向上を図っている。確かに軽いメソッド、例えば処理時間の短いメソッドであれば、並列度が大きいほど性能は向上する。しかし、大量のメモリ消費やディスクアクセスを必要とする重いメソッドの場合、CPU やメモリなどの計算機リソースの競合を引き起こすため、並列度は小さい方が良い。

EJB コンテナの中には、生成されるインスタンス数の上限を Bean 毎に決めることで、同時に受け入れる呼び出しを制限できるものもある。そのような EJB コンテナでは、インスタンス数の上限を設定ファイル (配置記述子) に記述することで固定的に接続数を制限することができる。しかし、最適な並列度はその他の Bean や他のプロセスの挙動に依存するため静的には決められない。また、静的に決める制限値では、実行時に変動する余剰リソースを効率的に利用するような並列度の設定も行うことはできない。

そこで我々は、分散コンポーネント間での通信に利用されているリモートメソッド呼び出しについて、最適な並列度を動的に算出・適用する Method-level Queue Scheduling を提案する。本手法では、リソース競合を起こさずに効率よく処理できる並列度を、メソッド単位で設定する。また我々は、Java の分散コンポーネントである EJB でリモートメソッドの呼び出しに

利用されている Java RMI 用の API として、本手法を実装した。

本稿の残りは、次のような構成からなっている。第 2 章では、並列度の制限に関する問題点と関連技術について、第 3 章では、本研究で提案する制御法を、第 4 章では、Java RMI 上での実装を、第 5 章では、本研究の提案手法を用いた実験を、第 6 章でまとめを述べる。

第2章　問題点と関連技術

2.1　Webシステムでの並列度制限に関する問題点

従来 Web サーバやアプリケーションサーバでは、サーバ計算機の処理性能を向上させるためにマルチスレッドやマルチプロセスを用いてクライアントからのリクエストを並列に処理している。確かに、消費するリソースが少ない処理であれば、並列化することにより、CPU やメモリなどの計算機リソースが効率的に利用され、サーバの処理性能を向上させることができる。例えば、I/O 処理待ちによる CPU のアイドル時間を、別スレッドの実行に利用することにより、効果的に並列処理を行うことができる。しかし、大量のリソースを消費する処理では、並列に処理することで、リソースの競合を引き起こし、サーバの処理性能を低下させてしまう場合がある。例えば、メモリ資源が競合すると、利用可能なメモリを増やすために GC やスワップイン・スワップアウトが多発するため全体の処理性能が低下してしまう。

表 2.1: 実験環境

	サーバマシン	クライアントマシン
台数	1	15
CPU	Intel(R)Xeon(TM)CPU2.4GHz × 2	Pentium 733MHz
Memory	2GB	512MB
OS	Linux2.4.20	Linux2.4.19-Ovl11(VineLinux)
NIC	100BaseTX	100BaseTX

このような、最大の性能を引き出す並列度の違いというのは、Web サーバやアプリケーションサーバに限らず、内部で利用されている分散コンポーネント間の通信でも同様のことが言える。なぜなら、分散コンポーネント間の通信にはリモートメソッド呼び出しが一般的に利用されているが、リモートメソッド呼び出しも Web サーバやアプリケーションサーバ同様、クライアントサーバモデルに基づいており、サーバ側でメソッドが並列に処理されているからである。リモートメソッドによって最適な並列

度が異なることを検証するため、Java の分散コンポーネントである EJB のリモートメソッド呼び出しに利用されている Java RMI を使用して、軽い処理、重い処理についての簡単な性能実験を行った。実験環境は表 2.1 のようになっている。 軽い処理としては、単純な数値計算を、重い処理と

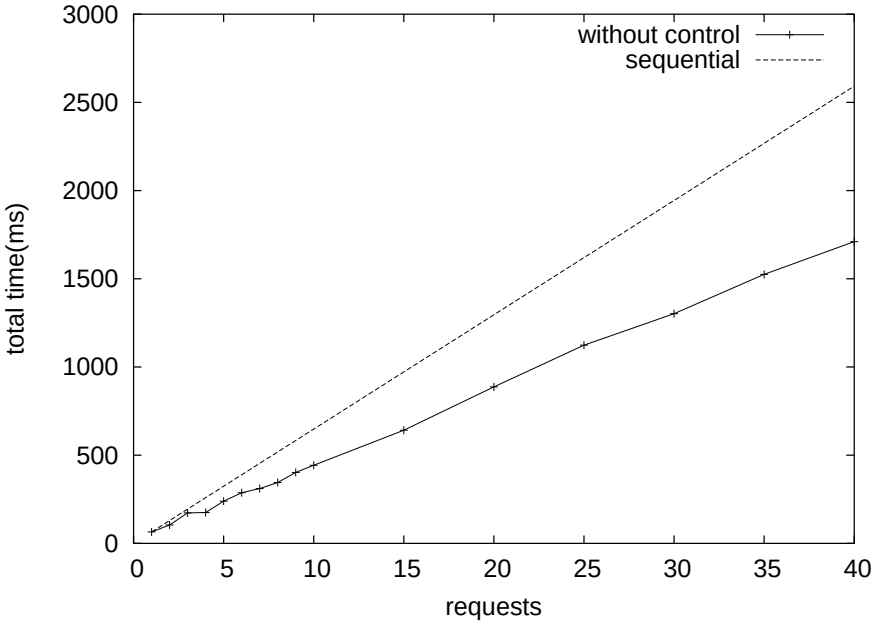

図 2.1: 並列処理により性能向上

しては、120KB 程度の XML ファイルのパース及びデータ探索を行った。実験内容は、最大で 40 の処理要求を同時に処理した際にかかった総処理時間の測定で、軽い処理の結果は図 2.1、重い処理の結果は図 2.2 の通りである。軽い処理については逐次的に処理する場合と比べると、並列に処理することにより性能が向上していることがわかる。しかし重い処理については、逐次的に処理した方が性能はよいことがわかる。これは、探索の際大量のオブジェクトを生成するため、競合が起きてしまっているからである。したがって、分散コンポーネント間の通信で利用されるリモートメソッドの処理でも、最適な並列度は異なっていることがわかる。

実際の分散アプリケーションでは、このようにサーバへの負荷が異なるリモートメソッドが分散コンポーネント間で複数種類呼び出されることになる。もちろん、呼び出されているリモートメソッドやその呼び出し数に応じて、サーバへかかる負荷は変化する。そのため分散コンポーネントで処理を円滑に行うためには、この変化に応じてそれぞれのリモートメソッドの並列度を決定する必要がある。そうでないと、サーバ計算機のリソー

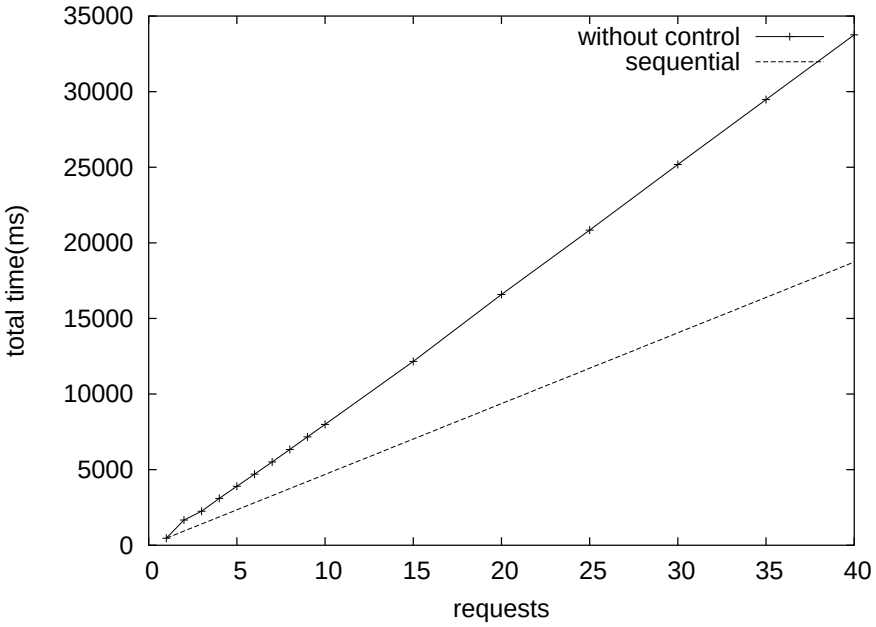

図 2.2: 並列処理により性能劣化

スを有効利用できないばかりか、サーバに負荷がかかりすぎてリソース競合が起き、処理性能を低下させてしまう場合がある。

以下では、並列度の制御に関する先行研究について述べる。

2.1.1　過負荷管理

過負荷はインターネットサービスでは避けることのできない問題であり、Web サービスの過負荷管理についてはいくつかの方法が提案されている。大きく分けると、過負荷管理には負荷分散と流量制限がある。負荷分散とは、リクエストを複数のノードに分散させ処理を実行することである。複数のノードで分散処理することにより、システム全体としてのスループットを向上させる。対して流量制限とは、リクエストをキューイングし、流入するリクエスト数を絞ることで、システムが過負荷状態になるのを防ぐことである。ここでは、並列度に関連する流量制限を中心に先行研究を述べる。

リソース制限

インターネットサービスでの資源管理の典型的な方法は、静的に資源を制限することである。すなわち、過度にリクエストを受け付けてしまう事態を避けるために、アプリケーションやサービスに割り当てられる資源を前もって制限してしまう。静的にというのは、システム管理者がプロセスやアプリケーションの使用する資源に制限をかすことである。資源の制限は時間とともに変化するかもしれないが、リソース制限では、システムを監視しパフォーマンスのフィードバックを利用して制限の変更をする、といったことはしない。

