修　士　論　文

HORB を用いた分散計算と感染伝搬の分析

指導教官　　林　幸雄　助教授

北陸先端科学技術大学院大学
知識科学研究科　知識システム基礎学専攻

350040　丹野　聖司

審査委員：林　幸雄　助教授（主査）
池田　満　　教授
佐藤　賢二　助教授
橋本　敬　　助教授

2005 年 2 月

目次

第1章 はじめに

1.1 研究の背景

近年さまざまな技術革新に伴い，処理速度や記憶容量などに関して計算機の性能は飛躍的に向上してきたといえる．計算機性能の向上によって，単体の計算機で行える処理の幅は大きく広がってきたといえるが，現在の計算機の性能を用いても計算するのに非常に時間がかかる高負荷な問題を取り扱う分野，例えば，惑星の軌道計算や遺伝子の解析，新薬の開発などでは，いまだにスーパーコンピュータを使用しなくてはならない状態にある．しかし，スーパーコンピュータは開発や維持，管理に非常に多くのコストがかかり，容易に使用することができないのもまた現状である．

こういった高負荷な問題を扱う分野において，近年注目を集めているのが分散処理である．分散処理とは，複数台の計算機を局所的にネットワークで接続した環境を作成し，擬似的に超高性能計算機を実現する手法である．その際，高負荷な問題を細分化し，複数台の計算機に細分化した負荷を計算させることによって，低コストで高速に問題を処理しようとするものである．さらに，近年のネットワーク技術の発展と，環境の整備により，多くの計算機資源を有効に活用するという観点から，Grid コンピューティングといった，より広域的な分散処理の研究が盛んに行われ，実現されてきている[1]．また，ソフトウェアにおいても，ネットワーク間でオブジェクトを共有させ，かつネットワークを透過的に扱うための分散オブジェクトが整備されてきた．

分散処理の分野は近年のさまざまな技術革新の恩恵を受け発展を遂げてきているが，さまざまな問題点を抱えている．その 1 つとして，処理機構の一貫性の保持が挙げられる．複数台の計算機に対してある負荷を割り当てる際，その負荷を処理するプログラムやソフトウェアをすべての計算機に配置しなくてはならない．また，それらの配置は計算機の処理内容の一貫性を保持するために，処理内容の変更やバージョンアップのたびに行わなくてはならない．このような作業は，計算機の台数が多くなれば多くなるほど非常に大変な作業になる．

また，負荷の分散方法も大きな議論点となっている．負荷分散手法とは，システム内の負荷量を調節することによって，効率的に分散処理を行おうとする

ものである．既存の負荷分散手法には，システム内の負荷をできるだけ均一にする手法などさまざまな手法が研究されてきている[2]．しかし，これらの負荷分散手法をおこなうためには，システム内の負荷を継続的に観測する必要があり，この制御自体がボトルネックになってしまう可能性がある．

このように，分散処理はいまだに発展途上の技術であるが，今後も非常に重要になる技術であるということがわかる．

1.2 研究の目的

前節のような背景の中，本研究では，Java 言語と，Java 用分散オブジェクト HORB を用いて，クラスファイルの動的取得機能を備えた，マスタースレーブ方式の分散処理環境を構築する．本研究では，ランダムな大規模ネットワークモデルの作成と，一般的に膨大な計算量となるネットワーク上のモンテカルロ的なウィルス伝搬シミュレーションを考え，分散処理環境におけるそれらの性能を評価すると共に，ネットワーク上でのウィルス伝搬特性を解析することを目的とする．

1.3 論文の構成

本論文の構成は以下のとおりである．

- 2 章：本システムを構築知るために使用した技術である Java 言語，Java 用分散オブジェクト HORB について説明する．
- 3 章:本システムの構成を, ハードウェアとソフトウェアの双方から解説し，本システムの備えるクラスファイルの動的取得機能について解説する
- 4 章：本システムで作成するネットワークモデルの説明と，ウィルスの伝搬方法を規定したウィルス伝搬モデルについて説明する．
- 5 章：本システムで実際に作成したネットワークモデルとウィルス伝搬シミュレーションの詳細について説明する．また，システム全体で処理するタスク数を求め，分配するタスクの説明を行う．
- 6 章：本システムを用いて行った実験とその結果をまとめる．
- 7 章：本論文の結論と考察，そして今後の課題について述べる．

第2章　使用技術について

本章では, 分散処理環境を含む全システムの開発に使用した Java 言語と分散オブジェクト技術 HORB について述べる.

2.1　Java 言語

本システムの開発は全て Java（jdk1.5.0）を用いて行った．その理由としては，マルチプラットフォーム対応によって使用 OS を考慮する必要がない，CORBA や RMI，HORB といった分散オブジェクトを使用することができる，ガーベージコレクションなどといった Java の持つ有用な特徴が挙げられる [3]. 本節では，これら Java の持つ特徴が，本システムの開発にどのように適しているかを示す.

2.1.1　マルチプラットフォーム対応

Java は開発当初，汎用的なプログラミング言語であり，Web のための言語ではなかった．しかし現在では，Java でソフトウェアを構築した場合，プログラムの変更やコンパイルのやり直しを必要とせず, 主要な OS でそのまま利用できるという特徴を持つ言語になっている.

このマルチプラットフォーム対応という特徴は，分散処理環境を考える上で非常に重要なものであるといえる．例えば，高負荷の処理を地理的に分散した複数の計算機で分散処理しようとした場合, 各計算機のスペックや OS が必ずしも同一であるという保証はない．そういった場合，Java を使用することによって, 各計算機の OS を考慮することなく, 必要なプログラムを開発することができる.

6 章で述べる具体的な実験における分散処理環境は, マスタースレーブ方式の局所的な分散処理を基本とするため，管理者が全てを管理することができ，スレーブマシンは全て同一の OS で構築されている．しかし, 今後さらなる計算機の増台や，地理的に分散した計算機を使用する分散処理への発展を考えた際, やはりマルチプラットフォーム対応の Java が開発に適していると考えた.

2.1.2　分散オブジェクトの使用

ネットワーク型の分散システムの構築といった場合，CORBA（Common Object Request Broker Architecture）が一般的に知られている．CORBA は ORB（Object Request Broker）を用いて作成され，ORB とインターフェース定義言語 IDL（Interface Definition Language）によって多くのプログラミング言語に対応した分散オブジェクト技術の一つである．また，Java にはリモートサーバ上の特定のメソッドを呼び出す RMI（Remote Method Invocation）が用意されている．

このように Java は，今回使用した HORB を含め，さまざまな分散オブジェクトに対応しておりにも，将来的な機能の追加などを考慮した際のシステム開発に適していると考えた．

2.1.3　ガーベージコレクション

本システムでは計算量の多い，高負荷の処理をスレーブマシンに行わせるため，メモリの確保や解放などの管理が非常に重要になってくる分，複数台のスレーブマシンのメモリを管理するのは非常に困難な作業となる．C 言語などでは，メモリの管理をプログラムで定義しなければならず，管理者の負担が大きいと考えられる．

しかし，Java はガーベージコレクションによって，一定時間使用されなくなったオブジェクトのメモリを自動的に解放することができる．さらに，ガーベージコレクションはバックグラウンドでスレッドとして実行されているので，管理者がメモリ管理を考慮することなく，システム開発を行うことができる．この機能によって，メモリ管理に関する作業を軽減できると考えた．

2.2 HORB

本研究で分散処理環境を構築する上で使用した分散オブジェクト技術がHORB である．本節では，分散オブジェクトに関して説明すると共に，HORBに関して詳しく説明する．

2.2.1 分散オブジェクト

分散環境を実現する手法や技術は複数存在するが，ネットワークコンピューティングに伴う複雑さをより軽減する技術の一つに分散オブジェクト技術がある．

分散オブジェクトとは，ネットワーク上のどこにでも存在することができるオブジェクトのことである．分散オブジェクトは独立して存在するコードであり，遠隔のクライアントからはメソッド呼び出しによってアクセスされる．

分散オブジェクト技術が実現する最も重要な機能は，分散オブジェクトの呼び出しである．オブジェクトを提供する側をサーバ，オブジェクトを呼び出す側をクライアントとした場合，クライアントはネットワーク上におけるオブジェクトの位置を意識することなくそのメソッドを呼び出すことができる．また，サーバはあたかもローカルなオブジェクトを実装するかのように，外部から呼び出し可能なオブジェクトを実装することができる．このように，分散されたオブジェクト間の通信は透過的に実行される．

しかし，この呼び出しの裏側では，サーバに到達するためのネットワークアドレスの解決とコネクションの確立，要求や応答のメッセージの組み立てと送受信といった複雑な処理が行われている．これらの複雑な作業を隠蔽するためにオブジェクトを利用するのが分散オブジェクト技術である．

クライアントが分散オブジェクトのメソッドを，そのオブジェクトの位置に関係なく呼び出すことができると述べたが，クライアントが呼び出していたのは，ターゲットとなる分散オブジェクトと同じメソッドとパラメータを持つ代理（プロキシ）オブジェクトというローカルオブジェクトである．プロキシオブジェクトは，必要に応じてサーバとのコネクションを確立し，要求の組み立てと送信等を行う．

また, 異なるプログラムミング言語や OS 間を超えてメソッドコールを実現す

るために,パラメータなどのデータをプログラミング言語やOSの違いに依存しない共通形式に変換する必要がある．このような各プログラム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ことをマーシャリングといい，逆に共通形式からネイティブなデータ表現に変換することをアンマーシャリングという．このマーシャリングとアンマーシャリングもプロキシオブジェクトが行う機能の一つである．つまり，リモートメソッドコールの詳細をクライアントから隠蔽している．

サーバ側のリモートメソッドコールの詳細を隠蔽しているのがスケルトンオブジェクトである．スケルトンオブジェクトはクライアントから受信した要求に対して，実際のオブジェクトのメソッドをコールし，その結果をクライアントに返却する．

ただし，一般的な分散オブジェクトでは，さまざまな言語が利用されているため，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ーフェースをIDLによって定義する必要があり容易ではない．

2.2.2　HORBの概要

本システムは，分散オブジェクト技術の一つであるHORBを使用することによって分散処理環境を実現している．HORBは，独立行政法人　産業技術総合研究所の平野聡博士が開発したJava用分散オブジェクトである［4］.

HORBは，さまざまな種類のコンピュータをネットワーク上で接続し，あたかも一つのコンピュータのように見なし活用する事を目標に開発された．コンピュータとネットワークが発明されて以降，1970年ごろに分散オブジェクトなどの概念は確立されてはいたが，その実現は非常に難しく，本当の意味で使える分散オブジェクトとなるとHORBが先駆け的存在であったといえる．HORB以前にもCORBAを基盤とした考えがあったが，異なる機種間での相互運用性を実現したのはHORBが初めてであった.

HORBは世界で始めて実現されたJava用分散オブジェクトであり，**SUNのJavaと100%互換性があり，あらゆるJava処理系で動作可能**である．また，**マルチプラットフォーム対応**であり，**IDLを記述する必要がない**などの特徴を備える.

2.2.3　HORB アーキテクチャ

HORB は，HORB コンパイラ，HORB サーバ（ORB の一種），HORB クラスライブラリで構成されている． HORB コンパイラでコンパイルしてできた Java オブジェクトは分散環境で使用することができる．HORB は Sun の提供する Javac コンパイラ，Java インタプリタ，Java システムクラスとともに動作する．Sun のプログラム・ソースコードには，HORB ツールキットのためにいかなる変更も加えられていない．それは HORB のポータビリティを保つためと，Sun のソースコードライセンスに束縛されないためである．つまり HORB はプラットフォーム固有のネイティブメソッドを使っておらず，マシンアーキテクチャ独立である［5］.

2.2.4　HORB における基本的なサーバ・クライアントシステム

まず，サーバとクライアントマシンがあると仮定し，それぞれにサーバクラスとクライアントクラスを置くとする．プログラムを開始させると，まずクライアント・オブジェクトがサーバ・システム内にサーバ・オブジェクトを生成する．次にサーバ・オブジェクトのメソッドを呼び出す． 2 つのオブジェクト間のメソッド呼び出しをシームレスで透過（トランスペアレント）にするためにプロキシとスケルトン，および ORB が用いられる．ここでいうプロキシとスケルトンは 2.2.1 章で説明したものと同等のものである．

プロキシとスケルトンオブジェクトは HORB コンパイラによって生成されるので，ユーザはクライアントとサーバ・オブジェクトを記述して，サーバ側で HORB を起動し，クライアント側のプログラムを動作させるだけでよい．

また，下位レベルのネットワーク層と上位レベルのアプリケーション層の間の API（Application Program Interface）を定義するオブジェクト間通信インターフェースの定義である IOCI（Inter Object Communication Interface）に関しても，HORB コンパイラによってプロキシとスケルトン内に自動的に組み込まれるため，ユーザは IOCI を気にかける必要がない．

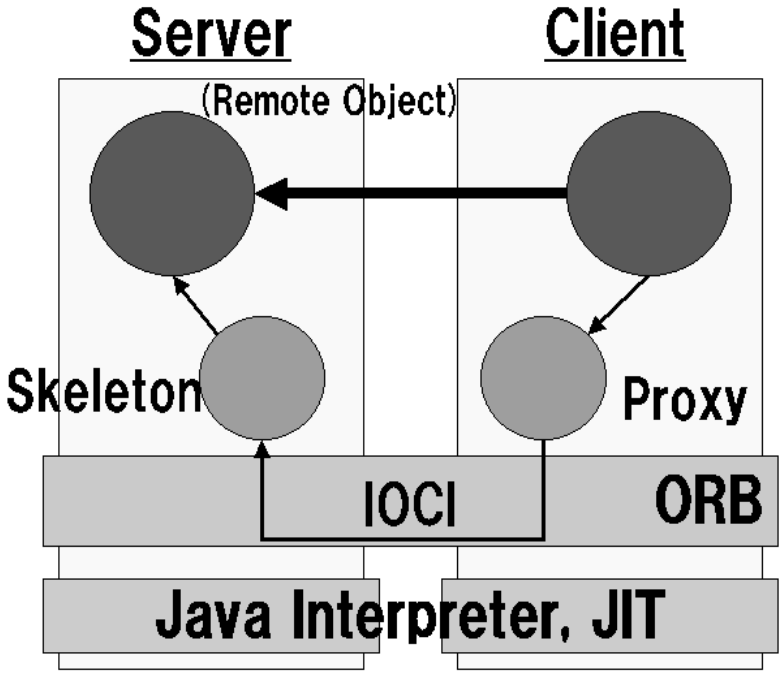


図 2.1　HORB の全体像

2.2.5　HORB プログラミングの基本とその処理流れ

本節では，これまで述べてきた HORB によって実現される処理の流れを，簡単な HORB プログラムを例に説明する．

表 2.1　簡単な HORB プログラム

［Server. Java］

```
１：public class Server{
２：　　public String message(String name){
３：　　　　return “Hello　“ +name+ “!”;
４：　　}
５：}
```

［Client. Java］

```
１：class Client{
２：　　public static void main(String args[]){
３：　　　　Server_Proxy server = new Server_Proxy(“horb://locathost”);
```

```
４：              String result = server.message(“World”);
５：              System.out.println(result);
６：    }
７：}
```

表 2.1 は，String オブジェクトをやり取りする基本的なプログラムである．このプログラムを用いて，オブジェクトをやり取りするためには，まず Client.java の 3 行目で Server.java の代理オブジェクトとなる Server_Proxy オブジェクトを生成する必要がある．これにより，Server.java の message メソッドを，ローカルメソッドのように呼び出すことができる．

この例では，Client.java でプロキシオブジェクトを生成する際に “localhost” としているが，これは 1 台のコンピュータ上でこのプログラムを動作させる際の記述であり，他のコンピュータとの間で動作させる場合には，そのコンピュータのホスト名もしくは IP アドレスが必要になる．

次に，Client.java の 3 行目のプロキシオブジェクトの生成について流れ図を用いて説明する．

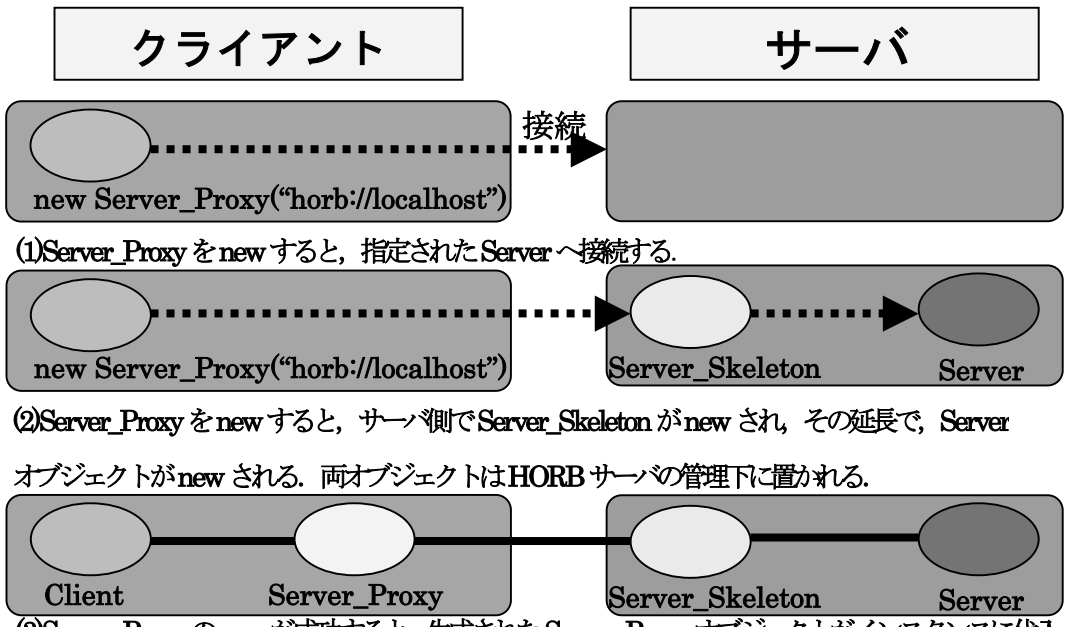
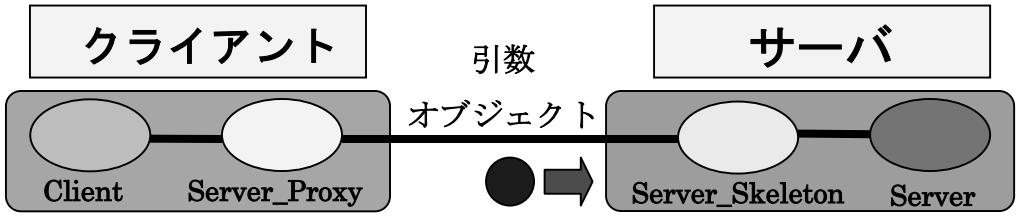
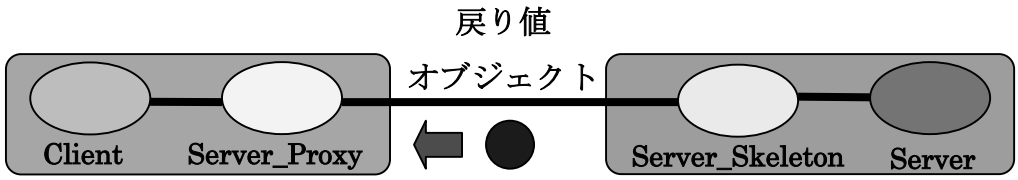


図 2.2　プロキシオブジェクトの生成

また，Client.java の 4 行目にある「message メソッドのリモート呼び出し」についても流れ図を示す.



(1)Client.java から message メソッドが呼び出され，Server_Proxy と Server_Skeleton を経由して，サーバに存在する Server オブジェクトのメソッドとして実行される．この時，引数の String オブジェクトがネットワーク間で転送される.



(2)Server オブジェクトのメソッドの戻り値が，Server_Skeleton と Server_Proxy を経由してネットワーク転送され，Client に渡される.

図 2.3. サーバ・メソッドのリモート呼び出し

次に, 表 2.1 のプログラムをネットワーク環境で動作させるための方法を説明する．ユーザまず HORB コンパイラを用いて Server.java をコンパイルする．すると HORB コンパイラは Server_Proxy.java, Server_Skelton.java を自動的に作成し，javac コンパイラによってコンパイルする．次に，Client.java をコンパイルする．以上のコンパイル作業によってできた，各クラスファイルを表 2.2 のように配置する.

表 2.2　クラスファイルの配置

クライアント側に配置する クラスファイル	Client.class
	Server_Proxy.class
サーバ側に配置するクラスファイル	Server_Skelton.class
	Server.class

2.2.6　その他の HORB の特徴

これまで述べてきたことから，HORB は分散オブジェクトを非常に簡単に実現できる技術であるということがわかる．さらに HORB には，**安定性やセキュリティ面から非常に信頼性が高く，RMI や他の ORB と比べて 2 倍以上高速**であり，**非同期メソッドをサポートしている**など，他の分散オブジェクト技術よりも優れた面を数多く持っている．また，無償で提供されていることや，オープンソースであること，そしてニュースグループが存在し，比較的簡単に開発に携わった人や HORB を使用している人などとコミュニケーションを図れるなど，研究者にとっては非常に使用しやすい分散オブジェクトであるといえる．

第3章 システム構成

本章では，分散処理環境を構成するハードウェア，ソフトウェアの構成に加えて，本システムで導入しているクラスファイルの動的取得機能について述べる．

3.1 ハードウェア構成

本節では，分散処理環境を構成するマスターマシンとスレーブマシンのスペックや接続構造といったハードウェア構成に関して述べる．

3.1.1 マシンスペック

本システムは，マスターマシン 1 台とスレーブマシン 25 台でもって構成される．マスター，スレーブの各スペックは以下のとおりである．

表 3.1 計算機のスペック

	Master	Slave
CPU	Intel Xeon™ 3.06GHz ×2	Pentium Ⅱ 400MHz
HDD	70GB + 2.75TB	6GB
Memory	2GB ×2	640MB
OS	RedHat Enterprise ES release 3	Vine Linux 2.6r4

マスターマシンの HDD には，外部記憶装置として容量 2.75TB の RAID が組んであり，ネットワークシミュレーションの結果を保存する．

3.1.2 ハードウェアの接続構造

本システムでは上記の 26 台の計算機を，マスターマシンをルートとする 1 段の木構造で接続する．マスターマシンに 0，スレーブマシンに 1～25 の ID を割り当て，学内のネットワークとは別のローカルネットワーク（192.168.1.0）を用いて接続する．現在，学内を含む外部ネットワークへの接続は行っていないが，マスターマシンのみ外部への接続が可能な環境になっている．

3.2　ソフトウェア構成

本節では，構成された分散処理環境上で動作するソフトウェア構成に関して述べる．

3.2.1　プログラム配置

本システムを構成する基本となるプログラムを，それぞれ配置される側と共に以下に示す．

表 3.2　基本プログラムの配置

Master	Simulation_Server
	Simulation_Server_Skelton
Slave	Simulation_Client
	Simulation_Of_Infection
	readGraph
	Create_Model
	Simulation_Server_Proxy

3.2.2　プログラムの動作内容

次に，各プログラムについて説明する．なお，各種パラメータや作成するネットワークモデル，シミュレーションの内容などについては，4 章の実験内容で詳しく説明する．

Master

（1） Simulation_Server

Master 側のメインプログラム. 実験に使用される各パラメータの決定と，実験結果の保存を行う．

（2） Simulation_Server_Skelton

HORB コンパイラによって作成される，Simulation_Server の Master 側の代理オブジェクト．複雑な作業の隠蔽を行い，Slave 側とのオブジェクトやデータの受け渡しに使用される．

Slave

（1） Simulation_Client

Slave 側のメインプログラム. HORB サーバとして起動した Master へアクセスし，シミュレーションに必要となるパラメータを取得する．そして取得したパラメータをもとに（2）～（4）のオブジェクトを作成する．

（2） Create_Model

ネットワークモデル作成オブジェクト. Simulation_Client が取得したパラメータをもとにネットワークモデルを作成する．作成したネットワークモデルは，辺情報ファイルとして保存する．

（3） readGraph

辺情報ファイルを読み込むオブジェクト．辺情報ファイルを読み込みネットワークデータを作成する．

（4） Simulation_Of_Infection

ウィルス伝搬シミュレーションを行うオブジェクト．Simulation_Client が取得したパラメータを使用し，作成したネットワークモデル上でウィルス伝搬シミュレーションを行う．

（5） Simulation_Server_Proxy

HORB コンパイラによって作成される, Simulaion_Server の Slave 側の代理オブジェクト．Simulation_Server_Skelton と同様に，作業の隠蔽と Master 側とのオブジェクトやデータの受け渡しに使用される．

3.3　クラスファイルの動的取得機能

これまで述べてきたように，HORB を用いて異なる計算機の間で何らかの処理を行おうとした際，各計算機にそれぞれ必要なクラスファイルをあらかじめ用意しておく必要がある．これを，スレーブ側が必要なときに必要なクラスファイルを動的に取得できるようにする機能が，クラスファイルの動的取得機能である [6]．本節では，このクラスファイルの動的取得機能について説明する．

3.3.1　動的取得機能の構成と処理の流れ

本システムでは，スレーブ側が行うネットワークモデル作成やウィルス伝搬シミュレーションなどの処理を行うクラスファイルを，手動により直接スレーブマシンに配置せず，スレーブ側が必要としたときに必要なクラスファイルをマスター側から自動的に取得できる，クラスファイルの動的取得機能を実現した．これまで，この動的取得機能の可能性は論じられているが，実際に実装した例はほとんど見当たらない．特に複数台の分散処理としては初めての試みと思われる．

クラスファイルの動的取得機能を実現するための主要なプログラムを，処理の流れと共に以下の図に示す．

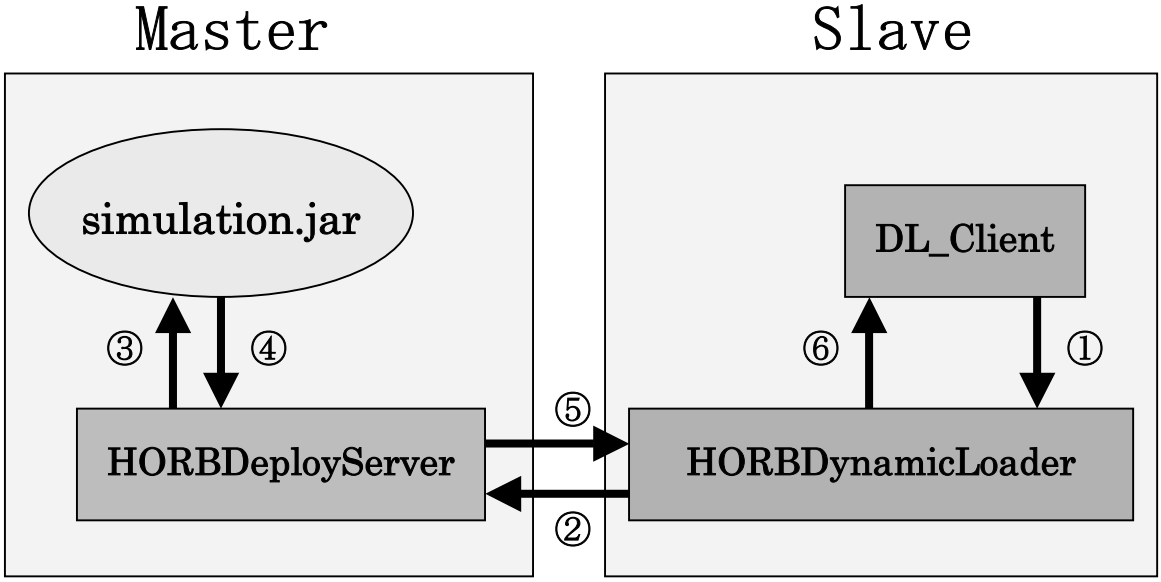


図 3.1　クラスファイルの動的取得処理の流れ

・ HORBDeployServer

HORBDyanamicLoader からのクラスファイル取得要請を受信し，対応するクラスファイルを.jar ファイルから呼び出し，スレーブ側へ返送する．

・ HORBDynamicLoader

スレーブ自身のクラスローダでロードできなかったクラスファイルを，HORBDeployServer から取得し，独自のクラスローダにロードする．

・ DL_Client

クラスファイルを動的に取得するためにスレーブが実行するプログラムである．HORBDynamicLoader を介して HORBDeployServer に接続し，HORBDynamicLoader がロードしたクラスを実行する．

・ simulation.jar

今回の動的取得機能を使用して取得するクラスファイルをまとめたアーカイブファイル．Simulation_Client，Simulation_Of_Infection，readGraph，Create_Model の 4 種類のクラスファイルをまとめ, マスターマシンのカレントディレクトリにある storeDir ディレクトリに格納する．今回，スレーブの動作に必要なクラスファイルを simulation.jar としてまとめているが，クラスファイルをそのまま storeDir に格納しておくことも可能である．しかし，今後新たな機能が追加されたとき, 関連するクラスファイルを 1 つの jar ファイルにまとめておくほうが管理しやすいということで jar ファイルを使用している．

DL_Client（表 3.3）は，起動するとすぐ，2 行目で HORBDynamicLoader を介して Master の HORBDeployServer へ接続する（図 3.1①）．そして，HORBDynamicLoader の getObject()メソッドを用いてクラスファイルを取得し実行する（図 3.1⑥）．ここでロードするクラスファイルは 10 行目で示すとおり，Simulaion_Client である．

表 3.3　DL_Client.java

［DL_Client.java］

```
1 ：public class DL_Client{
2 ：    private HORBDynamicLoader hdl =
            new HORBDynamicLoader(new HorbURL("horb://localhost"));
3 ：    public void Client(String object){
4 ：        DoCommon obj = (DoCommon)hdl.getObject(object);
5 ：        if(obj != null)
6 ：            obj.kickDo(hdl);
7 ：    }
8 ：    public static void main(String args[]){
9 ：        DL_Client dl = new DL_Client();
10：        dl.Client("Simulation_Client");
11：    }
12：}
```

ここで，4 行目の“DoCommon obj = (DoCommon)hdl.getObject(object);”は，getObject()が返すのはオブジェクトなので，DoCommon にキャスティングする必要がある．そして取得した DoCommon（今回は Simulation_Client）の kickDo()を呼び出す.

表 3.4　Simulation_Client.java

［Simulation_Client.java］

```
1 ：import horb.orb.*;
2 ：public class Simulation_Client implements DoCommon{
3 ：Simulation_Server_Proxy ss =
                    new Simulation_Server_Proxy("horb://localhost");

4 ：    public void kickDo(){
5 ：        System.out.println("Simulation_Client");
6 ：    }
```

7 :　public void kickDo(ClassLoader loader){

…

Similation_Client.java の引数を持たない kickDo()は，スレーブマシンのカレントディレクトリに Simulation_Client.class が存在した場合の動作である．今回の場合，スレーブ側にはこのクラスファイルが存在することはないため，5 行目のように特に意味を持たない処理を行うようになっている．引数を持つ kickDo(ClassLoader loader)では，3.2.2 節で説明した処理を行うようになっている．

HORBDynamicLoader（表 3.5）は ClassLoader を継承している（8 行目）．DL_Client から呼んでいた getObject()では引数として渡されたクラスをインスタス化しその Object を返すが，その際 download()を実行する．

download()では，最初にシステムのクラスローダから受け取ったクラスファイルがロード可能かどうか調べる．そしてロードできない場合，つまり ClassNotFoundException をキャッチした場合，HORBDeployServer からクラスファイルを取得する（図 3.1②）．

55 行目では，HORBDeployServer から，引数として渡したクラス名に関係するファイル群を JAR ファイルのバイト列として受け取っている．その後，受信したバイト列を解析し，クラスをクラスローダにロードし実行可能な状態にする（図 3.1⑤）．

表 3.5　HORBDynamicLoader.java

［HORBDynamicLoader.java］

```
8 : public class HORBDynamicLoader extends ClassLoader{
                                        …
50 : private void download(String name) throws Exception{
51 :     try{
52 :       loadClass(name);
53 :     }catch(ClassNotFoundException e){
54 :       try{
55 :         byte data[] = server.getJarFromClass(name);
                                        …
```

```
115：public Object getObject(String name){
116：      try{
117：        download(name);
118：       return loadClass(name,true).newInstance();
                                        …
```

マスター側では, HORBDeployServer が起動時に storeDir ディレクトリにある JAR ファイルとクラスファイルを確認する（図 3.1③）. 50 行目の getFromClass()では，指定されたクラスファイルが格納されている JAR ファイルを読み出し，バイト列として返信する（図 3.1④）.

表 3.6　HORBDeployServer.java

[HORBDeployServer.java]

```
50：   public byte[] getJarFromClass(String className){
51：       String fileName = (String)map.get(className+".class");
52：       if(fileName != null){
53：         return readFile("./storeDir/" + fileName);
54：       }
                                        …
```

3.3.2　動的取得機能を実現するためのクラス配置

3.3.1 節で示した，クラスファイルの動的取得機能を実現するためのプログラムをコンパイルした結果生成されるクラスファイルの配置を以下に示す.