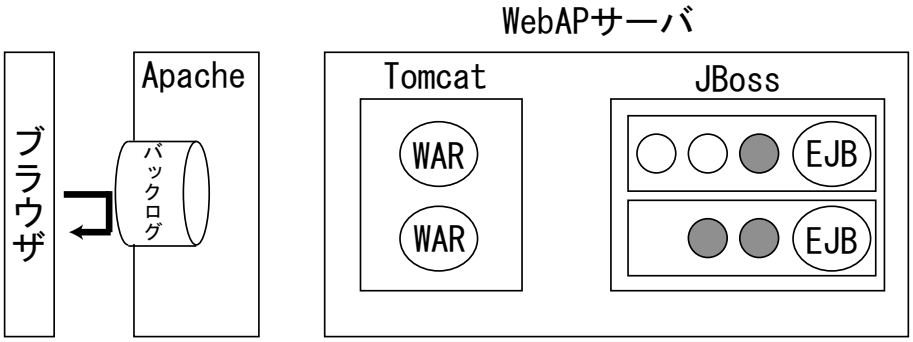

図 2.3: Apache の接続バックログ

接続毎に別のスレッドを利用する典型的な Web サーバでは、過負荷対策としてサーバが割り当てるプロセスや接続の数を制限する。サーバのスレッドがすべて利用されている場合、サーバは新たな接続の受け入れを止める。これは、Apache[1] で使用されている過負荷制御の方法である (図 2.3)。Apache では、リクエストを一旦バックログに格納する。バックログは格納するリクエストの数を設定することができ、それ以上の要求が来た場合にはエラーとなる。これにより、システムが過負荷状態になるのを防ぐことができる。しかし、エラーになった場合、ブラウザはシステムとは接続ができない状態なので、「サーバが見つかりません。」などのブラウザのエラーメッセージが表示されてしまう。そのため、ユーザにはサーバが過負荷であることが知らせられない。Web ブラウザは単純にサーバへの接続を待機していることを報告するだけである。また TCP が新しい接続の確立をするのに、指数的に増加するタイムアウトを利用していると、待ち時間が大変長くなってしまう。例えば、接続に失敗したと判断されるまでに数分かかってしまう。

他に、Web アプリケーションサーバの JBoss[3] では、リクエスト数を一旦キューイングする機能はないものの、起動するインスタンス数の上限

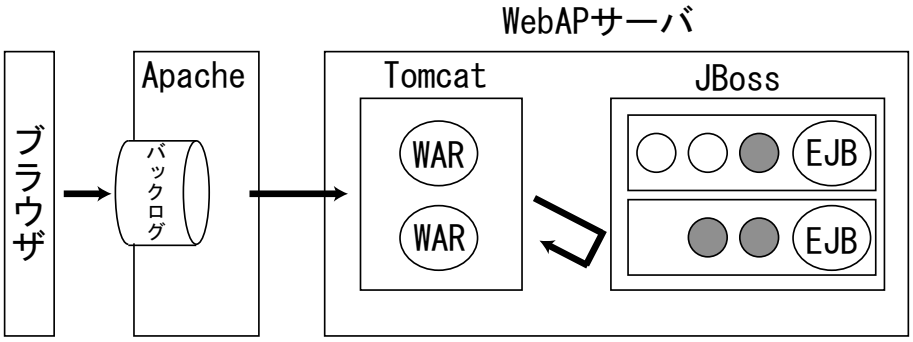

図 2.4: EJB のインスタンス数制限

を EJB 毎に設定することができる。これを使用して流量制限を行うことができる (図 2.4)。また、設定した上限以上の要求が来た場合はエラーとなるが、「ただいま混雑しております」などのエラーメッセージをユーザに表示するよう、リモートメソッドを呼び出した側で対処することはできる。また Apache のような処理の重さに関係なく要求を拒否してしまうのに比べると、よりサーバへの負荷を考慮した上で処理を拒否できる。

別のリソース制限としては、リアルタイムやマルチメディアシステムで典型的な方法がある。この方法では、”プロセス P が CPU 時間の X パーセントを得る”のように、予約や共有の形でリソースが制限される。このモデルでは、OS が各プロセスのリソースの使用を注意深く管理しなければならない。アプリケーションは保障されたリソースのセットを与えられ、システムはスケジューリングや強制終了によって、保証された以上のリソースが使用されないようにする。予約や共有型のリソース制限を利用したシステムの例としては [4] があげられる。しかしこの場合、カーネルを改造しなければならない。

この予約や共有型のリソース制限はリアルタイムやマルチメディアアプリケーションに関してはうまくスケジューリングすることができる。それは、簡単な固定値で表せる、比較的静的に決まるリソース要求があるからである。例えば、マルチメディアアプリケーションのストリーミングではビデオを表示し続けるために満たされなければならない時間制限がある。この種のアプリケーションでは多数のリクエストが同時に処理されることを保証するよりもリソースが利用可能であることを保証するほうが重要である。さらに、これらのシステムはプロセスやセッションに対するリソース割り当てに焦点をあてている。インターネットサービスでは、特定のリクエストに関して保証するよりは多数のリクエストに対して統計的な性能目標を達成するほうが望ましい。

これらリソース制限の基本的な問題は、動的な環境における理想的なリソースの制限を前もって決定するのは不可能であるという点である。制限値を低くしすぎると、資源が十分に利用されず、高くしすぎると、過負荷による深刻な性能の低下を起こす恐れがある。Web サーバへの接続数を制限する例に戻ると、正しい制限値というのは負荷に大きく依存する。静的な Web ページへのアクセスであれば、数千の接続を受け入れることができるであろうし、リソースを多量に使用するコンテンツでは、可能な接続数は数十に制限されるかもしれない。また、関連する別のプロセス、例えば、関連する処理をする Web アプリケーションサーバやデータベースが同一ホストで動作する場合、その負荷による影響も考慮する必要がある。これもやはり、静的に決められた制限で、理想的な制限を行うのは不可能である。

サービス劣化

リクエストを拒否してしまうのではなく、過負荷時にはより少ない資源しか必要としない処理を選択して、クライアントにより低い精度のサービスを配信する。この手法を利用する過負荷管理のポリシーをサービス劣化と呼ぶ。

最も単純なサービス劣化は、クライアントに配信される画像の圧縮解凍の質を下げるなど、静的な Web コンテンツの質を下げることである。多くの場合、画像の質を下げるのはサーバのネットワークで大域幅の消費を減らすためであるが、メモリの節約など他の効果もある。

サービス劣化の他の例は、すべての Web ページをより軽量な Web ページと取り替えることである。これは、2001 年 9 月 11 日に世界貿易センターとペンタゴンにテロリストが攻撃をした際、CNN.com でサーバが過負荷状態となり使用された方法である。CNN では一面を単純な HTML のページに取り替えた。しかし、これはシステム管理者によって手動で行われた非常手段であり、負荷に対応して自動でサービスの劣化が行われたわけではない。また、すべてのサービスについて、このサービス劣化を効果的に利用するのも難しい。

フィードバックを利用した集中制御

サービス設計の重要な要素に過負荷の防止があり、いろいろな形式がある。広く使用されているのは、接続数を制限するなどの静的なリソース制限に基づいた上記の方法である。しかし、サーバにかかる負荷は、分散コンポーネントの場合であればリモートメソッドの組み合わせや、関連する他のプロセスに依存するので、各リモートメソッドの最大並列度をどの程

度に設定すべきか静的に決定するのは難しい。

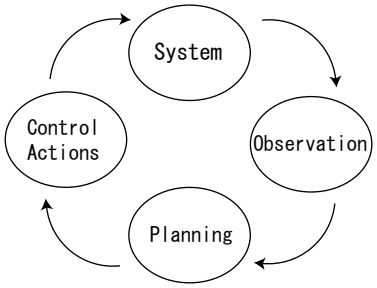

図 2.5: フィードバックのループ

したがって、実行時にサーバの情報を利用して制限を動的に調整するフィードバック制御が望ましい。フィードバック制御とは、「フィードバックによって制御量を目標と比較し、それらを一致させるように操作量を生成する制御」である。即ち、制御した結果を目標とする値と比較して、目標と結果が一致するまで反復して制御を繰り返す閉ループシステムのことである。この閉じたループは、図 2.5 のようになっており、System で実際の処理を、Observation でその処理結果であるレスポンスの観察を、Planning で観察された値の分析を、Control Actions で分析の結果から必要な設定の変更を行う。

分散コンポーネントで、最高性能を引き出すように各リモートメソッドの最大並列度を調節するには、関連するすべての処理のフィードバックから、各リモートメソッドの最大並列度を算出し、適用するというのが理想的である。しかし、そのように一箇所ですべてのフィードバックを把握して最大並列度を集中制御する場合、リモートメソッドの増加とともに最適な最大並列度を算出するのに比較対象となる最大並列度の組み合わせが膨大な数になってしまうため現実的ではない。また、同一ホスト上で関連した処理をする他のプロセス、例えばデータベースなどが動作している場合、それらのフィードバックを得ることができないため、すべての関連処理の進捗を把握すること自体できないことになる。

したがって、すべてのリモートメソッド進捗から最高性能を引き出す最大並列度を算出するというのは、理想的ではあるがあまり現実的なものではない。そこで我々は集中制御ではなく、フィードバックを利用してより制御しやすい局所的な制御を行うスケジューリング方法を提案する。

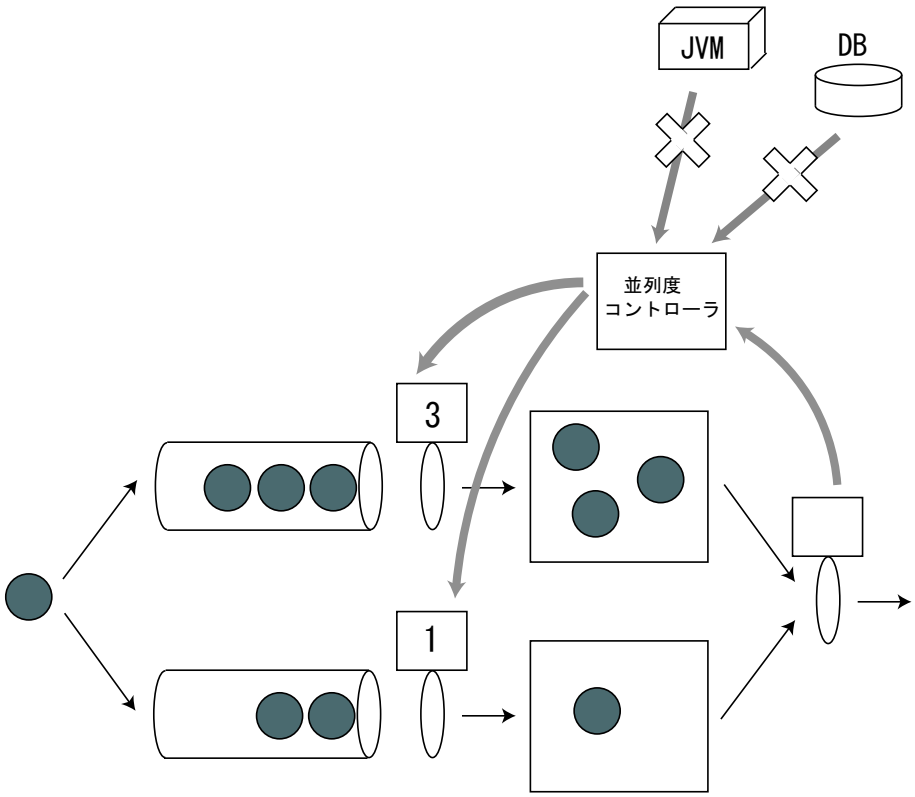

図 2.6: フィードバックによる集中制御

2.2　関連技術

本研究で提案する制御法を、Java の分散コンポーネントである EJB でリモートメソッド呼び出しに使用されている Java RMI 用の API として実装した。ここでは、この Java RMI について説明をする。また、提案する制御手法で利用している progress-based regulation というスケジューリング方法について説明する。