表 3.7　クラスファイルの配置

Master	HORBDeployServer
	HORBDeployServer$1
	HORBDeployServer_Skelton
	Simulation_Server
	Simulation_Server_Skelton
storeDir	simlation.jar
Slave	DL_Client
	DoCommon
	HORBDynamicLoader
	HORBDynamicLoader$EntryData
	HORBDynamicLoader$ImageData
	HORBDynamicLoader$ClassData
	HORBDeployServer_Proxy
	Simulation_Server_Proxy

表 3.7 を見てわかるように，マスター側には HORBDeployServer と Simulation_Server 本体とそのスケルトン，スレーブ側にはそれぞれのプロキシが配置される．図 3.1 では，HORBDeployServer と HORBDyanamicLoader が直接通信を行っているように示したが，実際は 2 章の分散オブジェクトで説明したとおり，それぞれの代理オブジェクトが間に入っていることに注意しなければならない．

また，表 3.7 のようにクラスファイルを配置し，これらのクラスファイルの動的取得機能を実現した後，マスターが持つ simulation.jar 内のクラスファイルを書き換え，最コンパイルし JAR ファイルを再度 storeDir に上書きするだけでスレーブの行う処理を変更することができる．また，ネットワークモデル作成やウィルス伝搬シミュレーションに必要となるパラメータを変更する際も，Simulation_Server のパラメータ値を変更し，HORB コンパイラでコンパイルした後，Simulation_Server と Simulation_Server_Skelton をマスター側に上書きするだけでシミュレーション内容を変更することができる．

ただし，Simulation_Server クラスファイル内に新しいメソッドを加えた場合に限り，各スレーブマシンの Simulation_Proxy クラスファイルを書き換えなくてはならない．

3.3.3　クラスファイルの動的取得機能によって得られるメリット

今回のように，クライアントとなるスレーブマシンが多く必要となる場合，通常の方法だと処理内容の変更や修正のたびに 25 台の計算機に必要なクラスファイルをコピーする必要がある．FTP（File Transfer Protocol）や NFS（Network File System）などを使用したとしても，非常に手間のかかる作業である．しかしながら，このクラスファイルの動的取得機能によって，その作業の手間を省くことができる．これは大規模な分散処理を行う場合には非常に有効な手段であるといえる．

また，クラスファイルの動的取得機能の実現によって，既存のクラスファイルの変更点が少ないということも非常に大きなメリットであるといえる．例えば，本システムでは，スレーブが処理に使用するクラスファイルは Simulation_Client，Simulation_Of_Infection，readGraph，Create_Model の 4 種類であったが，動的取得機能を実現する以前の内容に変更を加えたのは，スレーブマシンから直接呼び出される Simulation_Client だけである．HORBDynamicLoader の 55 行目で説明したとおり HORBDeployServer に渡した引数のクラスに関係するファイル群をロードすることができる．そのため，Simulation_Client によって生成されるオブジェクトである他の 3 種類のクラスには，動的取得機能のために手を加える必要がない．

さらに，マスターマシンの持つクラスファイルをダウンロードするということから，スレーブマシン間でのクラスファイルのバージョンの違いや，処理の不一致などを解消することができる．

第4章 ネットワークモデルと ウィルス伝搬モデル

本章では，本システムにおいて作成するネットワークモデルと，ウィルス伝搬によるノードの状態遷移を規定するウィルス伝搬モデルについて述べる．

4.1 作成するネットワークモデル

本節では，実験で作成したネットワークモデルである，Coupled duplication divergence モデル［7］について，その生成過程やその特徴，さらにその背景にあるスケールフリーネットワーク［8］について述べる．

4.1.1 スケールフリーネットワーク

古典的なネットワーク研究は，ランダムあるいは結晶構造などの規則的なグラフ構造を前提に行われてきた．しかしながら近年，Internet のルータの接続関係や現実の多くのネットワークには，Scale-Free（SF）と呼ばれるべき乗則に従う共通の構造的特徴が存在するということが発見された［9］．

べき乗則に従うネットワークとは，各ネットワークノード（頂点）の持つ辺数（次数）を k としたとき，その分布が $P(k) \approx k^{-\gamma}$ に従うというものである．さらに，多くの現実のネットワークにおいて，べき指数は $2 < \gamma < 3$ であることがわかっている．一般的にべき乗則に従うネットワークには，極端に大きな次数を持つハブといわれるノードが存在し，その他の次数が少ないノードと共にネットワークを形成する．このようなべき乗側に従うネットワークは，ランダム構造を持つネットワークにおける平均次数（分布のピーク値付近）のようなネットワークの特徴的なスケールを持たないことから，スケールフリーネットワークと呼ばれる［10］．

このスケールフリー特性は，Internet などの電子的なネットワークのみならず，電力網，論文の引用関係，俳優の共演関係，遺伝子やたんぱく質などの生態系などさまざまなネットワークに見られる特性であるということがわかってきている．

4.1.2　Coupled duplication divergence model

今回, 実験に使用するネットワークモデルは, SF ネットワークの一種である, Coupled duplication divergence（CDD）モデルである.

CDD モデルは，たんぱく質間の相互作用ネットワークの持つ機能的役割を示すためのモデル（たんぱく質の発現を表すグラフ）であるといえ，その生成過程に duplication（複写）と divergence（拡散）プロセスをもつモデルである[7]. ここでいう複写プロセスとは，たんぱく質や遺伝子が自身をコピーする作業に相当し，拡散プロセスは複写プロセスの際に，退化変異などによって遺伝子間の接続が欠落することを意味する．さらに自己相互作用（セルフーインタラクション）によって被複写頂点と複写頂点の間に新しい接続が生まれることも考慮されている．CDD モデルの生成過程を図 4.1，4.2 に示す.

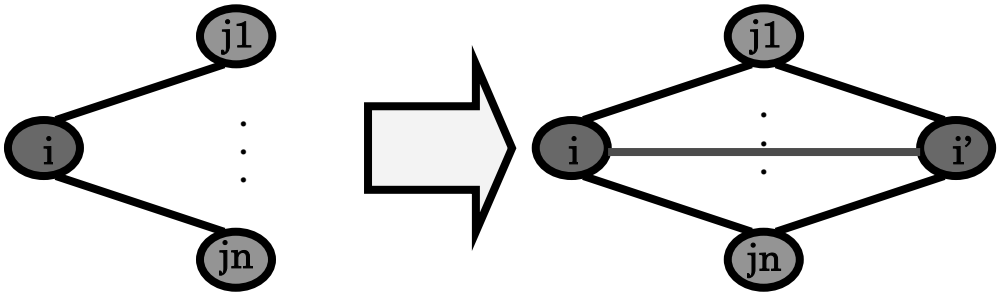


図 4.1　duplication（複写）プロセス

ある時点でのネットワークを図 4.1 の左の状態としたとき, そのネットワーク内から被複写頂点 i をランダムに選択する. 次に新規頂点として, 被複写頂点 i と同じ接続情報を持つ複写頂点 i' が作成される（図 4.1 右）．複写頂点 i' が生成された後，被複写頂点と複写頂点の間に，確率 q_v で対結線（図 4.1 右赤線）が生成される.

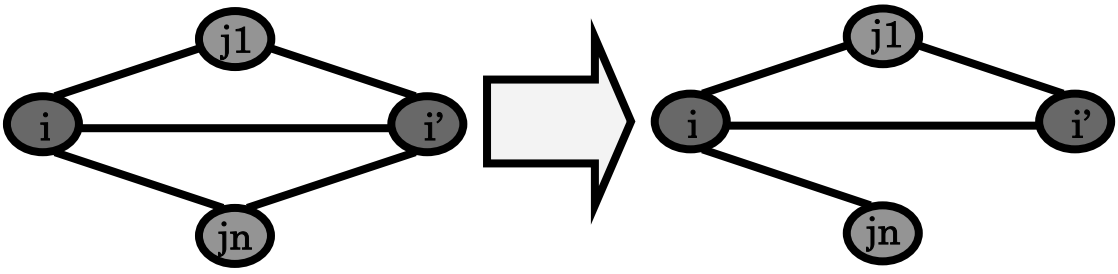


図 4.2　divergence（拡散）プロセス

拡散プロセスでは，複写プロセスによってできたネットワークのうち，被複写頂点と複写頂点に接続された複数の頂点 j に関して，辺 (i, j) と辺 (i', j) をランダムに選択し，選択した辺を確率 $(1-q_e)$ で削除する．図 4.2 の右の図では，頂点 jn に関して，辺 (i', jn) が削除された状態を示している．

CDD モデルでは，確率 q_v が高い場合，セルフインタラクションが強くなり，その結果，類似したノード数に基づくハブとハブが接続される確率が高くなる．つまり，高次数ノード同士のつながり（正の結合相関）が強くなり，高次数ノードと低次数ノードのつながり（負の結合相関）は弱くなる．言い換えれば，この q_v によって次数の結合相関をコントロールすることができる．このことは，社会ネットワークで見られる正の結合相関と，技術：生物ネットワークで見られる負の結合相関に対応したネットワークを生成できることを意味する．

次に，作成した CDD モデルを可視化した例を示す．ここで示す図は，本研究室の宮崎が作成したネットワークモデルの可視化ツールを使用して作成したものである．図 4.3，4.4 ともに初期ノード数を 2，辺数 1 からノード数は 200 まで成長させた CDD モデルの例である．

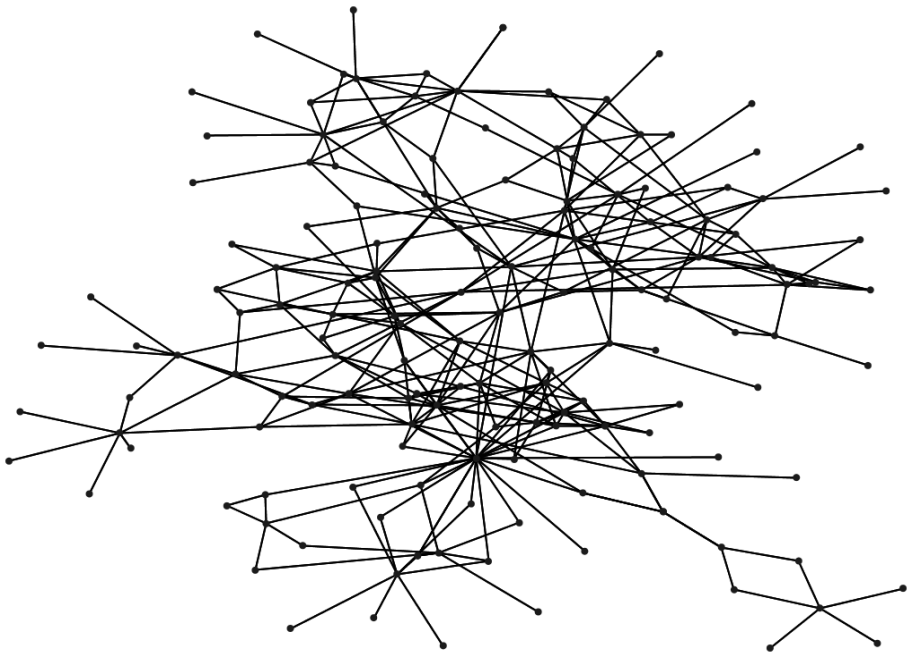


図 4.3　$q_v = 0.1$　での CDD モデル例

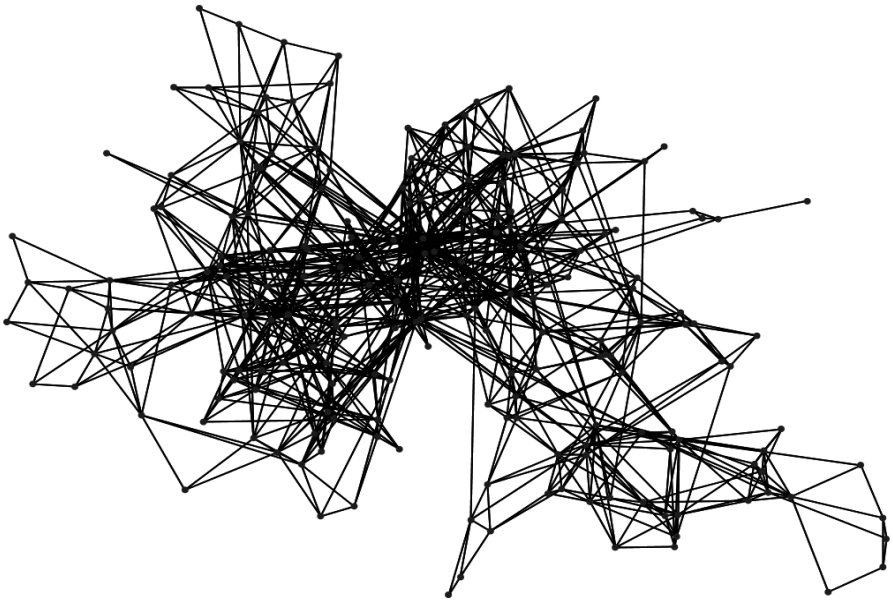


図 4.3　$q_v = 0.9$　での CDD モデル例

4.2　ウィルス伝搬モデル

本節では，ウィルスの伝搬によるネットワークノードの状態遷移を規定するウィルス伝搬モデルに関して，実験で採用した SIR モデルをもとに述べる．

4.2.1　従来のウィルス伝搬シミュレーション

従来のウィルス伝搬シミュレーションは，一様ランダムネットワークや完全グラフ上で行われてきた．しかしながら，一様ランダムや完全グラフは，均一なウィルスとの接触機会を表すため現実的なモデルとは言いがたい．現実の接触感染ネットワークは SF 構造を持つ［11］.

一方 SF ネットワークでは, 感染率がどんなに低いウィルスでもネットワーク全体に広がっていくということが示されている．つまり，SF ネットワーク上でのウィルスには絶滅の閾値がないということである［12］.

ウィルスの感染モデルには，SIS（Susceptible-Infected- Susceptible）モデルや SIR（Susceptible-Infected- Recovered）モデル，さらに SHIR（Susceptible-Hidden infected-Infectious- Recovered）モデル［11］などが存在する．本研究では，感染ノードの発見と駆逐（免疫）の後に永続的な免疫状

態を持つ SIR モデルを使用し SF ネットワーク上の感染伝搬シミュレーション実験を行う．

4.2.2 SIR（Susceptible-Infected- Recovered）モデル

SIR モデルでは，ネットワークノードの状態を未感染（S），感染（I），免疫（R）のいずれかの状態とする．感染したノードと接触状態にある（リンクで接続されている）ノードは，ある一定の感染確率bで状態が S から I に遷移する．さらに，感染状態にあるノードは，ある一定の免疫確率dで状態が I から R に遷移する．免疫状態になったノードは，永続的に免疫状態であり，再び感染することはない．このようにノードの状態が確率的に遷移していき，その遷移過程は感染確率bと免疫確率dに依存する．

また，今回考えるウィルスは，現実に存在するウィルスの感染媒体やその被害，さらに感染の時限起動などの細かい点は考慮せず，接触状態にある未感染ノードの状態を感染状態に遷移するという単純なものを対象にしている．

第5章 システムの詳細

本章では，作成した分散処理環境で行うネットワークモデル作成とウィルス伝搬シミュレーションの内容について述べる．さらに，今回の実験で行う全体のシミュレーション回数を述べ，実際にスレーブマシンに分配するタスク数について言及する．

5.1 ネットワークモデル作成

本節では，分散処理環境で行うネットワークモデル作成について述べる．

5.1.1 ネットワークモデル作成パラメータ

評価実験で作成するネットワークモデルは 4.1 節で述べた CDD モデルであり，マスターマシンが設定したモデル作成パラメータをもとに，スレーブマシンで作成される．ここで，マスターによって指定されるモデル作成パラメータは以下の 4 つである．

- N ：最大ノード数

ネットワークの初期状態をノード数 2，辺数 1 とし，ノード数がこの値になるまでネットワークを成長させる．ただし，次数 0 のノードはカウントしない．

- E ：基準辺数

ノード数が規定値まで成長したネットワークの辺数を指定する値である．

絶対辺数E＝(N×平均次数÷2)±5%　（平均次数は全ノードの次数の平均値）

- q_v ：対結線確率

ネットワークモデル作成過程でセルフインタラクションが起こる確率

- q_e ：非除線確率

ネットワークモデル作成過程で辺を削除しない確率

CDD モデルの性質に大きく関わる要素は対結線確率q_vである．CDD モデルは，セルフインタラクションの強弱によって大きく性質を変えるモデルある．そこで，正負の結合相関を変化させるセルフインタラクションの強弱がウィル

ス伝搬にどのように影響を及ぼすかを調べるために，q_vの値を{0.1,0.3,0.5,0.7,0.9}の 5 種類とし，基準辺数の範囲内に総辺数を合わせるためにq_eを変動させる．つまり，表 3.1 のようにノード数Nごとにq_vの値に従った 5 種類の CDD モデルを作成する．また，基準辺数を決める平均次数は今回 8 としている．実際に実験を行ったモデル作成パラメータを以下に示す．

表 5.1　ネットワークモデル作成パラメータ

N	1000	2500
E	4000±5%	10000±5%
qv	0.1,0.3,0.5,0.7,0.9	0.1,0.3,0.5,0.7,0.9
qe	変動	変動