2.2.1　Java RMI

Java には、分散オブジェクトを実現する仕組みとして RMI（Remote Method Invocation）が用意されている。分散オブジェクトとは、異なるプロセスや異なるマシン上で動作しているオブジェクトにアクセスする技術を指し、通常のオブジェクトは同じプロセス上にあるオブジェクトにしかアクセスできないが、分散オブジェクトは間にネットワーク通信を介して互いにアクセスすることができる。分散オブジェクト技術を使うこと

で、ネットワーク・プログラミングやセキュリティ・メカニズムなどの詳細を気にせずに、あたかもローカルにあるオブジェクトのメソッドを呼び出すのと同じ感覚で、リモート・マシンに存在するオブジェクトを呼び出すことが可能になる。

RMI で使用される用語

- リモートインタフェース
  java.rmi.Remote インタフェースから継承したインタフェース。他の Java 仮想マシンから呼び出されるメソッドは、ここに定義する必要がある。

- リモートオブジェクト
  リモートインタフェースを実装したクラスのオブジェクト。他の Java マシンからそのリモートインタフェースが参照可能なものに限る。

- リモートメソッド
  リモートオブジェクトに実装されたメソッド。リモートインタフェースに定義されているものに限る。

- レジストリ
  名前付けされたリモートオブジェクトの登録、検索を行うネームサーバ。サーバはリモートオブジェクトに名前を付けてレジストリに登録をし、クライアントはレジストリに対して名前でリモートオブジェクトを検索し、その参照を得る。

これらの用語を用いた RMI の概念図は図 2.7 のようになる。分散オブジェクト技術を使用することで、クライアントは、あたかも通常の Java のローカル・オブジェクトを呼び出すようにリモート・オブジェクトを呼び出し、サーバは、あたかも Java のローカル・オブジェクトを作成するイメージで分散オブジェクトを実装することが可能になる。これを実現するための基本的な仕組みは、ほとんどの分散オブジェクト技術で共通しており、歴史的にはリモート手続き呼び出し（RPC: Remote Procedure Call）に代表される分散コンピューテイング技術から引き継がれたものである。以下では、この仕組みについて解説することにする。

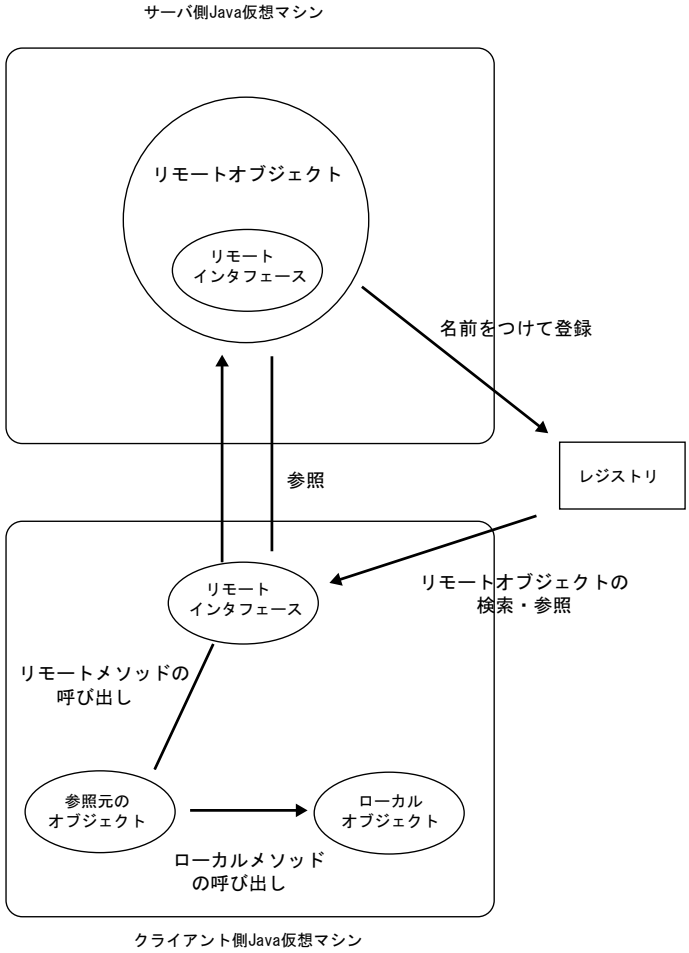

図 2.7: RMI の概念図

インタフェースの定義

アプリケーション開発者は、クライアントとサーバの実装を始める前に、サーバがクライアントにアクセスを許すオブジェクトのインタフェースを定義する必要がある。Java RMI では、Java のインタフェース定義を使用してインタフェースを定義する。

スタブとスケルトン

このようにして作成したインタフェース定義をコンパイルして、クライアント側で使用するスタブと呼ばれるコードと、サーバ側で使用するスケ

ルトンと呼ばれるコードを生成する。

- スタブの役割
  先ほど、クライアント・アプリケーションは、あたかもローカル・オブジェクトであるかのようにリモート・オブジェクトのメソッドを呼び出すといったが、ここでクライアントが呼び出していたのは、実は代理（プロキシ）オブジェクトと呼ばれるローカル・オブジェクトである。代理オブジェクトは、リモート・オブジェクトとまったく同じメソッドとパラメータを持っており、クライアントからのメソッド呼び出しをターゲットのリモート・オブジェクトに中継するという役割を持っている。

  代理オブジェクトは、必要に応じてサーバとのトランスポート・コネクションを確立し、送信パラメータのマーシャリング、要求メッセージの組み立てと送信、応答メッセージの受信と解析、受信パラメータのアンマーシャリングを行ってくれる。つまり、リモート呼び出しの詳細をクライアント・アプリケーションから隠ぺいしているのが、代理オブジェクトなのである。また、代理オブジェクトの実装コードは、インタフェース定義から生成されるスタブ・コードの中に含まれています。

- スケルトンの役割
  逆に、サーバ・アプリケーションからリモート呼び出しの詳細を隠ぺいしているのがスケルトンである。スケルトンは、クライアントから受信した要求メッセージをアンマーシャリングし、アプリケーション開発者が作成した実装オブジェクトのメソッドを呼び出してくれる。そして、メソッドの実行結果を応答メッセージにマーシャリングして、クライアントに返却する。

- マーシャリングとアンマーシャリング
  プログラミング言語や OS の違いを超えてメソッド呼び出しを実現するためには、パラメータなどのデータをこれらの違いに依存しない共通形式に変換した上で、メッセージをネットワークに流す必要がある。各プログラミング言語のネイティブなデータ表現を共通データ形式に変換する処理をマーシャリングと呼ぶ。逆に、共通データ形式をネイティブなデータ表現に変換する処理をアンマーシャリングと呼ぶ。共通データ形式は、分散オブジェクト標準ごとに規定されている。

オブジェクト・リファレンス

同一プロセス内でオブジェクトのメソッドを呼び出す場合、つまりクライアントとサーバが同じプロセスに存在する場合には、クライアントはオブジェクトに対する参照またはポインタを使用してメソッドを呼び出すことができる。プロセスやマシンをまたがってメソッドを呼び出す場合には、クライアントは同一プロセス内の代理オブジェクトを呼び出すことで、その舞台裏でリモート呼び出しが起動される。代理オブジェクトはリモート・オブジェクトの身元やネットワーク・アドレスを特定するのにオブジェクト・リファレンスを利用する。

オブジェクト・リファレンスは、分散環境でオブジェクトを一意に特定して、そのオブジェクトにアクセスするのに必要なすべての情報が格納されているデータである。どのような情報が含まれているのかは、それぞれの分散オブジェクト標準に依存するが、一般的にサーバに到達するためのプロトコルやアドレス情報、そしてサーバ内でオブジェクトを特定するための情報が含まれている。

- オブジェクト・リファレンスの所得方法
  オブジェクト・リファレンスは、オブジェクトを実装するサーバ側で作成し、クライアントはこのオブジェクト・リファレンスを JNDI (Java Naming and Directory Interface)、または簡易版のネーミング・サービスである RMI レジストリを使用して入手する。

リモートオブジェクトの構造

RMI を利用したシステムは、4 つの層から構成され、図 2.8 のようになる。

- アプリケーション層
  リモートオブジェクトへのアクセス、リモートオブジェクトのエクスポートなどを行うためのコード、つまりクライアントとサーバのコードが属す層

- スタブ・スケルトン層
  スタブやスケルトンが属する層。スタブやスケルトンはクライアントやサーバと直接やりとりし、リモートメソッドへのコールの発行、パラメータや戻り値として送られるオブジェクトのマーシャリングなどを行う。サーバとクライアント同士が直接やりとりするのではなく、このスタブとスケルトンを通してやりとりする。

- リモート参照層

リモートオブジェクトへのアクセスに関する細かい処理を行う層。オブジェクトの複製に関する処理もこの層で行う。

- トランスポート層
  コネクションの確立、マシンからマシンへのデータ転送などを行う層。

図 2.8: RMI の階層

Java RMI を使用した開発手順

ここで、Java RMI を使用したアプリケーションの開発手順について説明する。

- インタフェースの定義
  Java RMI のアプリケーションを作成する場合、まず Java インタフェースでオブジェクトのインタフェースを定義する。java.rmi.Remote インターフェイスを拡張し、各メソッドは java.rmi.RemoteException を投げる。

```
import java.rmi.Remote
import java.rmi.RemoteException;

public interface Hello extends Remote{
    public String sayHello() throws RemoteException();
}
```

- サーバ・プログラムの作成
  次にこのインタフェースをサポートするサーバ・クラスを作成する。

Hello インタフェースの各メソッドにアプリケーション・ロジックを実装する。

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class HelloImpl() extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException {
        super();
    }
    public String sayHello() throws RemoteException() {
        return "Hello World";
    }
}
```

- メイン・プログラムの作成
  さらにメイン・プログラムも作成する。メイン・プログラムでは、リモート・オブジェクトの作成とオブジェクト・リファレンスの登録を行う。オブジェクト・リファレンスの登録には、JNDI または RMI レジストリのいずれかを使用する。

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class Server {
    public static void main(String argv[]) {
        try{
            //RMI を利用可能にするセキュリティマネージャの設
定
            if(System.getSecurityManager() == null)
                System.setSecurityManager(new RMISecurityManager());
            //レジストリサーバに登録するリモートオブジェク
トの作成
            HelloImpl obj = new HelloImpl();
            //レジストリサーバに登録
            Naming.rebind("rmi://localhost/helloObj",obj);
        }catch(Exception e){
            e.printStackTrace();
        }
```

```
    }
}
```