制約条件としての基準辺数を指定する理由は，一般的にノード数Nの CDD モデルを作成する場合，確率q_v，q_eによって，最終的にランダムに生成されたネットワークが持つ辺数は変動するということが挙げられる．すなわち，平均的挙動を調べる際に本来は総辺数を一定値にしたいが，基準辺数を指定しない場合，同一パラメータでネットワークを生成しても，平均辺数に偏りができてしまう［13］．今回行うウィルス伝搬シミュレーションでは，ウィルスの感染は接触感染で起こるため，総辺数の違いによって各ノード間の接触頻度に差が生まれ，その結果シミュレーションの結果に大きな差を生む原因になる．

そのため実験でのモデル作成では，ネットワークを生成した結果，総辺数が基準辺数の範囲内にならない場合は，ネットワークを作り直すという処理を追加している．

5.1.2　作成するネットワークモデル数

実験では，5 種類の CDD モデルをそれぞれ 100 個ずつ，合計 500 個作成する．CDD モデルは同一パラメータで作成しても，q_vとq_eの確率によって最終的な状態が変化する．そのため，各パラメータの組み合わせごとに 100 個ずつ作成し，そのそれぞれに対してウィルス伝搬シミュレーションを行い，最終的に 100 個のネットワークモデルで得られたデータを平均化し，パラメータの組み合わせごとの結果を出す．これはネットワークのランダム生成と状態遷移のランダ

ムさに対して，平均的なウィルス伝搬の挙動調べていることに相当する．

5.1.3　ネットワークモデルファイルの構成

実際に作成されたネットワークモデル情報は，どのノードとどのノードが接続しているか，つまりどのノード間に辺が存在するかを示した辺情報ファイルに格納される．ネットワークモデル作成過程で生成されたノードはそれぞれ ID を持ち，辺情報ファイルはその ID のペアを列挙することによって，ノード間の接続情報を記述する．

表 5.2　辺情報ファイルの例

0　1
0　12
4　5
4　21
10　137
…

5.1.4　実際に作成したネットワークモデル

実際に表 5.1 のパラメータで作成したネットワークの次数分布と結合相関を示す．それぞれのデータは，表 5.1 のパラメータで作成した CDD モデル 100 個のデータの平均値である．

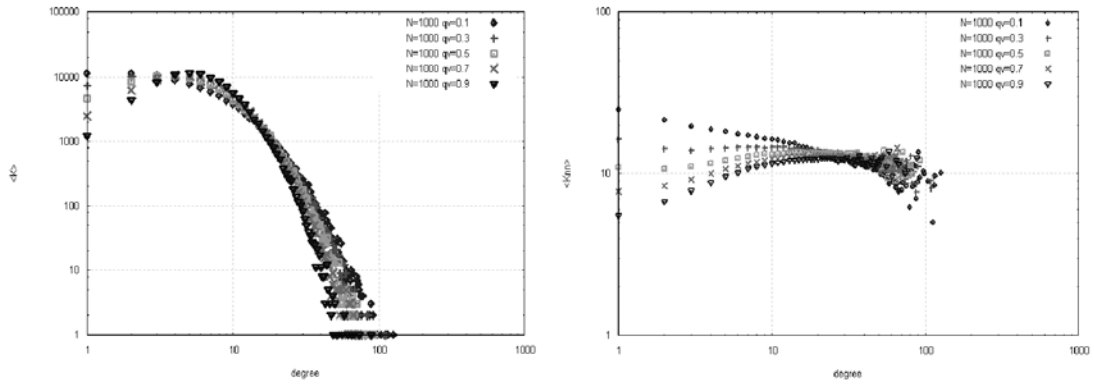


図 5.1　次数分布（左）と平均結合相関（右）（ノード数 1000）

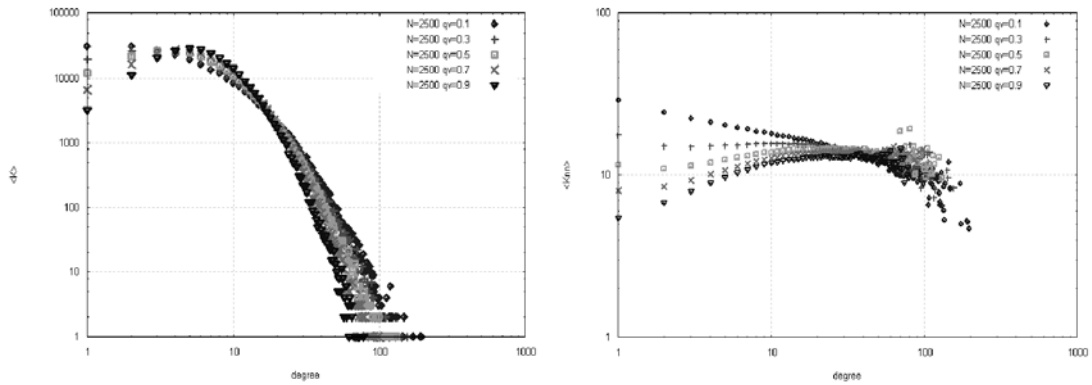


図 5.2　次数分布（左）と平均結合相関（右）（ノード数 2500）

次数分布とは，次数kのノード数の分布であり，式 5-1 で表される．

$$P(k)=\frac{N(k)}{N} \tag{5-1}$$

式（5-1）の$N(k)$は次数kのノード数である．CDD モデルはスケールフリーネットワークの一種であり，図 5.1，5.2 の結果を見ると，その特徴であるべき乗則（両対数グラフにおける直線部分）が表れている．

次に平均結合相関であるが，これは次数kのノードと次数k'のノード間の結合にどのような相関があるかを示すものであり，式 5-2 で表される．

$$\begin{aligned} <Knn>&=\sum_{k'}k'\cdot P(k'|k) \\ P(k'|k)&=\frac{N(k'|k)}{k\cdot N} \end{aligned} \tag{5-2}$$

式 5-2 の$N(k'|k)$は，次数kのノードと次数k'のノード間の連結している辺数を表す．つまり，平均結合相関が正の相関を持つ場合，高次数のノードと高次数のノードが接続し，逆に負の相関を持つ場合，高次数のノードと低次数のノードが接続するということを表す．CDD モデルは，対結線確率q_vが高くなると結合相関に正の相関（正の傾き）が表れ，逆にq_vが低い場合は負の相関（負の傾き）が表れる．図 5.1，5.2 をから，今回作成したネットワークモデルにもその特徴が表れていることがわかる．

5.2　ウィルス伝搬シミュレーション

本節では，5.1 節のネットワークモデル作成によって作成された CDD モデル上で行うウィルス伝搬シミュレーションについて述べる．

5.2.1　ウィルス伝搬シミュレーション

4.2.2 節で説明した SIR モデルにおいては, ネットワークノードの状態遷移は, ウィルスの感染力を表す感染確率bと, 感染したノードがウィルスを駆除し免疫化されるための免疫確率dに依存する．

今回の実験では感染確率b, 免疫確率dの強弱の組み合わせによるウィルス伝搬状況の変化を調べるために，感染確率，免疫確率をそれぞれ$\{0.1,0.3,0.5,0.7,0.9\}$の 5 種類とし，その組み合わせ 25 通りに関してウィルス伝搬シミュレーションを行う．

ウィルス伝搬シミュレーションは，5.1 節の SF ネットワークである CDD モデルを作成した後，初期状態を未感染ノード数$N-1$，感染ノード数 1，免疫ノード数 0 の状態からはじめ，感染ノード数が 0 になるか，一定の時間を迎えた段階で終了する．この時間はマスターが指定するパラメータの 1 つで，基本的にはこの終了時間を迎える前に感染ノード数は 0 になる．

5.2.2　ノード状態の変化のタイミング

作成したネットワークの各ノードは，未感染（S），感染（I），免疫（R）のいずれかの状態にある．その状態がウィルス伝搬パラメータによって確率的に遷移していく．感染状態 I のノードは，接触によって未感染状態 S である隣接ノードの状態を感染確率bによって状態を I に遷移させる．また，感染状態 I にあるノードは免疫確率dのよって状態 R に遷移する．ここで，前者の感染を拡大させる状態遷移は，後者の免疫化よりも先に起こるものとする．つまり感染ノードは, 接触しているノードの状態遷移を行ってから, 自身の状態遷移を行う．具体的には図 5.4 で示すように状態遷移が行われる．

各ノードの状態は 1 システム時間に 1 状態のみ遷移するものとし, 1 システム時間内に未感染状態 S のノードが 2 ステップ先の免疫状態 R に遷移することはない．例えば図 5.3 のような状態遷移は誤りである．ノード X がノード Y の状

態を I に遷移した（赤）あと，同時刻内にノード Y の免疫化処理（緑）が行われている．これはノード X の処理が終わった時点で，ノード Y の状態を変化させてしまっていることに問題がある．ノード X の処理の後ノード Y は自身が感染状態であるため，免疫化処理を行ってしまっているのである．

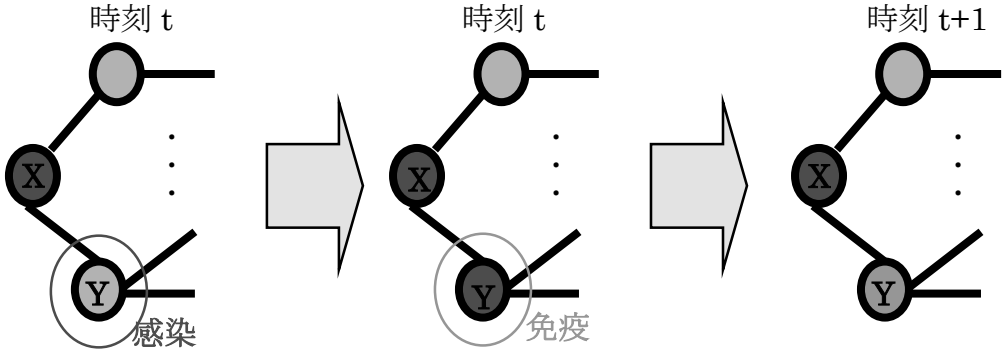


図 5.3　誤った状態遷移

このような誤った遷移方法を改善する手法が，ノードの状態遷移のタイミング操作である．これは 1 システム時間内に S から I，または I から R へ状態遷移することが決定したノードの状態を移行状態 C（黄）とし，全てのノードの状態遷移を確認するまでその状態で保持する．そして，すべてのノードの状態遷移を確認した後，移行状態 C から感染状態 I，または免疫状態 R へ状態を遷移する．これによってノードの状態遷移のタイミングを合わせることができる．

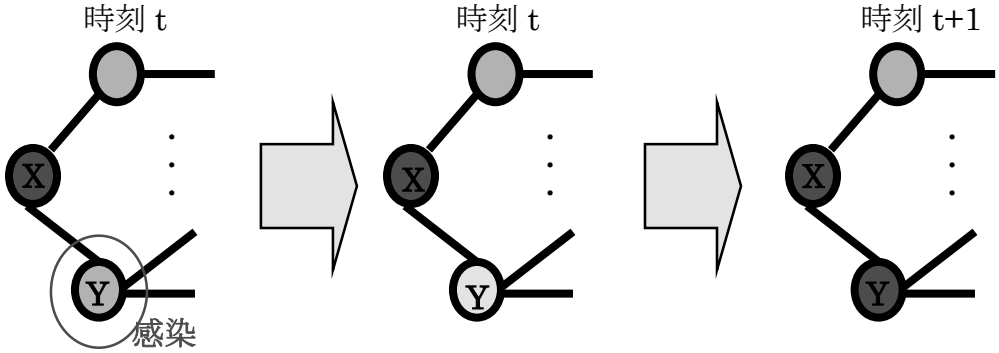


図 5.4　正しい状態遷移

5.2.3　シミュレーション回数

ウィルス伝搬シミュレーションは，25 種類のウィルス伝搬パラメータのそれぞれに関して 100 回ずつ行い，その結果をウィルス伝搬パラメータの組み合わせごとにそれぞれ 100 回分を平均化する．結果的に，1 つの CDD モデルに対して，合計 2500 回のウィルス伝搬シミュレーションを行うことになる．これは，ネットワークモデル作成同様，ウィルスの感染や免疫化は確率によって推移するため，同一パラメータで実験を行ったとしてもその結果にはさまざまな変化が現れるためであり，その 100 回のデータを平均化することによってデータを正当なものとして扱う．

5.2.4　初期感染ノードの選択

ウィルス伝搬シミュレーションには，シミュレーションを行うネットワークモデルと状態遷移のためのウィルス伝搬パラメータ（b と d ）の他に，ネットワーク内で最初に感染するノードが必要である．これを初期感染ノードとし，今回のウィルス伝搬シミュレーションでは，この初期感染ノードの選択を，ランダム選択，ハブ優先選択，betweenness centrality（betweenness）の最も高いノードの優先選択の 3 種類行う．

ここで，betweenness とは，ネットワーク内の負荷や，ネットワークの中心性と呼ばれ，近年注目を集めているパラメータである．ノード i の betweenness は，ノード i を除く，ネットワークノードのペア間の最短経路がノード i を通る回数と定義され，式 5-3 で表される．

$$C_B(i) = \sum_{w \neq w'} \frac{\sigma_{w,w'}(i)}{\sigma_{w,w'}} \qquad (5\text{-}3)$$

式 5-3 の $\sigma_{w,w'}(i)$ はノード i を通る最短経路数であり，$\sigma_{w,w'}$ は全体の最短経路数である．

今回，初期感染ノードを betweenness の最も高いノードとして実験を行った理由としては，betweenenss の最も高いノードはそれだけ多くの最短経路が通るノードなので，ハブとは違った理由から多くのノードと接触する機会が多いノードであると考えたためである．

ここで注意しなくてはならないことがある．それは，CDD モデル作成時に，ハブノードと betweenness の高いノードが同じノードになることがあるということである．実際に作成した CDD モデルの，次数と betweenness 値の相関を図 5.5 に示す.

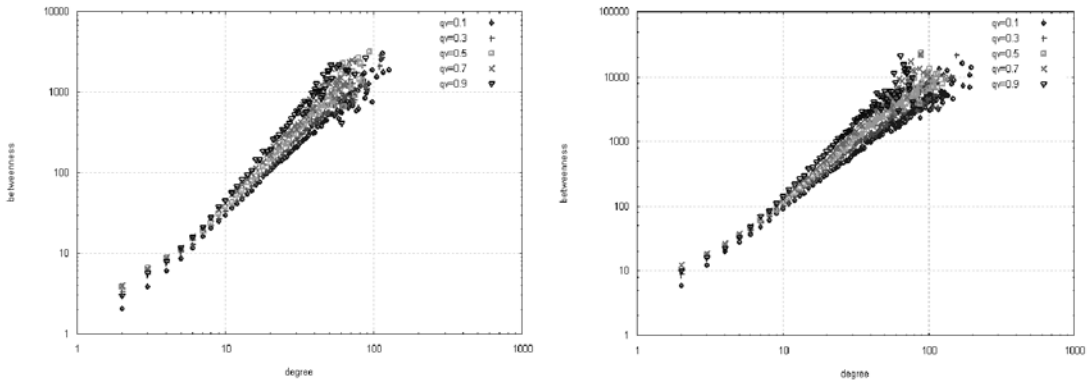


図 5.5　次数と betweenness の相関（N=1000（左），N=2500（右））

図 5.5 のように，今回作成した CDD モデルは高次数のノードは高い betweenness を持っているということがわかる．そのため，初期感染ノードを betweenness の高いノードと選択しシミュレーションを行う際，ネットワークモデル作成時に，もしハブが betweenness 最大のノードであったならばネットワークモデルを再度作り直すという処理を加えている．これによって初期感染ノードがハブの場合と，betweenness が高い場合を別々に分けることができる.

5.3　シミュレーション回数とタスク数

本節では，ネットワークモデル作成とウィルス伝搬シミュレーションで処理されるシミュレーション回数を求め，実際に分散処理環境で分散するタスク数について述べる．

5.3.1　システム全体の処理項目

5.1 節，5.2 節で述べてきたように，ネットワークモデル作成とウィルス伝搬シミュレーションには以下の処理が必要である．

- ネットワークモデル作成
  ネットワークモデルパラメータ 5 種類×100 個のモデル作成
- ウィルス伝搬シミュレーション
  ウィルス伝搬パラメータ 25 種類×100 回のシミュレーション

つまり，1 回のシステム全体の処理で，CDD モデルを 500 作成し，シミュレーションを, 500 モデル×2500 シミュレーションの合計 1250000 回行うということになる．

5.3.2　分散タスク数

ここで, 25 台のスレーブマシンに分配するタスクを考えた場合, この 1250000 回のシミュレーションを全て均等に分配するのは非常に非効率的である．なぜならば，あるネットワークモデルに関するウィルス伝搬シミュレーション（25×100 回）をスレーブ間で分配した場合，同じネットワークで感染確率bと免疫確率dの組み合わせに対する異なる状態遷移を行うために, 1 台のスレーブが作成したネットワークモデル情報（辺情報ファイル）を他のスレーブと共有する必要があるためである．また，その結果をまとめる際に手間がかかることや，あまり細かくタスクを分配した場合に通信のオーバヘッドが大きくなり処理効率が低下する可能性がある. そこで, 本実験ではスレーブに分配するタスクを，全体で作成する 500 個（ネットワークパラメータq_v 1 種類ごとに 100 個のネットワークモデル）のネットワークモデルとする．

5.4　システムの処理の流れ

本節では，実際に本システムが行う処理の流れについて説明する．

5.4.1　マスター・スレーブ間での処理の流れ

本システムでは，表 5.3 に示されるような流れでネットワークモデル作成，ウィルス伝搬シミュレーションを行う．本システムの処理の流れを，表 5.3 をもとに詳しく説明する．ここでは，必要となるクラスファイルが表 3.7 のように配置されているとする．また，以下の説明ではマスターとスレーブが直接やり取りをしているように記述するが，実際は各計算機の代理オブジェクトであるプロキシとスケルトンを介してやり取りを行っている．表中の矢印はマスター・スレーブ間のやり取りがあることを示す.

表 5.3　マスター・スレーブ間の処理の流れ

Master		Slaves
(1)HORB サーバとして起動		
	←	(2)クラスファイルの取得
(3)クラスファイルの返送	→	
	←	(4)モデル作成パラメータの取得要請
(5)モデル作成パラメータの返送	→	
		(6)ネットワークモデル作成
		(7)ウィルス伝搬シミュレーション
	←	(8)シミュレーション結果の送信
(9)シミュレーション結果の記録		(9)(4)へ

(1) HORB サーバとして起動

マスターは HORB サーバとして起動しておく必要があるため，コマンドラインから HORB コマンドを実行し HORB サーバとして起動する.

(2) クラスファイルの取得

各スレーブはネットワークモデル作成やウィルス伝搬シミュレーションに必要なクラスファイルを，HORBDynamicLoader を用いてマスターからロードする．実行ファイルは DL_Client.class である．

(3) クラスファイルの転送

マスターでは，HORBDeployServer が HORBDynamicLoader からの要求を受けて，カレントディレクトリにある storeDir ディレクトリ内のクラスファイルや JAR ファイルから必要なクラスファイルを返送する．この際返送されるクラスファイルは，Simulation_Client.class とその関連クラスの Simulation_Of_Infection.class, readGraph.class, Create_Model.class である．

(4) ネットワークモデル作成パラメータの取得要請

各処理に必要となるクラスファイルを取得した後，スレーブは Simulation_Client.class を実行し，ネットワークモデル作成パラメータの取得をマスターに要請する．

(5) モデル作成パラメータの返送

スレーブからのネットワークモデル作成パラメータの取得要請を受けたマスターは，スレーブに対してネットワークモデル作成パラメータと共に，

① Type：初期感染ノード（random，hub，betweenness から選択）
② No：モデルナンバー（作成するモデルの番号）
③ Simulation_No：b，d の組み合わせで行うシミュレーション数
④ System_Time：シミュレーションの終了時間

を返送する．

(6) ネットワークモデル作成

ネットワーク作成パラメータを受け取ったスレーブは，Create_Model によって CDD モデルを作成する．その際，出来上がったネットワークの総辺数が基準辺数の範囲内に収まらない場合，ネットワークを作り直す．また，初期感染ノードを betweenness 最大のノードとした場合は，作成したネットワークの次数

最大のノードと betweenness 最大のノードが同じノードであった場合，ネットワークを作り直す．作成したネットワークモデルは辺情報ファイルとして保存される．

(7) ウィルス伝搬シミュレーション

CDD モデルを作成した後，readGraph でその辺情報ファイルを読み込む．読み込んだ辺情報は，可変長配列 Vector 型のデータに頂点間の接続情報として格納される．次に，初期感染ノードを選択し，そのノードの状態を感染状態にする．初期感染ノードが決定したならば，Simulation_Of_Infection でウィルス伝搬シミュレーションを開始する．シミュレーションは 5.2.3 節で示した回数行う．

(8) シミュレーション結果の送信

ウィルス伝搬シミュレーションの結果得られる情報は

① システム時間ごとの未感染，感染，免疫状態にあるノード数
② システム時間ごとの感染ノードの平均次数
③ システム時間ごとの感染ノード内での最大次数
④ ネットワークモデル作成にかかった時間
⑤ ウィルス伝搬シミュレーションにかかった時間

(9) シミュレーション結果の記録

送信されてきた結果をファイルに出力する．

(9) (4)へ

スレーブは 1 つのネットワークに関してウィルス伝搬シミュレーションを終了した後，次のネットワークを作成し同様にシミュレーションを行う．

5.4.2　シミュレーション結果の記録ファイル

表 5.3 の“(9)シミュレーション結果の記録”では，25 台のスレーブから送信されてくるシミュレーション結果を混同や予期しない上書きなどを防ぐために，送信元のスレーブの持つ情報を使用してそれぞれのディレクトリを作成し，そのディレクトリに①から③の情報を格納する．

格納するディレクトリ名は，シミュレーションしたネットワークモデルパラメータ，ウィルス伝搬パラメータ，スレーブのホスト名，初期感染ノード，そしてモデルナンバーを使用してシミュレーション結果が重複しないように決定する．

ディレクトリ名：$cdd\_N\_q_v\_q_e\_b\_d\_No\_hostname\_Type$

というように命名する．戦闘のcddは Coupled duplication divergence の略である．例えば，ノード数 1000 で，(q_v, q_e)が(0.1,0.5)，(b, d)が(0.1,0.1)，モデルナンバーが 1，$hostname$が dolphin1，初期感染ノードがハブならば，そのシミュレーション結果を格納するディレクトリは

ディレクトリ名：$cdd\_1000\_0.1\_0.5\_0.1\_0.1\_1\_dolphin1\_hub$

となる．

ノード数と初期感染ノードを決定し，ネットワークモデル作成，ウィルス伝搬シミュレーションを行った場合作成されるディレクトリ数は，ネットワークモデル作成パラメータ数×100×ウィルス伝搬パラメータ数となり，合計で 12500 個できることになる．そして，作成したディレクトリに格納する実験結果のファイルは 5.4.1 の(8)で示した取り 3 種類存在し，それぞれ 100 個出力されるため，3750000 個のファイルが出力される．

さらに，これらの結果は実験するノード数や初期感染ノードによって同じ数だけ結果が出力される．

5.4.3　使用する乱数

本システムでは，ネットワークモデル作成でq_vとq_e，ウィルス伝搬シミュレーションでbとdという確率を使用している．確率的に操作を行ううえで必要となるのが乱数である．Java 言語にはもともと乱数を発生させる java.util.Random や Math.random が存在する．しかしながらこれらの乱数は周期が短く，Math.random に関しては乱数の種となる値を与えることができないことから，その乱数の再現性にかける．さらに Java では 48 ビット線形合同法を使用しているため，Java の処理の遅さに輪をかけた遅さになっている[14]．

今回本システムで使用している乱数は，Mersenne Twister とは，松本眞，西村拓士により 1996 年から 1997 年にわたって開発された擬似乱数生成アルゴリズムである．これまでの乱数にはない長周期（2 の 19937-1 乗，10 進数 6000

桁ほど）をもち，さらに乱数の種となる Seed を設定することによって，完全な再現性を持つ[15].

Java の乱数を使用した場合，周期の短さから，25 台のスレーブで同じネットワークモデルを作成したり，1 つのネットワーク内で同じウィルス伝搬シミュレーションを行う可能性がある．しかし，Mersenne Twister ほどの長周期性があれば Seed をうまく与えることによって，まったく異なるモデル作成やシミュレーションが可能である．

そこで，各スレーブで Mersenne Twister に与える Seed を完全に差別化するために，スレーブのホスト名を Hash コードで変換した値を使用する．それだけでは毎回同じ Seed の値になるため，処理の回数を記録した Operation_No をネットワークモデル作成ごとにインクリメントし，その Operation_No と Hash コードの積を Seed として使用する．

第6章 評価実験

本章では，今回作成した分散処理環境で行ったネットワークモデル作成とウィルス伝搬シミュレーションから得られた結果を示す.

6.1 ウィルス伝搬特性の解析

本節では，作成したネットワークモデル上で行ったウィする伝搬シミュレーションから得られた結果をもとに，ネットワーク上でのウィルス伝搬特性を解析する.

6.1.1 感染数の推移の比較

本節では，感染の経緯を示す感染数$I(T)$の推移をq_v，初期感染ノードで比較する.

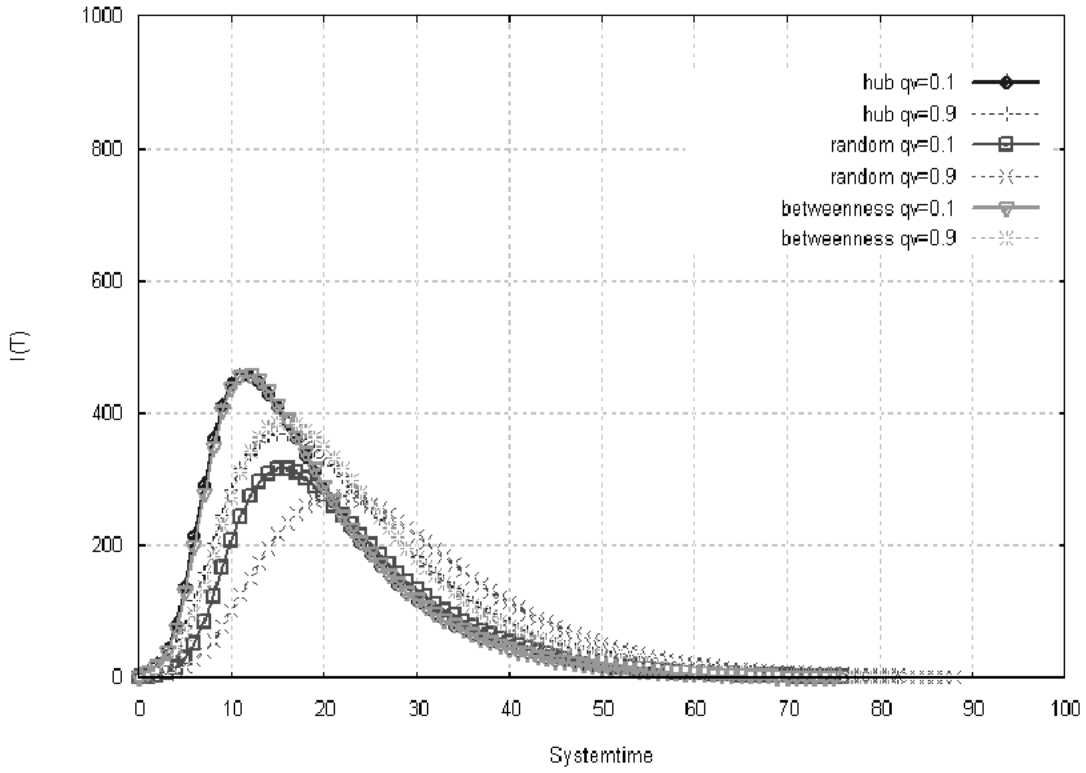


図 6.1 感染数の推移（$b = 0.1$， $d = 0.1$）

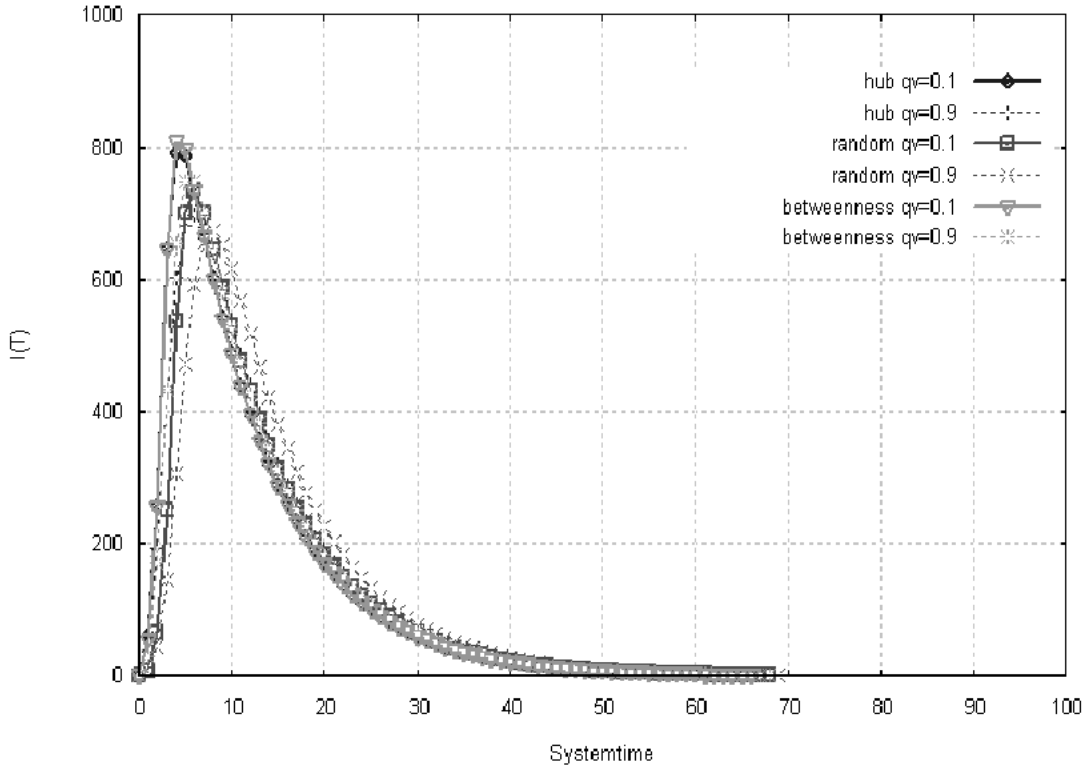


図 6.2　感染数の推移（$b = 0.9$，$d = 0.1$）

まず，図 6.1，6.2 の両方から言えることは，q_vが高くなると（破線のグラフ）$I(T)$の最大値が下がり，またその最大値にいたるまでのシステム時間が遅くなるということである．この理由としては，q_vが高くなる場合，平均結合相関が正の相関を持つため，低次数のノードは低次数のノードと接続される機会が増える．そのため，ネットワークの端のほうは低次数のノードの連結が主なものになる．よって，感染確率が高い場合でも感染の速度が鈍くなる．感染の速度が鈍るということは，免疫化される可能性も高くなるため，q_vが高いほど感染数の最大値が低くなり，最大値を記録する時刻が遅くなったと考えられる．

また, 今回作成した CDD モデルは, 基準辺数によって総辺数を制限している．そのためq_vが高くなる（正の結合相関）とハブとハブの結合が強くなり，q_vが低いときに比べて極端に次数の高いノードが生まれにくくなる．逆にq_vが低い場合（負の結合相関）は，高次数のノードと低次数のノードが接続されやすいため，超高次数のハブが生まれる可能性がある．これは，感染ノード内の最大次数を示した図 6.3，6.4 を見ても明らかである．