- スタブとスケルトンの作成
  上記で作成したサーバの実装クラスから、rmic コンパイラを使用してスタブとスケルトンコードを生成する。Java RMI では、インタフェース定義からではなく実装クラスからスタブとスケルトンを生成する。また、rmic コンパイラの引数で、JRMP を使用するのか RMI over IIOP を使用するのかを選択することができる。

- クライアント・プログラムの作成
  クライアント・プログラムでは、JNDI または RMI レジストリを使用して、リモート・オブジェクトのリファレンスを取得する。そして、リファレンスを使用してリモート・オブジェクトのメソッドを呼び出す。

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;

public class Client {
    public static void main(String argv[]) {
        System.setSecurityManager(new RMISecurityManager());
        try{
            Hello obj = (Hello)Naming.lookup("rmi://localhost/helloObj");
            System.out.println(obj.sayHello());
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

2.2.2　Progress-based regulation

progress-based regulation とは近年注目されている新しいスケジューリング手法でる。これは、プロセスの進捗状況に応じて資源の割り当てを変更する手法で、進捗状況のフィードバックを利用したスケジューリングである。

この progress-based regulation を利用した資源管理システムとして MS Manners[2] があげられる。MS Manners は重要性の低いアプリケーショ

ンへのリソース割り当てを制御するシステムで、重要性の低いプロセスとのリソース競合によって、重要性の高いプロセスの処理性能が低下させられるのを防ぐのに progress-based regulation を使用している。ここでは、リソースの競合を起こしているプロセスの性能への影響はほぼ対照的である、つまりリソースの競合によって重要性の高いプロセスの性能を低下させていれば、同じリソースを使用する重要性の低いプロセスの性能も低下させられることを想定している。MS Manners では重要性の低いプロセスの進捗を監視し、進捗の悪化からリソースの競合の発生を推測している。そして、進捗の悪化によりリソース競合が起きていると判断された場合は、重要性の低いプロセスを一時停止させることでリソースを解放し、他のプロセスに譲る。

測定されるプロセスの進捗には、様々な要因で誤差が含まれてしまう。そのため進捗を単純に比較しただけでは、誤った判断が頻繁にされてしまう恐れがある。そこで、progress-based regulation では統計を利用することで、これら誤差に対処することが重要である。

また、progress-based regulation を利用した MS Manners では、プロセスの進捗状況を監視することでリソースの競合を検知しているため、任意のリソースについて競合に対処することができる。サーバかかる負荷が予想できないような場合でも、プロセスの進捗にサーバにかかる負荷が反映されるため対処することができる。

第3章 Method-level Queue Scheduling

我々は、リモートメソッドの処理性能を向上させるために、最適な並列度を動的に設定する Method-level Queue Scheduling を提案する。Method-level Queue Scheduling では、リモートメソッド呼び出し間のリソース競合を配慮しつつ、効率的に処理するのに最適な並列度を動的に算出し、各リモートメソッドに適用する。リソース競合に配慮することにより、サーバへの負荷を抑える。リモートメソッド毎に最適な並列度は異なるので、並列度の制限はリモートメソッド単位で行う。

3.1 Method-level Queue Scheduling の概観

Method-level Queue Scheduling によるリモートメソッドの処理の流れを説明する。Method-level Queue Scheduling では、リモートメソッドを呼び出すと、まず各リモートメソッドに割り振られ、リモートメソッド毎に用意されているキューに格納される。リモートメソッド毎にキューを用意することで、各リモートメソッドによるリソースの使用状況、競合を考慮したスケジューリングを行うことができる。リモートメソッドの呼び出しはサーバに到達した順にキューに格納される。その際、優先権が設定されているホストからの呼び出しについては、一般のキューとは別に用意されている優先ホスト用のキュー (優先キュー) に格納される。実行の対象となるキューを変更することにより、実行順序を変更することができる。特に、優先ホスト用のキューにたまっている呼び出しから常に処理することにより、特定のホストからの呼び出しが優先的に処理されることになり、過負荷時でも一定のレスポンスを維持することができる。

キューと優先ホスト用のキューは、リモートメソッド毎に用意されたスケジューラによって管理され、同時に処理される呼び出しはそのリモートメソッドに設定された並列度以下に抑えられている。そして、優先キューに処理待ちの呼び出しが格納されている場合は、その優先キュー内の呼び出しが他の呼び出しに優先して処理されることになる。その際、優位性を維持するため、優先ホスト以外からのリモートメソッド呼び出しは処理が

見送られる。ただし、現在処理中のものは、そのまま処理される。

リソースの使用状況は、処理されているリモートメソッドの数や種類によって変化するので、リモートメソッド毎に用意されたモニターによって、処理の進捗状況 (スループット) が随時監視され、性能が測定される。測定結果はスケジューラに通知され、スケジューラはこの情報をもとにして、動的に並列度を変更する。

以上で述べた、Method-level Queue Scheduling におけるリモートメソッド呼び出しの処理の流れを図示すると、図 3.1 のようになる。

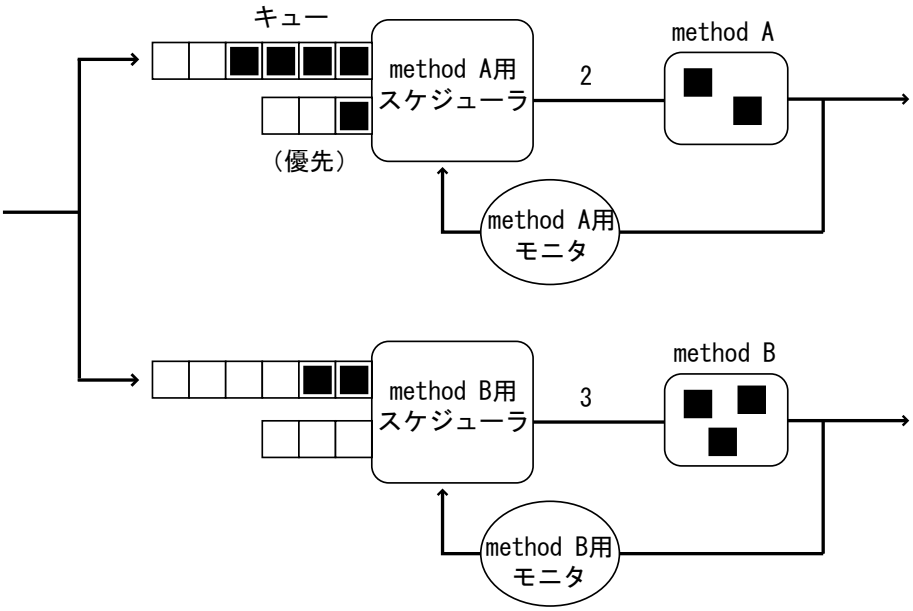

図 3.1: Method-level Queue Scheduling におけるリモートメソッド呼び出しの処理

3.2　最大並列度の動的な決定

Method-level Queue Scheduling では、リモートメソッド毎のスループットに注目し、並列度を Progress-based regulation に基づいて動的に決定する。Progress-based regulation とは、プロセスの処理の進捗状況に応じて資源の割り当てを変更するスケジューリング方法である。[2] では、重要性の低いプロセスの処理の進捗を常に監視し、進捗の遅れから、重要性の高いプロセスとのリソースの競合を推測していた。そして、重要性の低いプロセスを一時停止させることにより、競合を解消していた。RMI でも、リモートメソッド毎のスループットは、処理中のリモートメソッドの

数や消費されるリソース量に依存する。そこで、リモートメソッド毎に処理の進捗状況を測定することにより同様の情報が得られる。スループットが下がった場合には、スケジューラは最大並列度が高すぎたために何らかのリソース競合が起きたものと判断して最大並列度を下げる。

同じリモートメソッドに対する呼び出しでも、消費されるリソースは呼び出し毎に多少異なるので、スループットからリソースの消費を完全に推測することはできない。しかし、大多数のリモートメソッドでは、ほぼ一定のリソースを消費するものと仮定して、提案手法ではこのような方式を採用した。

また、スループットの単純な大小比較では、誤差による影響を受けやすく、誤った判定が頻繁に起る恐れがある。そこで、本方式では、スループットの比較をする際に、統計処理を利用している。

3.3　Method-level Queue Scheduling による効果

Method-level Queue Scheduling を利用することにより、リモートメソッドの処理において以下にあげる 4 つの効果が得られると考えている。

- リソースの有効活用
  リモートメソッド呼び出しを並列に処理することにより、スループットが上昇するリモートメソッドに関しては、最大並列度を積極的にあげる。そのようにすることで、サーバ計算機の余剰リソースを有効に活用し、サーバ全体の処理性能を向上させる。

- 競合を起こしやすいリモートメソッドのスループットを改善
  処理に大量のリソースを使用するため、呼び出しの増加により競合を起こしやすいリモートメソッドについては、低い最大並列度が設定される。その結果、従来であれば、呼び出しの増加により、スループットが低下していたリモートメソッドの処理性能が改善される。

- リソースの独占を防止
  スケジューラとキューをリモートメソッド毎に用意することで、リソースが特定のリモートメソッドによって独占されるのを防ぐ。例えば、サーバへの負荷が大きな処理をするリモートメソッドへの呼び出しが殺到した場合でも、競合により並列に処理される呼び出しの数が抑制されるため、他のリモートメソッドへの影響が抑えられる。

- ホストレベルでの優先度の設定
  優先度に応じて異なるキューにリモートメソッドの呼び出し順序が格納される。優先度の高いホストからの呼び出しについては処理さ

れる順序を早めることにより、過負荷時でも一定のレスポンスが維持される。

第4章　実装

我々は、以上に述べてきた Method-level Queue Scheduling を Java RMI 用の Java API(Application Programming Interface) として実装した。実装した API では、クライアントからのリモートメソッド呼び出しをインターセプトして、実行を制御するために必要なクラスが提供されている。

4.1　ライブラリの詳細

OpControler クラス

OpControler クラスでは、リモートメソッドの呼び出しを各リモートメソッドのキューに振り分ける機能が実装されている。リモートメソッドのメインロジック前後で OpControler クラスの before メソッドと after メソッドを呼び出すことで、キューへの振り分け、及び実行の制御が行われる。その際、メソッド名を表す文字列を引数に渡して呼ぶようにする。OpControler クラスでは、このメソッド名を利用して、呼び出しの振り分けを行う。

UniOpControler クラス

UniOpControler クラスでは、単一のリモートメソッドを管理するために必要なキューとスケジューラが実装されており、制御対象となるリモートメソッド毎に UniOpControler クラスのインスタンスが生成される。OpControler クラスでの呼び出しの振り分けとは、各リモートメソッドに対応した UniOpControler クラスのインスタンスに制御を渡すことである。UniOpControler クラスには、それぞれのリモートメソッドで同時に処理することのできる数 (並列度) を表す変数が保持されている。リモートメソッドを呼び出した際に、実行中のスレッド数が並列度に達していた場合には、スケジューラが wait メソッドを呼び出して、一時停止させる。リモートメソッドを呼び出した際の順序を保つため、各リモートメソッドの呼び出しで、異なるオブジェクトで wait メソッドを呼び出し、待機中にしている。このオブジェクトをキューに格納し、待機状態に入った順序で、

つまり、キューの先頭から順に notify メソッドを呼んでスレッドに通知することにより、実行順序を制御している。

優先するホストからのリモートメソッド呼び出しであるか否かは、java.rmi.server.RemoteServer クラスの getClientHost メソッドにより得られる、送信元の情報を利用している。

Binomial クラス

Binomial クラスでは、二項検定を行うのに必要なライブラリが実装されている。本方式では、並列度の決定を行う際に符合検定を利用しており、単一リモートメソッドの管理をする UniOpControler クラスの内部で利用されている。

4.2　並列度の決定

本方式では、並列度を最適なものに設定するために、並列度を常に変動させるという方針をとっている。現在の並列度が、現在のサーバの状態でリモートメソッドにとって最適なものであるか否かは、同時点での他の並列度の場合との比較によって決まる。そのため、常に並列度を変動させるという方針をとっている。

基本的には、ある時点で並列度を増加・減少させ、計測されるスループットの変動からその変更による効果があると判断できる場合、その変更を行い続けるという戦略をとる。例えば、並列度を増加させることによりリモートメソッドのスループットが増加した場合、並列度をさらに増加させて、呼び出しを処理しながらスループットを測定する。並列度の増加、減少の判断は、具体的には次のようにして行われる。

4.2.1　並列度変更のアルゴリズム

大小比較

現在の並列度、以前の並列度ではどちらの方が適切か、つまりスループットが良かった判定するのに、スループットの比較をする必要がある。単純に大小比較をしただけでは、計測値に含まれる誤差による影響を過度に受けてしまう。そのため、検定を利用する。測定値がどのような分布に従うかは不明なので、分布に関する情報を必要としない符合検定を本方式では採用している。符合検定を用いた大小比較では、比較対象より大きい、小さい、判断できない、の 3 通りの判定結果がある。判断できない

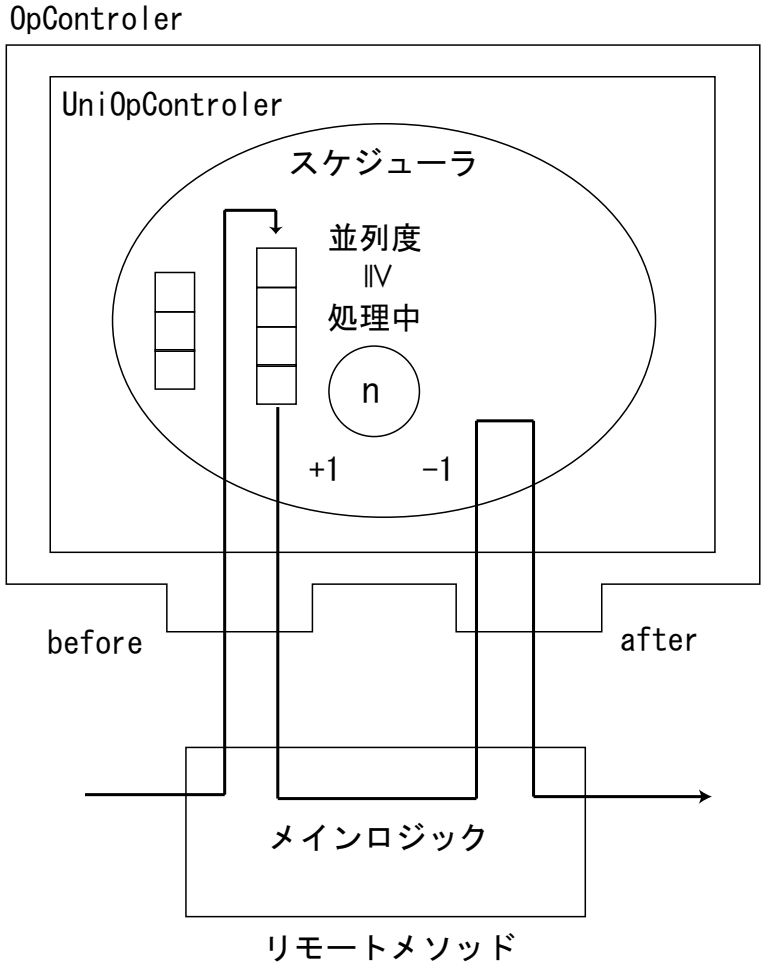

図 4.1: 実装の概念図

というのは、判断するにはデータが少ないとみなし、再度スループットを計測して判定を行う。本方式では、並列度を常に変動させるという方針をとっており、十分なデータが得られるまで、一定の並列度に固定されるのは望ましくない。そこで、一定回数を越えても大小関係が決まらない場合は、現在の並列度に変更してから得られたデータの平均値と前回の並列度のスループットとで単純な大小比較をして判定をすることにする。

並列度の変更

現在の並列度と、前回の並列度ではどちらが最適か判定するのに十分なデータが得られたら、以下の方法で並列度を変更する。判定が得られると

は、検定により、大小の判断が得られるか、一定回数を越えてしまい、平均値により判定をする場合のことをいう。

記述を簡潔にするために、スループットの大小関係を単純な不等号で表す。ある時点 T(n) における並列度を Max(n)、スループットを Throughput(n) とし、それよりも一つ前に実行した並列度を Max(n-1)、スループットを Throughput(n-1) で表す。そして、以下の条件式による分類にしたがって、並列度が変更される。

分類 1:

$$\Big(Max(n) - Max(n-1)\Big)\Big(Throughput(n) - Throughput(n-1)\Big) \geq 0$$

並列度の更新

$$Max(n+1) = Max(n) + \Delta k$$

この条件に当てはまるのは、
$Max(n) \geq Max(n-1)$ かつ $Throughput(n) \geq Throughput(n-1)$ 、
あるいは
$Max(n) \leq Max(n-1)$ かつ $Throughput(n) \leq Throughput(n-1)$
が成り立つとき。並列度が大きい方のスループットの方が良いと判断されるので、設定された増減分 Δk だけ増加させる。

分類 2:

$$\Big(Max(n) - Max(n-1)\Big)\Big(Throughput(n) - Throughput(n-1)\Big) < 0$$

並列度の更新

$$Max(n+1) = Max(n) - \Delta k$$

この条件に当てはまるのは、
$Max(n) > Max(n-1)$ かつ $Throughput(n) < Throughput(n-1)$ 、
あるいは
$Max(n) < Max(n-1)$ かつ $Throughput(n) > Throughput(n-1)$
が成り立つとき。並列度が小さい方のスループットの方が良いと判断されるので、設定された増減分 Δk だけ減少させる。