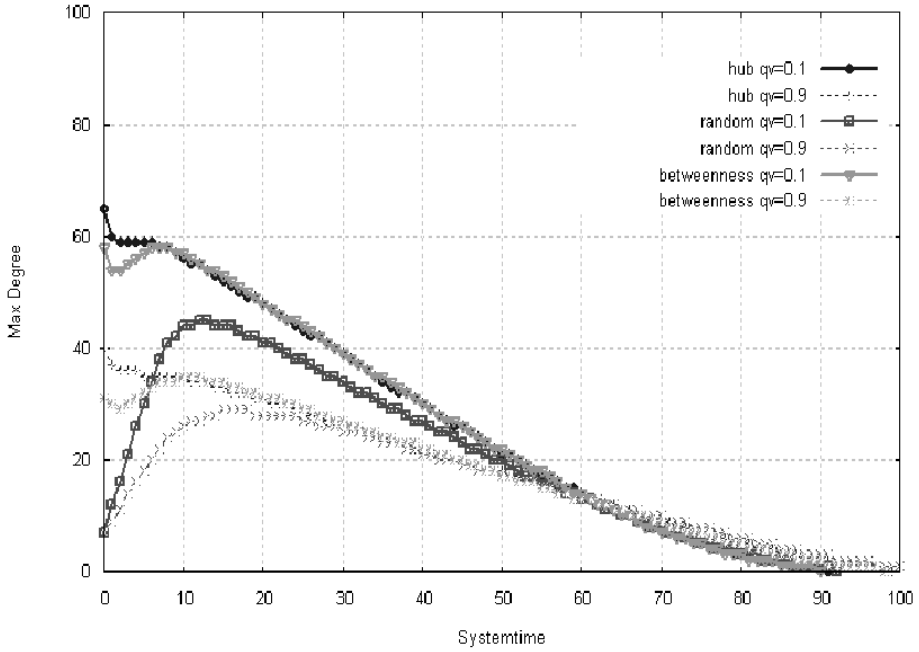


図 6.3　感染ノード内の最大次数（$b = 0.1$，$d = 0.1$）

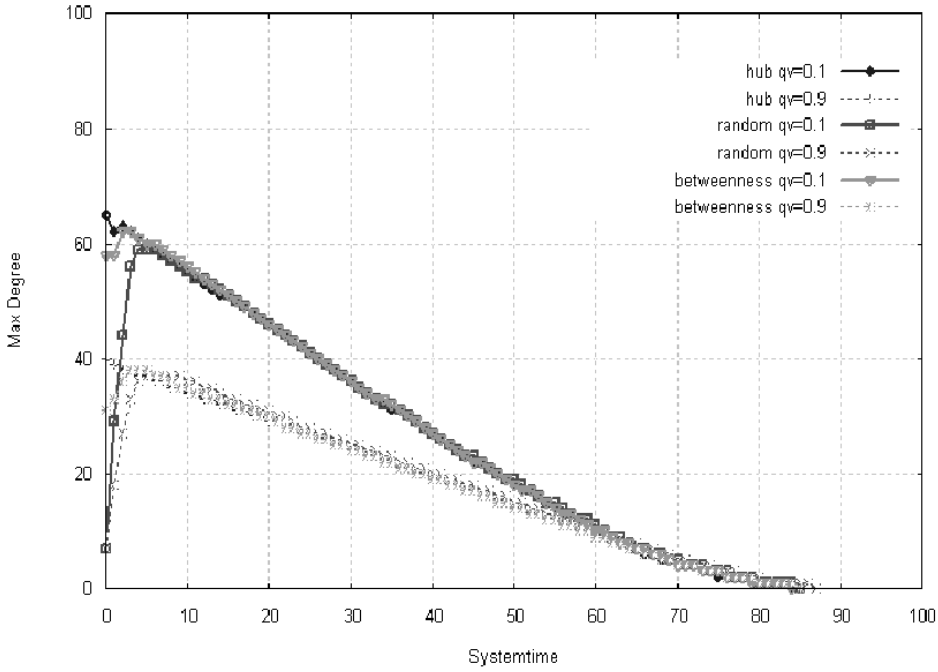


図 6.4　感染ノード内の最大次数（$b = 0.9$，$d = 0.1$）

図 6.3，6.4 より，q_vが高くなると（破線のグラフ），全体的に最大次数の値が低くなっていることがわかる．また，感染確率が高くなると，図 6.4 のように

初期感染ノードやq_vによらず非常に早い段階で高次数のノードに感染し，その後の傾向は全て同じようなものになる．早い段階で高次数のノードの感染するということは，それだけ感染拡大が早いということを表し，それは先程の図 6.2 を見ても明らかである．

6.1.2　総感染数の比較

今回行ったウィルス伝搬シミュレーションでは，初期感染ノード数を 1 として，最終的に感染ノード数が 0 になった時点でシミュレーションを終了する．感染ノード数が 0 になった時刻をTとし，感染規模を表す総感染数は時刻Tまでに免疫化されたノード数であるので，$R(T)$とする．ここで，対結線確率q_v，感染確率b，免疫確率dの違いによる$R(T)$の変化を図 6.5 に示す．

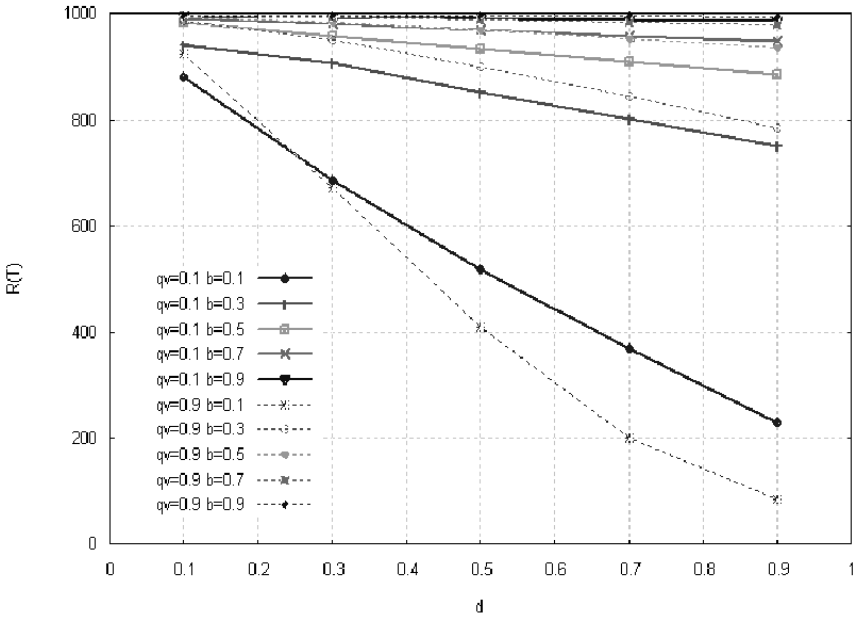


図 6.5　$R(T)$の変化①（$N=1000$，初期感染ノード：ハブ）

図 6.5 は，横軸を免疫確率d，縦軸を$R(T)$とし，実線が$q_v=0.1$のグラフで，破線が$q_v=0.9$のグラフである．図からわかることは，q_vが高い（正の結合相関を持つ）方が総感染数は多く傾向にあるということである．q_vが高いということは高次数ノード（ハブ）同士の接続が多いため，高次数ノードが感染する確率が高くなる．そうした場合，ある程度感染確率bが高くなると，感染の速度

が速くなり，感染規模が拡大すると考えられる．

しかしながら，感染確率bが極端に低い場合（図中青線），q_vが高い（正の結合相関を持つ）方の総感染数が，q_vが低い（負の結合相関を持つ）場合よりも少なくなっている．このq_vと$R(T)$の関係の逆転は，今回の実験で新しく発見された事実であり，原因として以下の理由が考えられる．

感染確率bが極端に低い場合，高次数のノードが感染したとしても，感染が広がる前に免疫化されてしまう可能性が高い．高次数のノードが感染を広げる前に免疫化されるということは，ウィルスの感染経路が極端に少なくなることを意味し，結果として感染の拡散が一気に沈静化するということが考えられる．

次に，初期感染ノードの選択による$R(T)$の変化を図 6.6 に示す．

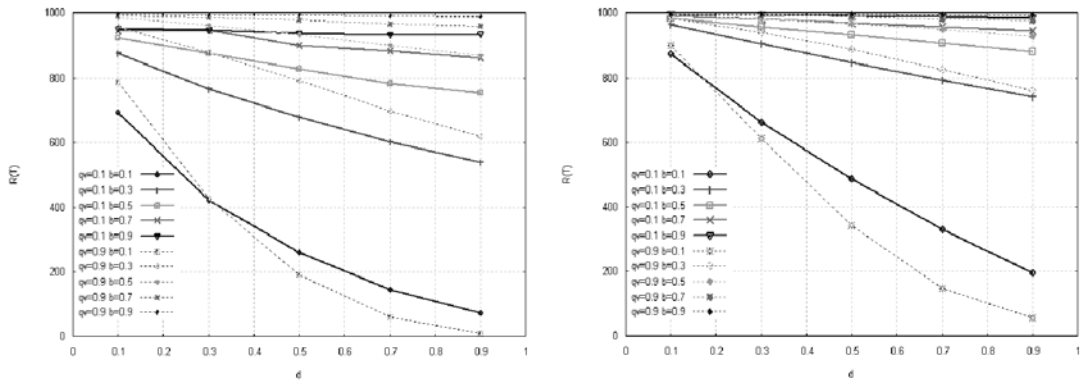


図 6.6　$R(T)$の変化②（初期感染ノード：ランダム（左），betweenness（右））

図 6.5 と図 6.6 を比較してみると，初期感染ノードをランダムに選んだ場合，感染の規模が小さくなっていることがわかる．しかし，$R(T)$の規模の推移の傾向は同じような結果が出ており，ここでも感染確率bが極端に低い場合，q_vが高い方の総感染数$R(T)$が，q_vが低い場合の結果よりも少なくなっている．

次に，ノード数 2500 の場合の結果を示す．

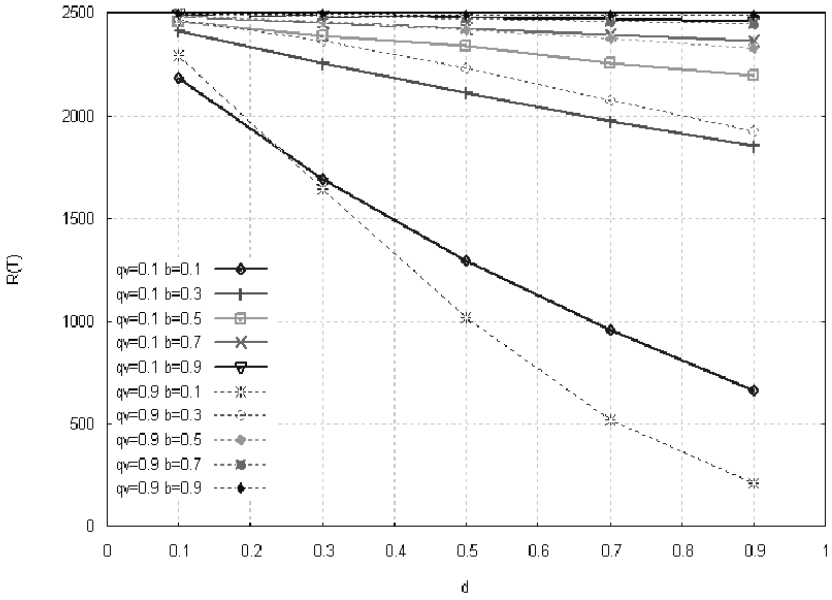


図 6.7　$R(T)$の変化③（$N = 2500$，初期感染ノード：ハブ）

図を比較すると，ノード数を増やし，ネットワークサイズを大きくしても，ネットワークサイズが大きい分$R(T)$の値は大きくなるが，推移の傾向は同じような結果であった．初期感染ノードをランダム，及び betweenness 最大のノードとした場合（図 6.6 と図 6.8）も，それは同様である．

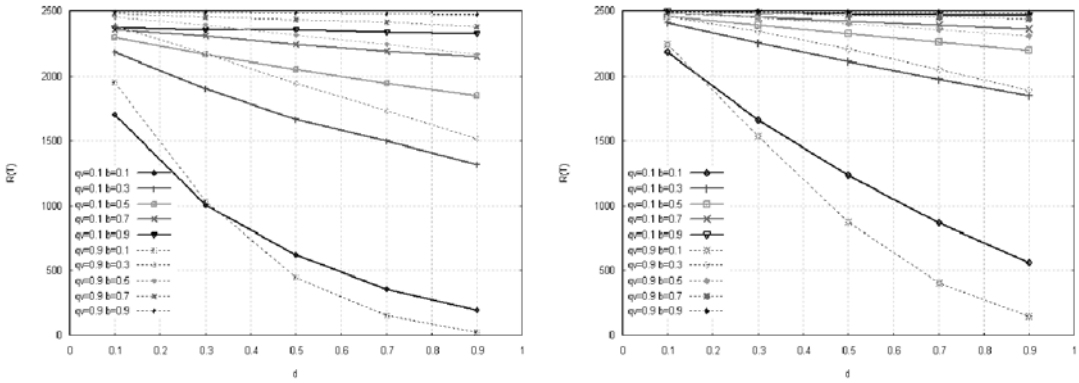


図 6.8　$R(T)$の変化④（初期感染ノード：ランダム（左），betweenness（右））

以上の結果から，CDD モデルでは，基本的にはセルフインタラクションが強い場合（正の結合相関ほど），感染の規模が拡大しやすい．そして，その傾向はネットワークサイズや初期感染ノードの種類によらない．しかしながら，ウィルスの感染力（感染確率）が極端に低い場合に限って，セルフインタラクショ

ンが強い場合の方が感染の規模が小さくなるという例外が発生する．

さらに，初期感染ノードをハブとした場合と betweenness が高いノードとした場合では，その傾向にほとんど差が現れなかった．CDD モデルにおいては，ハブを優先的に感染させた場合と，betweenness の高い（中心性，負荷が高い）ノードを優先的に感染させた場合では，同じ程度の規模で感染が起こるということがわかる．これは，図 5.5 で示すように，今回作成した CDD モデルにおいて，高次数ノードと betweenness の高いノードの間に正相関が成り立つためだと推測される．近年，ウィルスの免疫化の研究でハブの優先的な免疫化によって感染の沈静化を図るという研究がなされてきているが，今回の実験結果よりウィルス感染の被害を抑えるために，ハブ以外にも betweenness の高いノードを優先的に免疫化する必要があると考えられる．

6.2　処理時間の検証

本研究のもう 1 つ目的は，分散処理環境の実現にあり，これまでの述べてきたネットワークモデル作成やウィルス伝搬シミュレーションは，その評価のための実験として捉えることもできる．本節では，上記のシミュレーションにかかった処理時間をもとに，本システムの性能を評価する．また，単純な負荷分散と，比較的簡単なスケジューリングを行った場合の処理時間から，負荷分散について考察する．

6.2.1　分散処理環境の処理効率

実験では，25 台のスレーブに 5.3.2 節で説明したタスクを分散し，実験を行った．この分散環境で行った場合の処理時間と，スレーブと同じ状態にある計算機 1 台で全てのタスクを処理した場合の処理時間を比較する．処理時間は，

- Create_Time：ネットワークモデル作成時間
- Simulation_Time：状態遷移（感染伝搬）シミュレーション時間
- Execute_Time：全体の処理時間

の 3 種類比較する．

表 6.1，6.2 にノード数 $N=1000$，初期感染ノードをハブとした場合の処理時間の比較データを示す．

表 6.1　処理時間［msec］（ノード数：1000，初期感染ノード：ハブ）

	Execute	Create	Simulation
単体処理			
(1) スレーブ 1 台	130795790	3781566	126999654
(2) (1) / 25	5231831.6	151262.64	5079986.16
分散処理			
(3) 最短処理時間	6588873	108649	6456052
(4) 最長処理時間	6855236	195991	6695378
(5) 平均処理時間	6707507	149126	6557924
(3) /(2)	1.259381705	0.718280469	1.27087984
(4) /(2)	1.310293703	1.295699982	1.317991386
(5) / (2)	1.282057129	0.985874635	1.290933438
(5) / (1)	0.051282285	0.039434985	0.051637338

表 6.2　処理時間（ノード数：1000，初期感染ノード：ハブ）

	Execute	Create	Simulation
単体処理			
(1) スレーブ 1 台	36 時間 19 分 55 秒	1 時間 3 分 1 秒	35 時間 16 分 39 秒
(2) (1) / 25	1 時間 27 分 11 秒	0 時間 2 分 31 秒	1 時間 24 分 39 秒
分散処理			
(3) 最短処理時間	1 時間 49 分 48 秒	0 時間 1 分 48 秒	1 時間 47 分 36 秒
(4) 最長処理時間	1 時間 54 分 15 秒	0 時間 3 分 15 秒	1 時間 51 分 35 秒
(5) 平均処理時間	1 時間 51 分 47 秒	0 時間 2 分 29 秒	1 時間 49 分 17 秒

表 6.1, 6.2 よりスレーブ 1 台で処理を行った場合の処理時間をスレーブ数 25 で割った（2）のデータは，25 台で分散処理する場合の理想的な値であり，実際に分散処理を行った場合，その理想的な値よりは時間がかかる．それは，マスター・スレーブ間でのデータのやり取りや，セッションの確立など異なる計算機の間で通信を行うためスループットが発生するが原因であるといえる．また，

それ以外にもマスター側での結果ファイルの格納の際に行うファイル操作なども処理時間の遅れの原因であると考えられる. 表 6.3, 6.4 にノード数 $N = 2500$, 初期感染ノードをハブとした場合の処理時間の比較データを示す.