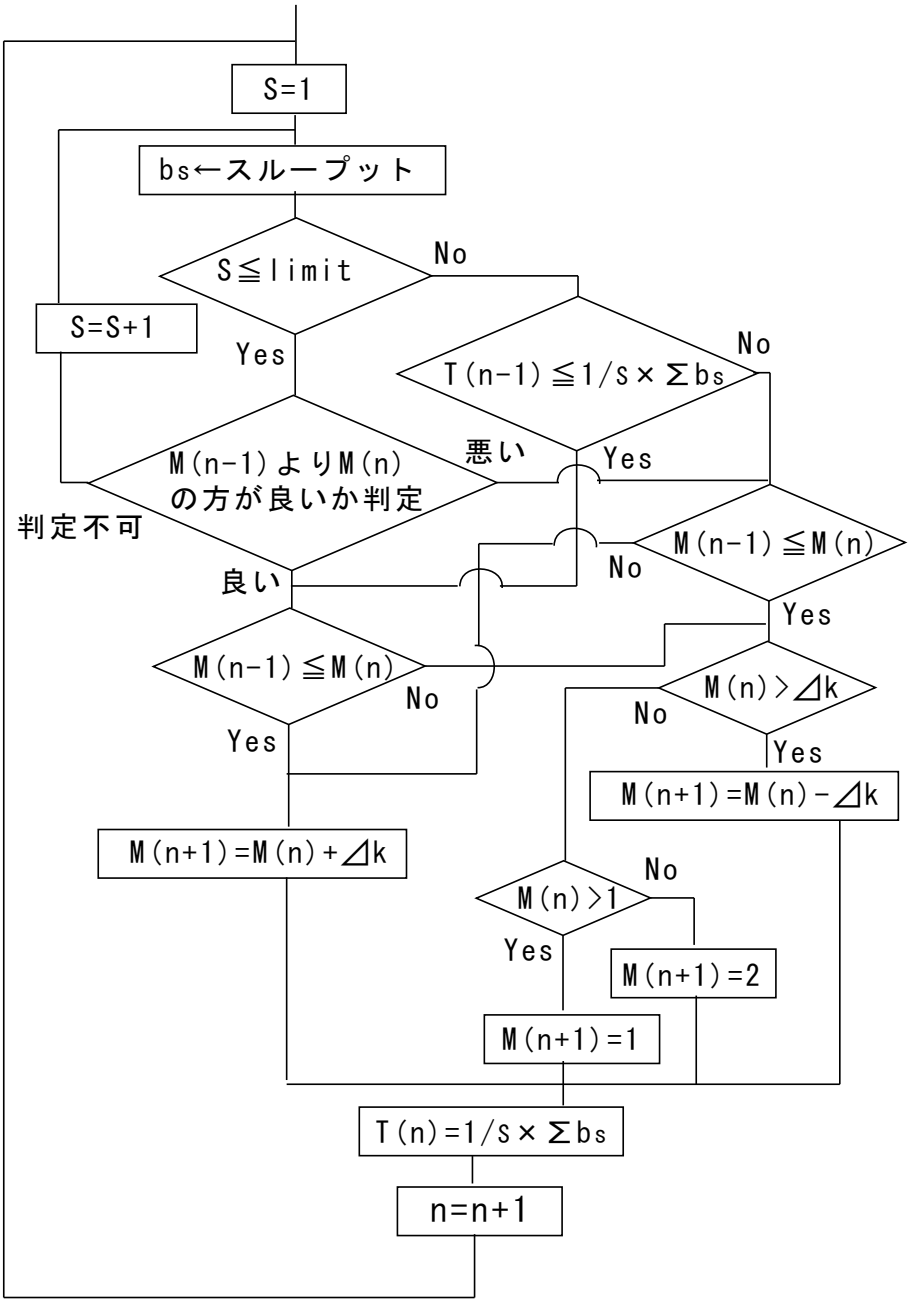

図 4.2: 並列度更新のアルゴリズム

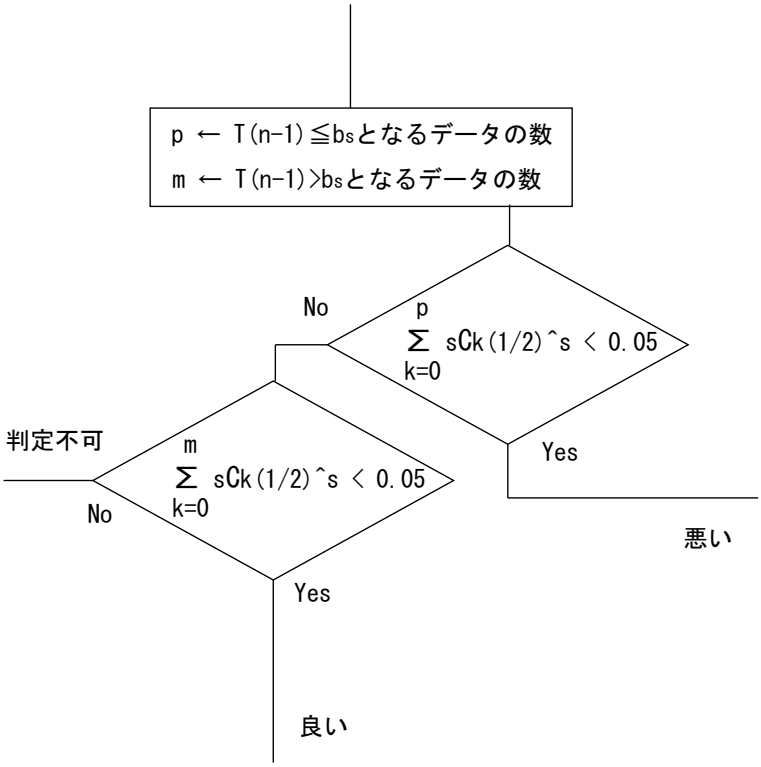
p ← T(n-1)≦bsとなるデータの数
m ← T(n-1)>bsとなるデータの数
No
p
Σ sCk(1/2)^s < 0.05
k=0
Yes
悪い
判定不可
m
Σ sCk(1/2)^s < 0.05
k=0
No
Yes
良い

図 4.3: 符合検定

第5章　実験

3 章で述べた、Method-level Queue Scheduling の効果を調べるために、4 章で述べた Java RMI 用のライブラリを使用して、複数の実験を行った。

5.1　実験環境

実験に用いた機材は次の通りである。

サーバマシン

CPU:Intel(R)Xeon(TM)CPU2.4GHz × 2
Memory:2GB
OS:Linux2.4.20
NIC:100BaseTX

クライアントマシン

CPU:Pentium 733MHz
Memory:512MB
OS:Linux2.4.19-Ovl11(VineLinux)
NIC:100BaseTX

クライアントには同様の性能を持つマシンを 15 台使用する。

5.2　負荷の大きいリモートメソッドに最適な並列度

実験内容

サーバへの負荷が大きいリモートメソッドに対応するものとして、実験では 116KB 程度の XML ファイルをパースしてデータ検索を行うリモートメソッドを利用する。そこで、このリモートメソッドについて最適な並列度 (スループットが最大となる並列度) を求めておく必要がある。測定方法としては、並列度を一定に固定するために、リモートメソッドの実装前後に以下のコードを追加する。40 のクライアントからリモートメソッドを繰り返し呼び出し、1 分間での処理数を計測する。

//メインロジックの前に入れるコード

```
synchronized(this){
  while(exe>=max){
    try{
      wait();
    }catch(InterruptedException e)
  }
  exe++;
}

//メインロジックの後に入れるコード
synchronized(this){
  exe--;
  notifyAll();
}
```

実験結果

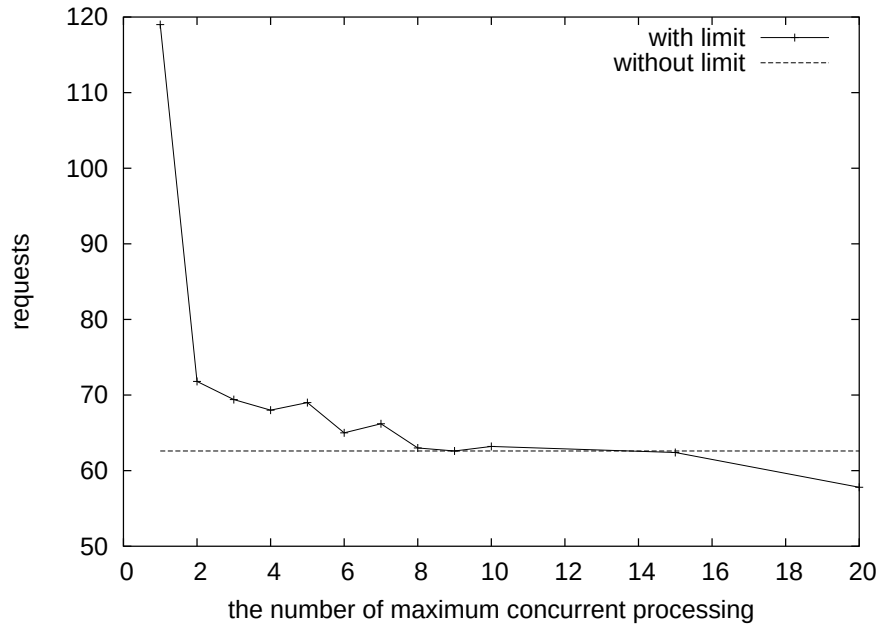

図 5.1: 並列度を固定した場合の処理数

測定結果は、図 5.1 の通りである。横軸が最大並列度、縦軸が一分間あたりのリモートメソッドの処理数になっている。横軸と平行な直線は、並

列度を固定しなかった場合の処理数である。最大並列度を 1 にした場合、処理数が最も多く、並列度の増加とともに処理数が減少している。並列度の増加とともに、資源の競合により処理性能が低下していることがうかがえる。

5.3　並列化すべき処理に関して最適な並列度

実験内容

単純な数値計算であれば、並列に処理をすることにより性能は向上する。そこで、そのような処理についても、並列度による性能の推移を調べておく。実験の対象としては、フィボナッチ数を求めるリモートメソッドを使用する。また、並列度を一定に保つために、5.2 と同様のコードを追加する。40 のクライアントからリモートメソッドを繰り返し呼び出し、15 秒間での処理数を計測する。

実験結果

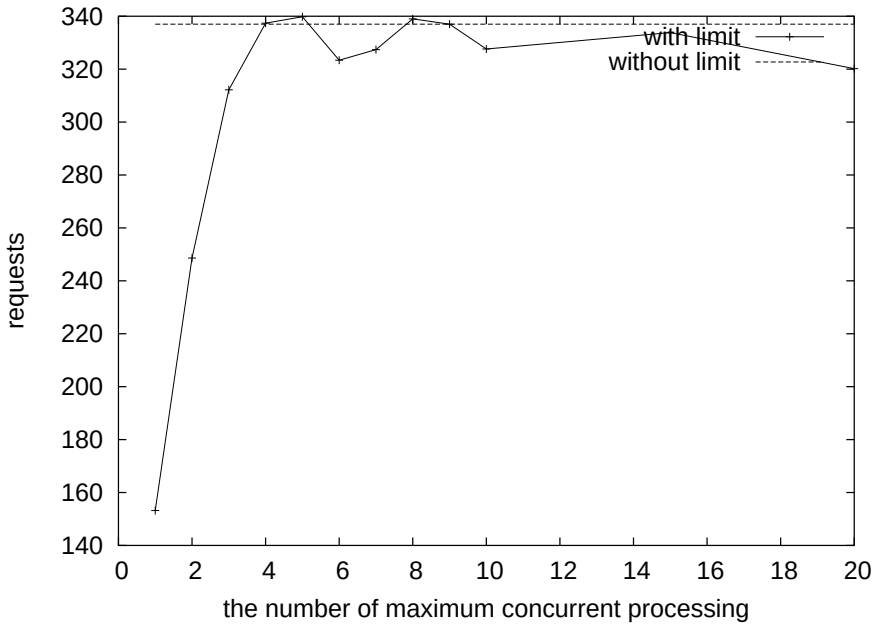

図 5.2: 並列度を固定した場合の処理数

測定結果は、図 5.2 の通りである。横軸が最大並列度、縦軸が 15 秒間

あたりのリモートメソッドの処理数になっている。横軸と平行な直線は、並列度を固定しなかった場合の処理数である。並列度が 5 以上では、それ以上の向上はみられない。

5.4　並列度の変異

実験内容

サーバへの負荷が大きいリモートメソッド、及びリクエスト処理の並列化により性能が向上するリモートメソッドに本方式を適用した際の、並列度の変化をみる。負荷の大きいリモートメソッドに対応するものとして、5.2 で使用したリモートメソッドを、並列化により性能向上するリモートメソッドに対応するものとして、5.3 で使用したリモートメソッドを用いる。40 のクライアントからリモートメソッドを繰り返し呼び出し、スループットが計測された時点での並列度を記録する。

実験結果

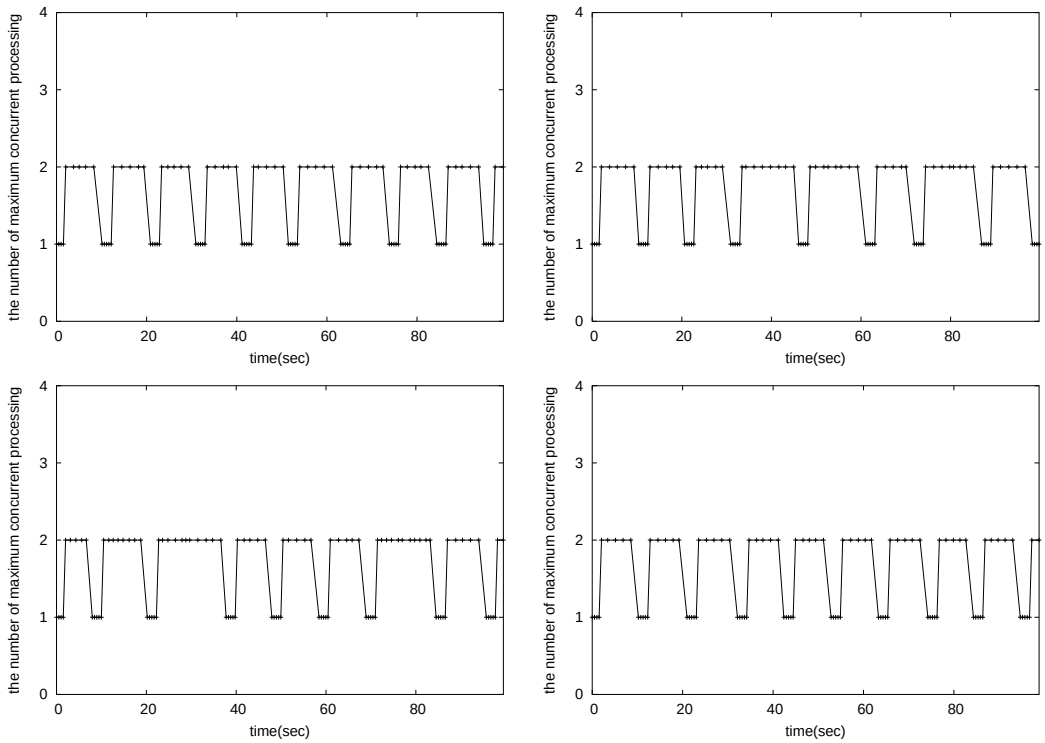

図 5.3: 並列度の変化

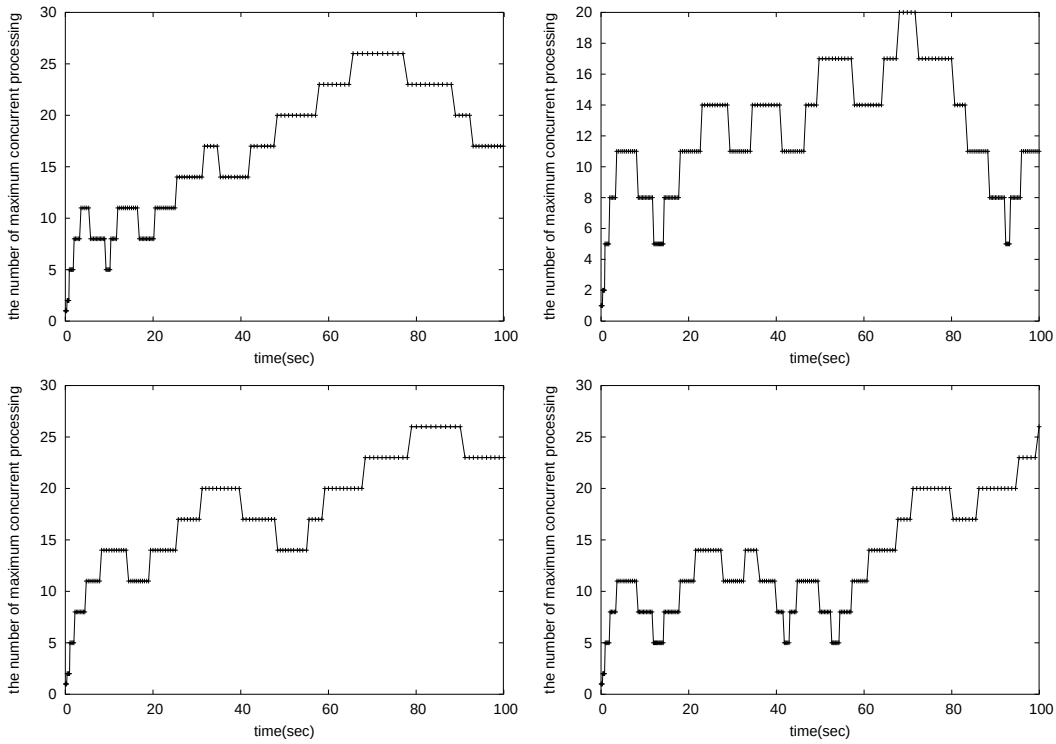

図 5.4: 並列度の変化

サーバへの負荷が大きい処理については、図 5.3 に、並列化により性能が向上する処理については、図 5.4 に測定結果を示す。それぞれについて、測定を 4 回行っている。

考察

サーバへの負荷が大きい処理については、並列度が 1 と 2 の間を変動している。5.1 での測定結果で、最適な並列度は 1 となっており、並列度が 2 になるとスループットが急激に低下していたので、図 4.2 が機能することにより、並列度の増加が抑えることができている。

スループットの測定回数に注目すると、並列度を変更する際に、最低でも 5 回行われている。これは、一つ前の最大並列度でのスループットとの大小判定を行う際に、「前回の最大並列度よりも現在の最大並列度の方がスループットは高い」と「現在の最大並列度よりも前回の最大並列度の方がスループットは高い」を帰無仮説として、それぞれ検定をしているためである。どちらかの帰無仮説を棄却して、5%の棄却領域に入るためには最低で 5 個のデータを必要とする。

Method-level-Queue-Scheduling を利用することにより最大並列度は確

かに抑えられてはいるが、スループットの測定回数は最大並列度が 1 の場合と 2 の場合でそれほどの違いがない。同じ計測 1 回でも並列度 2 の場合の方が計測に時間がかかるため、全体的に、スループットの悪い並列度 2 での処理時間の方が長くなってしまっている。これは、最大並列度を 1 にした方が性能はよくなるという過去のデータを利用して、最大並列度に重み付けを行い、重みが高く最大並列度での判定を遅らせるせることにより緩和できると思われる。

また、スループットの大小比較の結果を得るのに最低でも 5 個のデータが必要であり、そのぶん判定に時間がかかってしまっている。棄却領域をより広くすることにより早く判定できるようにするという方法も考えられる。

並列化により性能が向上するリモートメソッドについては、サーバへの負荷が大きい処理の場合と比較するとあまり安定はしていないが、最大並列度がだいたい大きな値で、少なくとも 5.3 で得た 5 よりは大きな値で維持されている。しかし、同時に処理されるリモートメソッドの処理の数が最大並列度に達しない限りスループットの測定が行われないため、大並列度の増加とともに、判定が出るまでの時間が増加している。最大並列度及び処理時間に応じて測定回数の上限を減らすなどして判定時間を短縮し、サーバの負荷による動的な調節をよりスムーズに行えるようにする必要がある。

5.5　競合を発生するリモートメソッドの性能改善

実験内容

サーバへの負荷が大きいリモートメソッドについて、本方式を利用することによる効果をみる実験を行う。負荷の大きいリモートメソッドに対応するものとして、5.2 で使用したリモートメソッドを用いる。そして、最大 40 のクライアントが同時にリモートメソッドを呼び出し、その総処理時間を、本方式を利用した場合と利用しない場合それぞれについて計測する。

実験結果

サーバへの負荷の大きい処理について、本方式を用いた場合と用いない場合の総処理時間は図 5.5 の通りである。最大 27%程度総処理時間が短縮されていることがわかる。

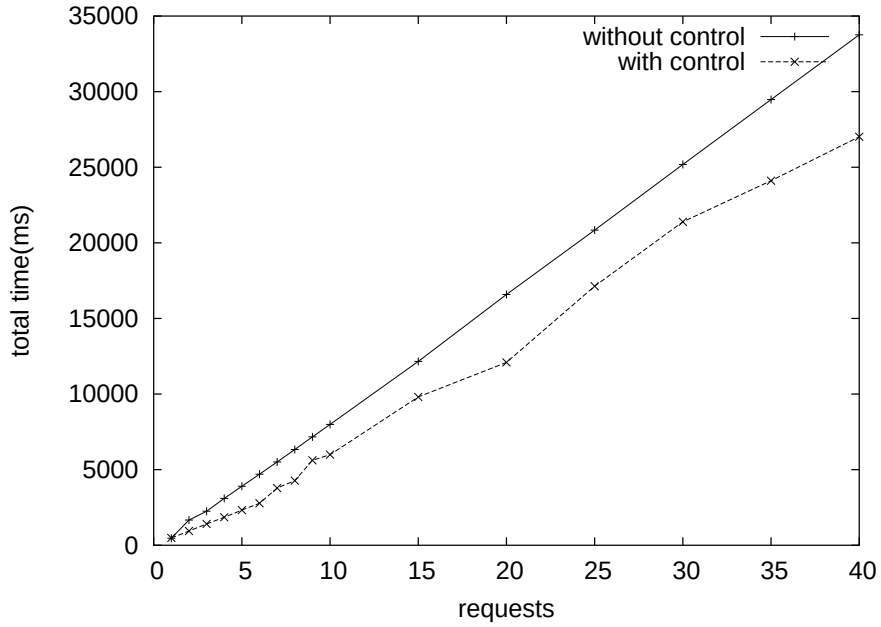

図 5.5: Method-level Queue Scheduling 利用による競合時の性能改善

考察

この結果から、単一のリモートメソッドに対する呼び出しの並列処理について、本方式を利用することにより、リモートメソッド呼び出し間で発生するリソースの競合がある程度解消され、性能の劣化が改善されていると思われる。

5.6　リソースの平等な振り分け

実験内容

サーバへの負荷が大きいリモートメソッドの呼び出しが急激に増加した際に、負荷の小さい軽いリモートメソッドの処理が滞ってしまうことがある。この実験では、同様の状況として、負荷の小さいリモートメソッドを常時 20 のクライアントから呼び出し、開始 10 秒経過したところで、常時 50 のクライアントから負荷の大きいリモートメソッドの呼び出しを行う。そして、0.1 秒間隔で処理された数を各リモートについて測定する。負荷の大きいリモートメソッドは、5.2 で使用したリモートメソッドを、負荷の小さいリモートメソッドには数値計算を少しだけ行うリモートメソッドを用いる。測定は、Method-level Queue Scheduling を使用した場合とし

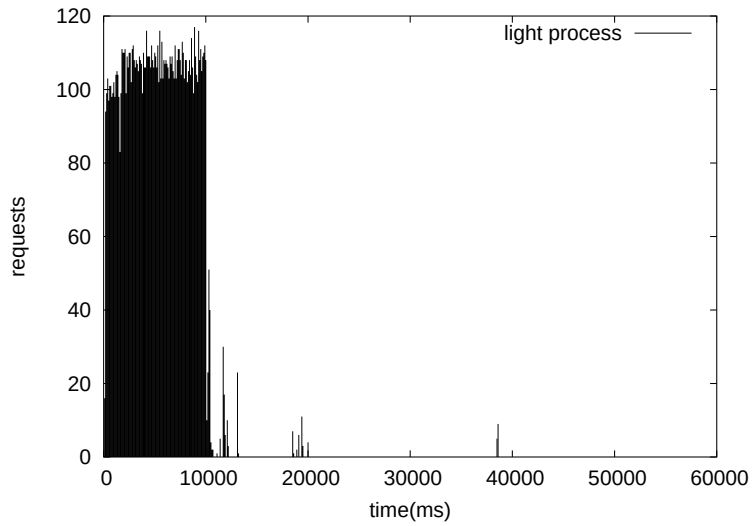

図 5.6: Method-level Queue Scheduling 未使用時の軽いメソッドの処理頻度

ない場合、それぞれについて行う。

実験結果

負荷の小さい軽いリモートメソッドについて Method-level Queue Scheduling を使用しなかった場合の実験結果は図 5.6、負荷の大きな重いリモートメソッドについて Method-level Queue Scheduling を使用しなかった場合の実験結果は図 5.7 になった。縦軸は 0.1 秒当たりの処理数である。重い処理が開始される 10 秒までは、軽い処理では一定の処理数が保たれているが、重い処理が開始されると急激に悪化している。重い処理の方は、開始されると一定の割合で処理されている。