表 6.3　処理時間［msec］（ノード数：2500，初期感染ノード：ハブ）

	Execute	Create	Simulation
単体処理			
(1) スレーブ 1 台	592882100	17928836	574938518
(2) (1) / 25	23715284	717153.44	22997540.72
分散処理			
(3) 最短処理時間	23500675	546967	22712967
(4) 最長処理時間	24634126	1131780	24026797
(5) 平均処理時間	24127062	731397	23395324
(3) /(2)	0.990950604	0.762691733	0.987625906
(4) /(2)	1.038744718	1.578155994	1.044755058
(5) / (2)	1.017363402	1.019861245	1.017296775
(5) / (1)	0.040694536	0.04079445	0.040691871

表 6.4　処理時間（ノード数：2500，初期感染ノード：ハブ）

	Execute	Create	Simulation
単体処理			
(1) スレーブ 1 台	164 時間 41 分 22 秒	4 時間 58 分 48 秒	159 時間 42 分 18 秒
(2) (1) / 25	6 時間 35 分 15 秒	0 時間 11 分 57 秒	6 時間 23 分 17 秒
分散処理			
(3) 最短処理時間	6 時間 31 分 40 秒	0 時間 9 分 6 秒	6 時間 18 分 32 秒
(4) 最長処理時間	6 時間 50 分 34 秒	0 時間 18 分 51 秒	6 時間 40 分 26 秒
(5) 平均処理時間	6 時間 42 分 7 秒	0 時間 12 分 11 秒	6 時間 29 分 55 秒

表 6.3 より，ノード数$N=1000$の場合に比べて，ノード数$N=2500$での分散処理時間は理想的な値に近づいたといえる．次に，ノード数$N=1000$とノード数$N=2500$の処理時間を比較した結果を表 6.5 に示す．

表 6.5　処理時間の変化と比率

	Execute	Create	Simulation
単体処理			
(1) N=1000	130795790	3781566	126999654
(2) N=2500	592882100	17928836	574938518
分散処理(1 台あたりの平均)			
(3) N=1000	6707507	149126	6557924
(4) N=2500	24127062	731397	23395324
比率			
(2)/(1)	4.532883665	4.741114131	4.527087279
(4)/(3)	3.597023753	4.90455722	3.567489346

表 6.5 において$N=1000$と$N=2500$を比較すると，ノード数が 2.5 倍になっているのに対して，単体処理の処理時間は約 4.5 倍となっている．それに比べて分散処理では，平均の処理時間が約 3.6 倍となっている．これは，分散処理によって，単体で処理する場合に比べて，各計算機の要する処理時間の増加を抑えることができたということを示している．この理由としては，

- 単体の計算機ですべての処理を行う場合，マスターとスレーブの処理をすべて行わなくてはならない．つまり，本来はマスター計算機の処理である各パラメータの割り当てや，シミュレーション結果のファイル出力などを単体で行う必要がある．
- また，ノード数の増加に伴ってネットワークモデルの辺情報ファイルの容量も大きくなる．単体で処理を行う場合に比べて，分散処理ではこの辺情報ファイルの作成と読み込みの回数が 25 分の 1 回で済む．
- 分散処理では，スレーブはスレーブの処理のみに専念するため，メモリの消費量が少なくて済む．

などが挙げられる．

6.2.2　タスク分散手法による処理時間のばらつき

分散処理では，タスクの分散手法は非常に重要なテーマである．タスク分散の目的としては，効率よくタスクを処理し，資源の有効利用と高速な処理の実現が挙げられる．資源の有効利用と高速な処理を実現するためには，処理を行う計算機が満遍なく動作していることが望ましい．たとえば，非常に忙しく動作しているものと何もしていないアイドル状態のものがあったのでは，効率的な処理が望めない．つまり，分散処理を行う計算機間の処理時間にばらつきがないほうが，システム全体が効率的に動作しているといえる．

そこで今回，以下の 2 つのタスク分散手法で実験を行い，各手法でのスレーブの処理時間のばらつきを測定する．そして，測定結果を表 6.6 から表 6.11 に示す．なお，表 6.6 から表 6.8 はノード数$N=1000$で初期感染ノードを変化させた結果，表 6.9 から表 6.11 にはノード数$N=2500$で初期感染ノードを変化させた結果を示す.

- 負荷の均一分散

これは，いたって単純な負荷分散である．全体のタスクは作成するネットワーク 500 個分なので，500 をスレーブの台数で割った一定数のタスクをスレーブに割り当てる手法である．

正確に言うと，ネットワークモデル作成パラメータ 5 種類に関してそれぞれ 100 個ずつモデルを作成するため，1 種類のネットワークモデルを各スレーブがそれぞれ 4 個ずつ作成するというものである．つまり，スレーブはそれぞれ 20 個ずつネットワークモデルを作成し，ウィルス伝搬シミュレーションを行うことになる．

- 単純なスケジューリング手法

Receiver Initiation 法と呼ばれるもので，これも単純な負荷分散である．仕事がないスレーブに対して，マスターが 1 つずつタスクを渡す方式である．スレーブは 1 つのタスクを処理し終えるとまたマスターに 1 つタスクを割り当ててもらう．

表 6.6　N=1000　初期感染ノード：ハブ

	負荷の均一分散	スケジューリング
最短モデル作成時間	84311	108649
最長モデル作成時間	209556	195991
平均モデル作成時間	148904	149126
最短シミュレーション時間	6046114	6456052
最長シミュレーション時間	6318441	6695378
平均シミュレーション時間	6220860	6557924
最短処理時間	6192686	6588873
最長処理時間	6519317	6855236
平均処理時間	6301709	6707507
モデル作成標準偏差	7119	23301
シミュレーション時間標準偏差	71672	64787
処理時間標準偏差	86675	65850

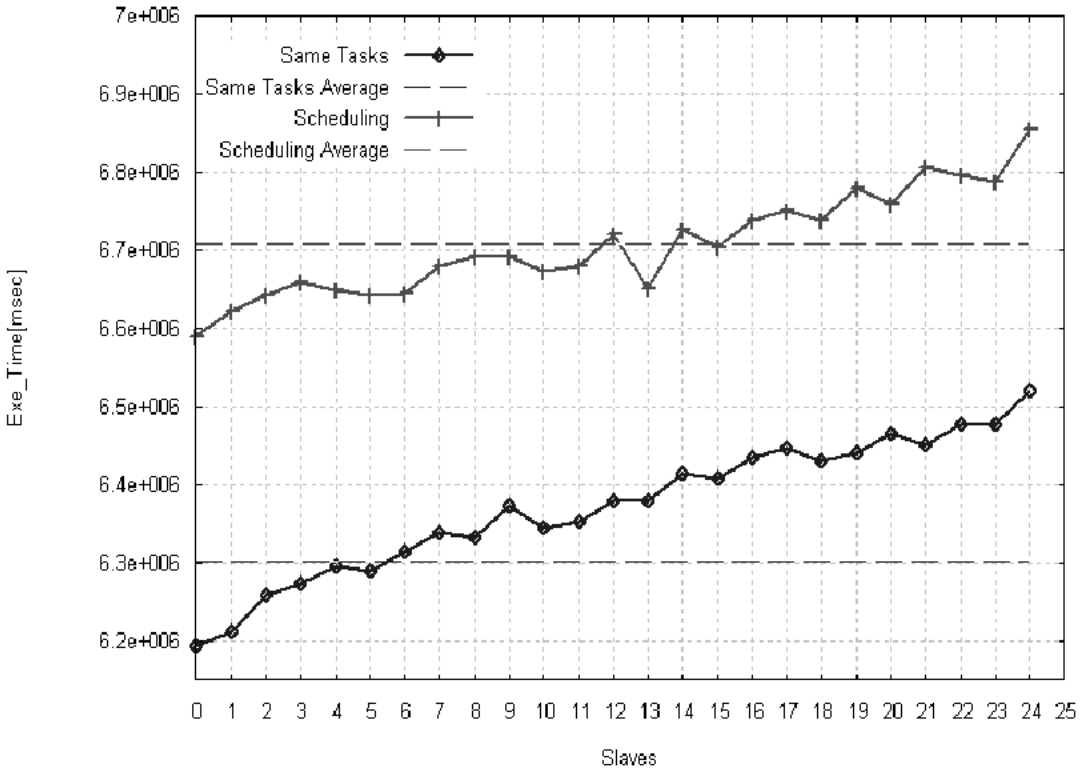


図 6.9　処理時間と平均値（N=1000　初期感染ノード：ハブ）

表 6.7　N=1000　初期感染ノード：ランダム

	負荷の均一分散	スケジューリング
最短モデル作成時間	84683	107333
最長モデル作成時間	210903	205895
平均モデル作成時間	149480	152141
最短シミュレーション時間	5726506	5854595
最長シミュレーション時間	5985638	6080245
平均シミュレーション時間	5869375	5938140
最短処理時間	5873728	5973797
最長処理時間	6146577	6227454
平均処理時間	6026867	6090725
モデル作成標準偏差	33102	25454
シミュレーション時間標準偏差	59504	58751
処理時間標準偏差	75467	62624

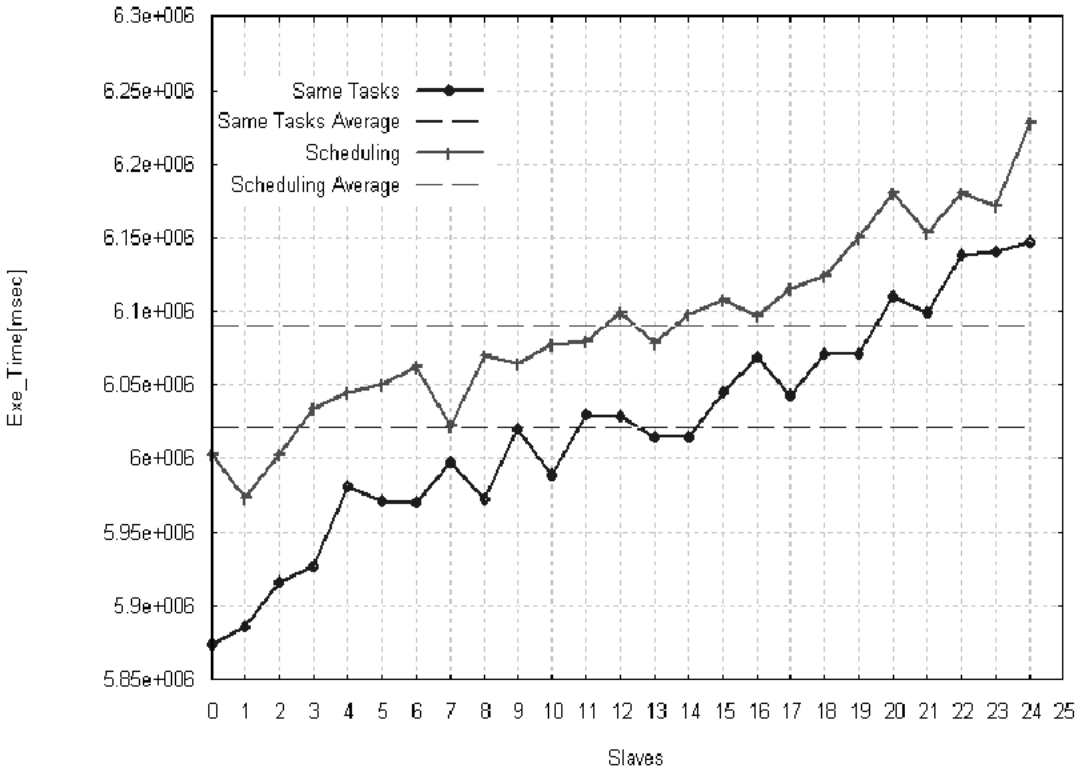


図 6.10　処理時間と平均値（N=1000　初期感染ノード：ランダム）

表 6.8　N=1000　初期感染ノード：betweenness

	負荷の均一分散	スケジューリング
最短モデル作成時間	641144	641237
最長モデル作成時間	1581646	1264430
平均モデル作成時間	1004167	951782
最短シミュレーション時間	6393435	5679379
最長シミュレーション時間	6631844	6613874
平均シミュレーション時間	6529976	6226606
最短処理時間	7111468	6993305
最長処理時間	8053195	7333409
平均処理時間	7535989	7178749
モデル作成標準偏差	256851	25454
シミュレーション時間標準偏差	63433	251970
処理時間標準偏差	250180	91016

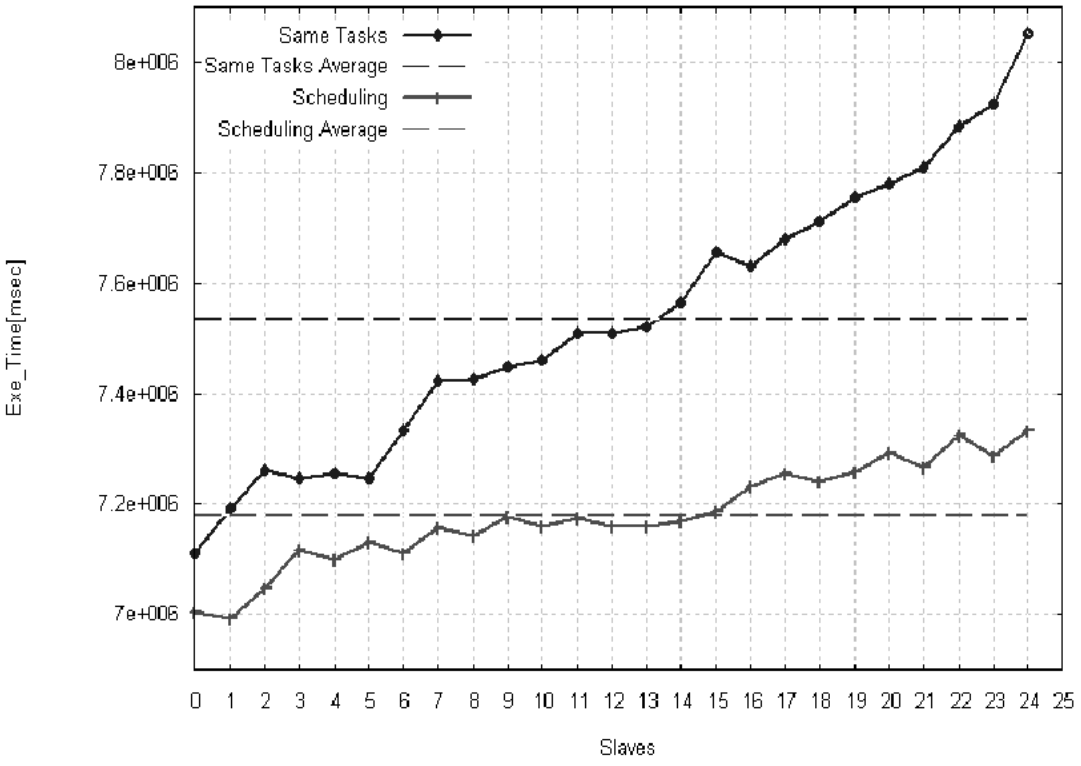


図 6.11　処理時間と平均値（N=1000　初期感染ノード：betweenness）

表 6.9　N=2500　初期感染ノード：ハブ

	負荷の均一分散	スケジューリング
最短モデル作成時間	463714	546967
最長モデル作成時間	957219	1131780
平均モデル作成時間	686645	731397
最短シミュレーション時間	22499422	22712967
最長シミュレーション時間	24596312	24026797
平均シミュレーション時間	23492375	23395324
最短処理時間	22964726	23500675
最長処理時間	25261361	24634126
平均処理時間	24180554	24127062
モデル作成標準偏差	147005	126490
シミュレーション時間標準偏差	463659	411184
処理時間標準偏差	490755	362255

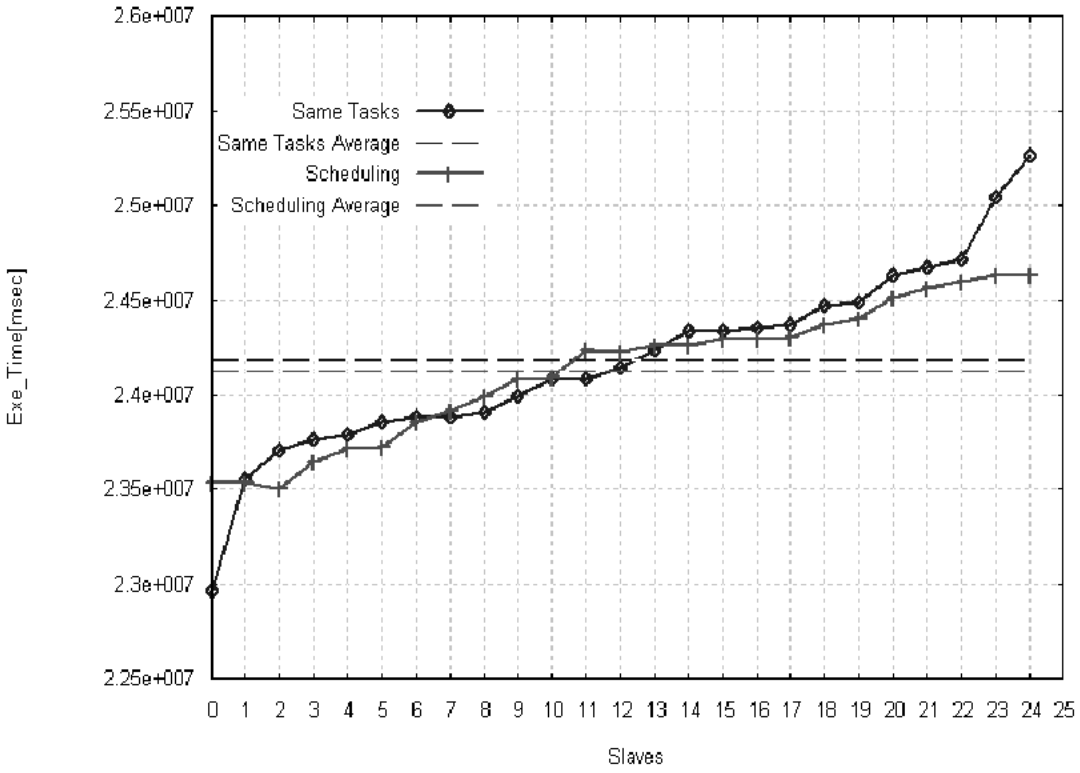


図 6.12　N=2500　初期感染ノード：ハブ

表 6.10　N=2500　初期感染ノード：ランダム

	負荷の均一分散	スケジューリング
最短モデル作成時間	464448	502278
最長モデル作成時間	952920	1006367
平均モデル作成時間	685554	720932
最短シミュレーション時間	20881817	20692931
最長シミュレーション時間	22219713	22144519
平均シミュレーション時間	21419588	21366023
最短処理時間	21363382	21629228
最長処理時間	22803741	22835341
平均処理時間	22106643	22087292
モデル作成標準偏差	146603	139467
シミュレーション時間標準偏差	375937	373981
処理時間標準偏差	404640	362036