対して、Method-level Queue Scheduling を使用した場合の軽い処理は図 5.8 のようになった。重い処理が開始される 10 秒までは、Method-level Queue Scheduling を使用した場合と同程度の処理数が保たれている。そして重い処理が開始されても、極端な処理の悪化はせずにある程度の処理数が維持されている。反面、重い処理については図 5.9 の通り処理数が減ってしまっている。

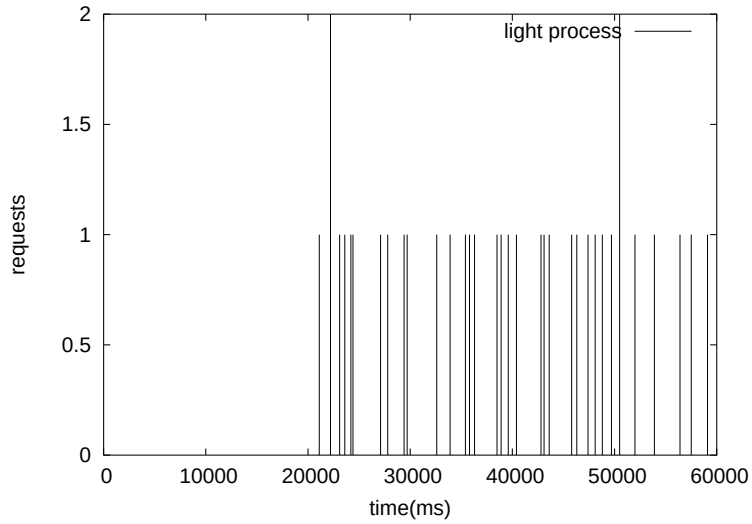

図 5.7: Method-level Queue Scheduling 未使用時の重いメソッドの処理頻度

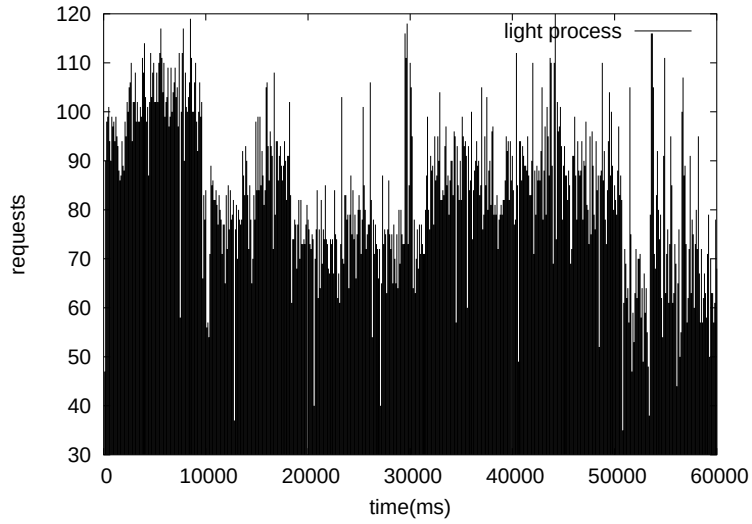

図 5.8: Method-level Queue Scheduling 使用時の軽いメソッドの処理頻度

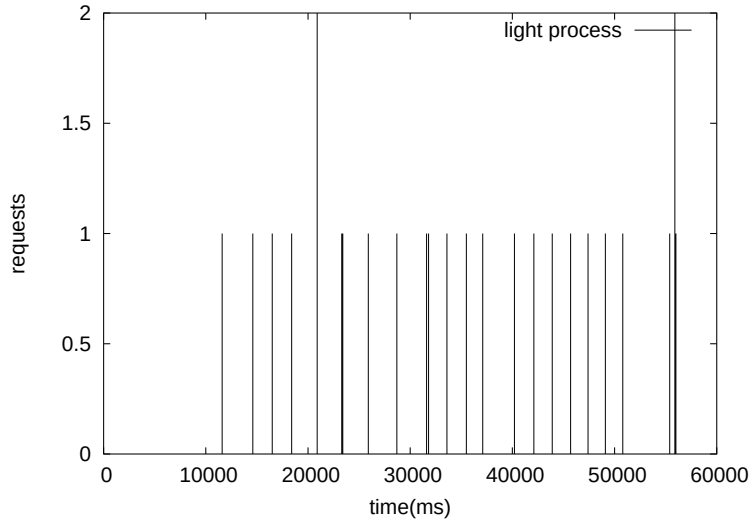

図 5.9: Method-level Queue Scheduling 使用時の重いメソッドの処理頻度

考察

Method-level Queue Scheduling を利用することにより、サーバへの負荷が大きい処理によって圧迫されることなく、サーバへの負荷が小さい処理を続けられる。これは、重い処理の最大並列度が低く保たれているからである。しかし、そのため実行状態にあるスレッドの数は重い処理によりも軽い処理の方がかなり多くなってしまうため、CPU 時間の多くは軽い処理をするリモートメソッドで消費されていることになる。その結果、負荷の大きい処理については、Method-level Queue Scheduling を使用しない場合に比べ処理数がかなり少なくなっている。したがって、サーバへの負荷が大きな重い処理の方を優先したいような場合には、Method-level Queue Scheduling の利用は適さないことになる。処理によっては極端に性能が悪くなってしまうことも考えられるので、リモートメソッド毎に優先度を付けることにより、そのような場合に対処する必要がある。

5.7　優先度利用による効果

実験内容

特定のホストからのリクエストを優先的に処理することにより得られる効果をみる。まず、5.2 で使用したサーバへの負荷が大きいリモートメソッドを常時 40 のクライアントから呼び出し、呼び出しが殺到している

状況をつくる。この状況下で、同じリモートメソッドを優先権を持つホストと、優先権を持たないホストから呼び出し、そのレスポンス時間を測定する。測定はそれぞれ、10 回ずつ行う。

実験結果

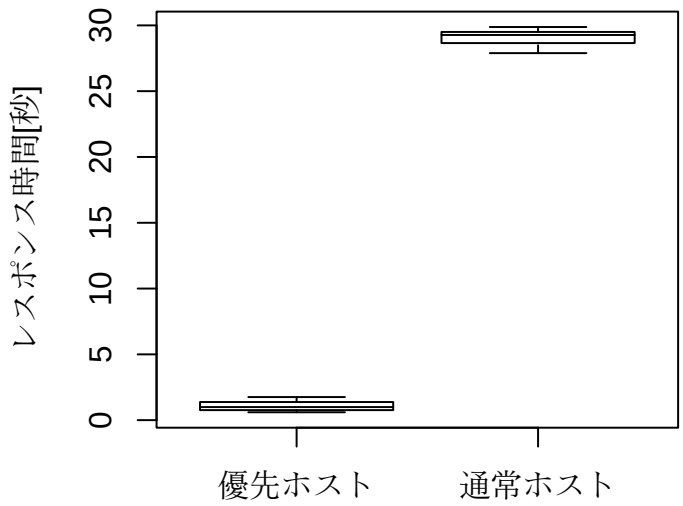

図 5.10: 優先ホスト利用による効果 (単位:ms)

レスポンス時間の測定結果は図 5.10 の通りである。ここでは、箱ひげ図を利用している。優先権を与えることにより、与えられていない場合に比べ平均して 1/27 のレスポンス時間に短縮されていることがわかる。

考察

サーバへの負荷が大きい処理に関しては、優先権を与えることにより、過負荷の状況下でもレスポンス時間をある程度保障することは可能だといえる。サーバへの負荷が大きい場合、5.3 で示されたように並列度が小さい値に抑えられる。そのため、キュー上で待機状態にある多数のスレッド

に優先して処理されている。しかし、サーバへの負荷が小さい処理の場合は、最大並列度はあまり制限されないため、負荷が大きい処理の時ほどの効果は得られない。

第6章　まとめ

本研究では、分散ソフトウェアにおける過負荷時の性能劣化を改善する手法として、Method-level Queue Scheduling を提案した。Method-level Queue Scheduling は、特に、分散コンポーネント間で相互の通信に利用されているリモートメソッド呼び出しに焦点をあてた手法で、リモートメソッドで個別に自身のスループットのフィードバックを利用して、より高いスループットが得られるように並列処理数を調節する。

Method-level Queue Scheduling ではすべての関連する処理の進捗を一括管理する集中制御でなく、単一のメソッドの進捗しか観察しない。実験により、このようなスケジューリング方式でもサーバへの負荷が高い処理による全体的な処理の低下を防ぎ、リモートメソッドの性能が向上させられることを確認した。さらに、優先度を設定することにより、特定のホストからのリモートメソッド呼び出しの処理時間を短縮され、レスポンス時間がある程度保証されることを確認した。

6.1　今後の課題

以下に今後の課題を列挙する。

- Method-level Queue Scheduling では、性能の向上、低下を検定により判断しているため、データが集まるまでに時間がかかり、最適な並列処理数に到達するまでにかなりの時間がかかってしまう。理由としては、棄却領域を 5%にしていることで最低でも 5 個のデータが必要であること、最大並列度の増加にともないスループットの測定に時間がかかってしまうことがあげられる。最大並列度に応じて、採取するデータの数を変更する仕組みを取り入れる必要がある。
- 高負荷な処理での並列処理数の変動をみる実験において、最適な並列処理数での処理時間が全体的に短くなっていた。これは、過去の測定結果の有効利用ができていないからである。並列処理数の推移から、並列処理数に重み付けを行うなどの対策をとる必要がある。
- 本研究の実験で用いたベンチマークはとても単純なものであり、実際のサーバへの負荷を想定した実験とはいえない。リモートメソッ

ドの種類を増やし、呼び出される間隔をいろいろと変えて行う必要がある。

参考文献

[1] The apache software foundation. `http://www.apache.org`.

[2] John R. Douceur and William J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 247–260. ACM Press, 1999.

[3] Jboss. `http://www.Jboss.org/`.

[4] David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 153–167. ACM Press, 1996.