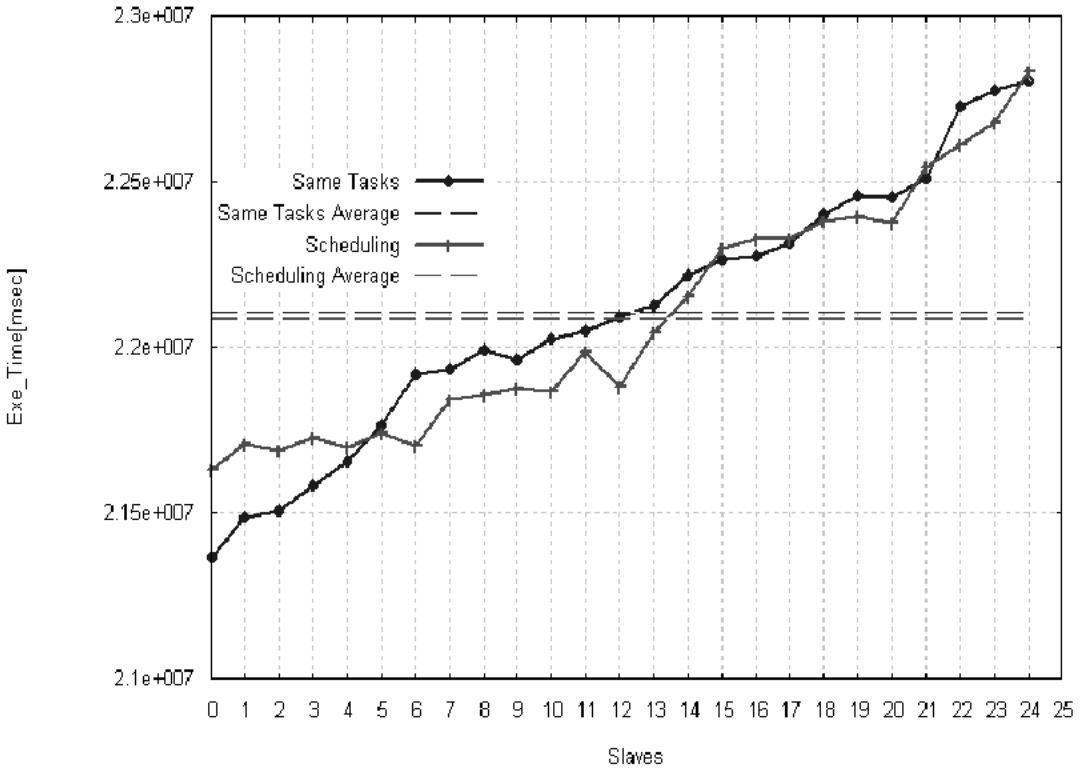


図 6.13　N=2500　初期感染ノード：ランダム

表 6.11　N=2500　初期感染ノード：betweenness

	負荷の均一分散	スケジューリング
最短モデル作成時間	3502329	3383582
最長モデル作成時間	6600966	7153866
平均モデル作成時間	4922700	4799460
最短シミュレーション時間	24285905	22552102
最長シミュレーション時間	25648063	26933588
平均シミュレーション時間	24922399	25121741
最短処理時間	28133241	29216775
最長処理時間	31362815	30666056
平均処理時間	29846617	29921521
モデル作成標準偏差	842077	1026840
シミュレーション時間標準偏差	382845	1140077
処理時間標準偏差	901469	435308

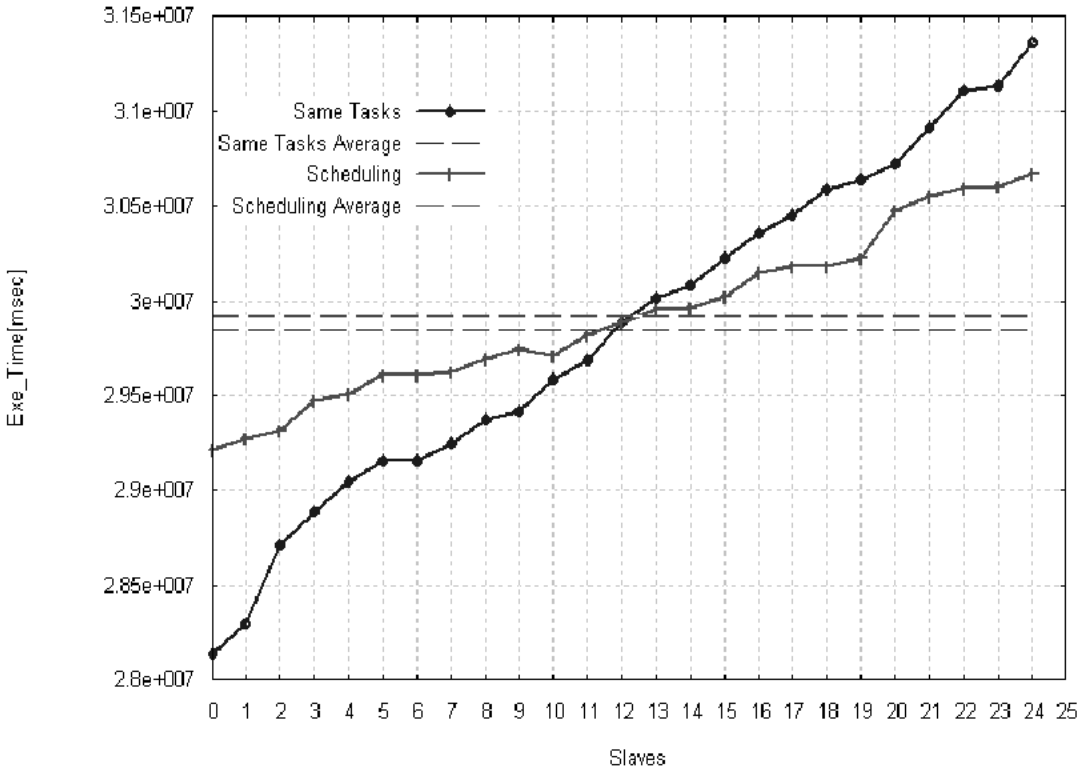


図 6.14　N=2500　初期感染ノード：betweenness

以上の結果から，単純なスケジューリングによって処理時間全体のばらつきを抑えることができたといえる．特に初期感染ノードを betweenness の高いノードとしたときが効果的である．これは初期感染ノードを betweenness の高いノードとしたとき，ネットワークの総辺数が基準辺数の範囲内であることに加えて，最大次数のノードが betweenness 最大のノードである場合にはネットワークモデルを作り直す，という制約がつくためである．そのため，初期感染ノードを betweenness の高いノードとしたとき，ネットワークモデルの作成しなおしの回数が増えてしまう．負荷の均一分散では，ネットワークモデル作成数が全スレーブで同数であるため，スレーブの作る乱数によっては，スレーブ間で処理の進行状況に大きな差が生まれてしまう．スケジューリングのほうでは，処理が終わったスレーブが次のタスクを受け取るため，作業効率が遅いスレーブの分を，早いスレーブが補うことができる．そのため，処理時間のばらつきを抑えることができた．

第7章 まとめ

7.1 実験結果のまとめ

7.1.1 ウィルス伝搬特性の解析

実験結果より，正の結合相関を持つネットワークでは，ウィルスの感染による被害が大きくなるということがわかった．しかしながら，ウィルスの感染力が極端に低い場合に限り，正の結合相関を持つネットワークの総感染数が，負の結合相関を持つネットワークの総感染数よりも少なくなるという例外が発生するということがわかった．この結果から，結合相関がウィルスの感染による被害の規模に影響を与えることが示された．つまり，従来注目されてきたウィルスの感染力に加えて，ネットワークの構造上の特徴がウィルス伝搬に影響を与えるということを示している．

また，今回作成した Coupled duplication divergence モデルにおいては，ハブと betweenness の高いノードを初期感染源とした場合，被害の規模を表す総感染数の規模と推移の傾向がほぼ同等の結果を示した．このことから，従来ウィルスの伝搬やウィルス対策の中心であったハブの存在に加えて, betweenness の高いノードがウィルス伝搬に大きな影響を与えるということがわかった．

7.1.2 処理時間の検証について

今回の実験をとおして，単体の計算機で全ての処理を行う場合に比べて，分散処理を行った場合，ネットワークのノード数の増加（負荷の増加）に伴う処理時間の増加を抑えることができたといえる．実際にノード数 $N=1000$ の結果では, 単体の計算機と比べて処理時間を約 20 分の 1 まで短縮できたのと比べて，$N=2500$ の結果では，単体の計算機と比べて処理時間を約 25 分の 1 まで短縮することができた．後者の結果は，単体の計算機の処理を 25 台の計算機で分散した場合の理想的な処理時間とほぼ同等の値である．このことから，本研究で作成した分散処理環境では，効率的に処理を行うことができたといえ，全体の計算負荷が大きくなった場合，単体の計算機と比べてより効率的に処理することができたといえる．

また，負荷分散の違いによる処理時間の比較では，単純なスケジューリング

でスレーブの実行時間のばらつきを抑えることができた．特にネットワークモデルの作り直しが多く必要な場合にはその効果を発揮するといえる．実際に，ネットワークモデル作成に制約の多い初期感染ノードを betweenness の高いノードとした場合，ばらつきを大きく抑えることができた．

7.2 考察

今回の実験をとおして，Java 言語と分散オブジェクト HORB を用いて比較的に容易に分散処理環境を構築することができた．ここでは，分散処理環境構築の過程で遭遇した問題点と改善策について述べる．

7.2.1 異なる計算機間で受け渡せるデータの制限

今回作成した分散処理環境では，マスターとスレーブの異なる計算機間でデータの受け渡しをしながら処理を行う．今回プログラム中で，ネットワークの情報を格納するために，可変長配列 Vector 型の変数を用いている．プログラム構築当初は，この Vector 型の配列をマスター・スレーブ間で受け渡す予定であった．しかし，HORB では，Object，String，boolean，byte，char，double，int，float，long はサポートしているが，異なる計算機間（プロキシとスケルトンの間）で Vector 型の変数のデータ転送をサポートしていないということがわかった．これは細かな問題であるが注意事項として記しておく．

7.2.2 HORB のコネクションのダウン

これは，表 5.3（8）のシミュレーション結果をマスターへ送信するときに起こったエラーである．実験の最中に，突然スレーブのうちの何台かが“Exception in thread “main” horb.orb.IOCIException:IOCI version mismatch”というエラーを出力して停止してしまうというものであった．HORB のコミュニティーメンバーとのやりとりから“HORB のクラスのバージョンの不一致”との指摘もあったが，実際はスレーブ側からマスターへ頻繁にデータの返信を行っていたために起きたエラーであり，データ返信の回数を減らすように改善すると正常に動作するようになった．

7.2.3　Java　VM のクラッシュ

このエラーは，実験の最中に突然起こるエラーであり，現在も原因が不明のエラーである．しかも，同じスレーブでも起こる場合と起こらない場合があり，また起こるとしてもそのタイミングが一定ではないという，非常に改善が難しいエラーであるといえる．未だに解決していないが，スレーブからマスターへの結果の返信や，ウィルス伝搬シミュレーション時のメモリの消費が原因ではないかと考えられる．

7.2.4　ディレクトリ数の限界点

5.4.2 節で説明したとおり，本システムで行うネットワークモデル作成とウィルス伝搬シミュレーションでは，シミュレーションの結果がディレクトリ数で 12500 個出力される．本システムを構築する計算機の OS は表 3.1 のとおりだが，それぞれの Linux には，1 つのディレクトリ内に作成可能なファイル数（ディレクトリも含む）が決まっている．スレーブに使用した Vine-Linux では 5000 を超える量を格納した場合，それ以上ディレクトリやファイルを作成することができなくなる．マスターの RedHat-Linux ではその限界数が 30000 を超えたところにあるということがわかった．また，そのディレクトリに格納されるファイルの容量も思いのほか大きく，1 つのディレクトリで 1M 強のサイズになる．そのため 12500 ディレクトリで 10GB を超えてくる．

そのため，計算量もさることながら記憶するデータの量も非常に多いため，1 台の計算機で行うには過負荷な問題であるといえる．

7.3　今後の課題

本研究では，比較的容易に高速処理を行える分散処理環境を構築できたといえる．しかし，7.2.3 節で述べたような問題点も存在する．この問題は，より大規模な計算を行う場合，プログラムが停止してしまうかもしれない致命的な問題になりうる．

さらに，今回実験した負荷分散は非常に単純なものであり，現在さまざまな負荷分散の研究が進んでいる．そういった負荷分散を使用した実験を行う必要があるといえる．

また，今後スレーブの増台や，ローカルネットワークよりももっと広域な環境での分散処理を行う場合，ソフト面やハード面で対応できるようにしなくてはならない.

謝辞

本研究を進めるにあたって，数多くの方々からご支援をいただきました．改めまして，深く感謝の意を表したいと思います．

本研究の指導教官である林幸雄助教授には，研究の進め方はもちろんのこと，論文のまとめ方や研究者としてのあり方など，さまざまなご指導を賜りました．2 年近くの歳月でしたが，私にとっては非常に意義のある学生生活だったと感じられます．本当にありがとうございました．

また，研究室のメンバーにも多くの面で支えてもらいました．研究へのアドバイスや計算機やソフトウェアの使い方などたくさんのことを教えていただいた松久保潤さん，また研究を進める上でネットワーク分析ツールを提供してくれた宮崎敏幸君，さまざまなアドバイスをくれた中村克久君に，特に深く感謝します．ありがとうございました．

ここ JAIST は，高専の専攻科時代とはまた違う，厳しくも楽しく有意義な時間を過ごさせてくれた場所だと思います．未熟な私をここまで支え，導いていただいた皆様に心より感謝致します．

最後に，私のわがままを許し，ここ JAIST ですばらしい経験をつむ機会を与えてくれた両親に心より感謝致します．

参考文献

［1］ 伊藤正敬，林幸雄，“ORB 分散コンピューティングを用いた Web リンク収集”， 信学総合大会， D-6-17， Mar.27-30， 2002.

［2］ 秋田晋吾，“分散環境における動的負荷の均一化に関する研究”，北陸先端科学技術大学院大学　知識科学研究科　2003 年 2 月.

［3］ ジェミー・ジョウォルスキー[著]，堀内泰輔[訳]，“Java プログラミング技法対話型WWW開発の全て”，株式会社プレティスホール出版.

［4］ 平野聡，“ネットワークコンピューティングの魔法のじゅうたん HORB Flyer's ガイド” ，
http://horb.a02.aist.go.jp/horb-j/doc/guide2.0/guide.htm

［5］ 萩本順三，“ＨＯＲＢではじめる Java 分散オブジェクトプログラミング”，株式会社秀和システム.

［6］ “HORB 工房”，http://www2t.biglobe.ne.jp/~o_kiku/

［7］ Alexei Vazquez，“Growiing network with local rules:Preferential attachment, Clustering, hierarchy, and degree correlations”，The American Physical Society 2003.

［8］ アルバート=ラズロ・バラバシ［著］，青木薫［訳］，“新ネットワーク思考～世界の仕組みを読み解く～”，日本放送出版協会.

［9］ 林幸雄，“成長するネットワークの生態学 - その共通構造と伝播の平均場近似解析に関して - ”，2003 年情報論的学習理論ワークショップ

［10］ 遠藤友基，“パケット輸送に影響するネットワークトポロジの特性”，北陸先端科学技術大学院大学　知識科学研究科　2004 年 2 月.

［11］ 林幸雄，箕浦正人，松久保潤，“ネットワーク成長によるメール型ウィルスの再流行と重点的なハブの免疫化の効果”，情報処理学会論文誌：数理化モデルとその応用，Vol.44 No.SIG14(TOM 9)

［12］ Pastor-Storras, R. and Vespignani, A.: “Epidemic dynamics and epidemic states in complex networks”， Physical Review E, Vol.63,066117(2001).

［13］ Isolatov，P.L.Krapivsky，and A.Yuryev，“Duplication-divergence model of protein interaction network”，arXiv:q-bio.MN/0411052 vl，30 Nov 2004.

［14］ 和田維作，“よい乱数・悪い乱数”，
http://www001.upp.so-net.ne.jp/isaku/rand.html

［15］ Mersenne Twister　Home Page
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/mt.html