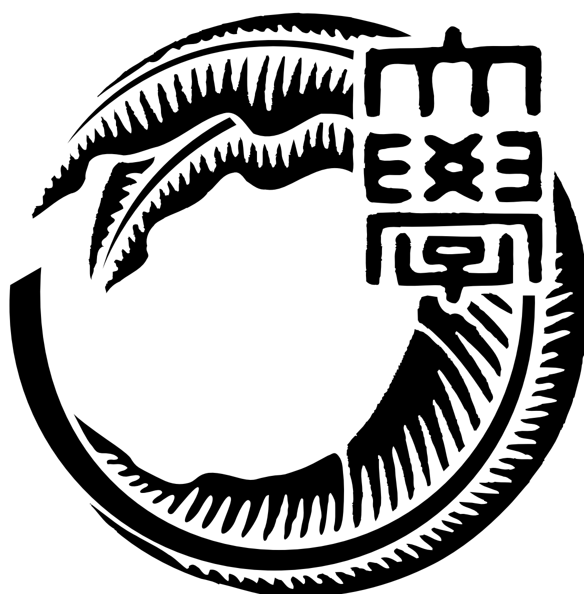
平成29年度 卒業論文

分散版Jungleデータベースの性能測定方法



琉球大学工学部情報工学科

145762E 仲松　栞

指導教員 河野 真治

目 次

第1章　はじめに **1**
- 1.1　研究背景 1
- 1.2　研究目的 2

第2章　分散版 jungle データベース **3**
- 2.1　Jungle データベースの構造 3
- 2.2　分散機構 3

第3章　評価実験 **7**
- 3.1　実験目的 7
- 3.2　実験概要 8
- 3.3　実験環境 9
- 3.4　TORQUE Resource Manager 9
- 3.5　分散フレームワーク Alice による分散環境の構築 11
- 3.6　データ書き込みプログラムの実装 17
- 3.7　時間計測プログラムの実装 18

第4章　性能評価 **23**
- 4.1　java 版 jungle の分散性能の評価 23
- 4.2　性能測定方法の評価 23

第5章　結論 **24**
- 5.1　まとめ 24
- 5.2　今後の課題 24

図 目 次

2.1 ツリー型のトポロジー 5
2.2 ring 型のトポロジー 5
2.3 メッシュ型のトポロジー 6

3.1 複数の jungle に書き込まれたデータが root の jungle へ到達する時間を計測する 8
3.2 TORQUE の構成 10
3.3 Alice による Jungle の木構造トポロジーの形成 14
3.4 トポロジーの形成 15

ソースコード目次

3.1 本実験で投入するジョブスクリプト 10
3.2 TopologyManager の起動方法 11
3.3 作成したトポロジーファイル 11
3.4 本実験で使用するトポロジーファイルを生成するプログラム 12
3.5 測定用プログラムの起動部分 17
3.6 Alice に実装した時間計測プログラムのプログラムの起動部分 18
3.7 Alice に実装した timestamp 部分 18
3.8 Alice に実装した timestamp 部分 20
3.9 Alice に実装した timestamp 部分 20
3.10 Alice に実装した timestamp 部分 21

第1章　はじめに

1.1　研究背景

天気予報やニュース、エンタメや生活に必要なありとあらゆる情報は、インターネット上で手軽に閲覧できるようになり、また、SNS により情報の発信も気軽にできるようになった。その利便性から、パソコンやスマートフォン、タブレット端末等のメディアがますます普及し、Web サービスの利用者は増大する一方で、サーバ側への負荷は増加している。Web サービスの負荷を減らすために、データベースには処理能力の高さ、すなわちスケーラビリティがますます求められてきている。データベースの処理能力を向上させるために、スケールアウトとスケールアップの方法が考えられる。スケールアップとは、ハードウェア的に高性能なマシンを用意することでシステムの処理能力を上げることを指す。スケールアウトとは、汎用的なマシンを複数用意し、処理を分散させることでシステムの処理能力を上げることを指す。単純に処理能力を上げる方法として、スケールアップは有効であるが、高性能のマシンには限界がある。コストはもちろん、そのマシン単体では処理できない程負荷がかかる可能性がある。それに対して、スケールアウトは、処理が重くなるたびに汎用的なマシンを順次追加していくことで性能を上げるため、ハードウェア的に高性能なマシンを用意せずにすみ、また柔軟な対応を取ることができる。よって、データベースの性能を向上させる方法として、このスケールアウトが求められている。本研究で扱うスケーラビリティとはスケールアウトのことを指す。

分散データシステムは、データの整合性 (一貫性)、常にアクセスが可能であること (可用性)、データを分散させやすいかどうか (分割耐性)、この 3 つを同時に保証するこ とは出来ない。これは CAP 定理と呼ばれる。一貫性と可用性を重視しているのが、現在最も使われているデータベースである Relational Database(RDB) である。そのため、データを分割し、複数のノードにデータを分散させることが難しく、結果スケールアウトが困難になってしまうという問題がある。分断耐性を必要とする場合は、NoSQL データベースを選択する。

　当研究室では、これらの問題を解決した、煩雑なデータ設計が必要ないスケーラビリ ティのあるデータベースを目指して、非破壊的木構造データベース Jungle[2] を開発している。Jungle は NoSQL を元に開発されているため、分断耐性を持っている。また、Jungle は、全体の整合性ではなく、木ごとに閉じた局所的な整合性を保証している。 整合性のある木同士をマージすることで新しい整合性のある木をす繰り出すことも 可能であるため、データの伝搬も容易である。Jungle は、これまでの開発によって木構造を格納する機能をもっている。

1.2 研究目的

Jungle は現在、Java と Haskell によりそれぞれの言語で開発されている。本研究で扱うのは Java 版である。これまでに行われた分散環境上での Jungle の性能を検証する実験 [5] では、Huskell で書かれた Jungle の方が、Java で書かれた Jungle よりも読み込み、書き込み共に高い性能差を出していた。

Huskell は、モダンな型システムを持ち, 型推論と 型安全により、信頼性に重きを置いてプログラミングを行う関数型言語である。対して、Java はコンパイラ型言語であり、構文に関しては C や C#の影響を受けており、プログラムの処理に関しては Huskell よりもパフォーマンスが高い言語であるといえる。処理速度においては Huskell よりも高いことを予想されていたのにもかかわらず、Java 版が Huskell 版よりも遅くなってしまった原因として、性能測定時に使用するテストプログラムのフロントエンドに Web サーバー Jetty が使用されていたことが考えられる。そこで、本研究では、新たに Web サーバーを取り除いた測定用プログラムを作成し、純粋な Java 版 Jungle の性能を測定する方法を提案する。

第2章　分散版jungleデータベース

Jungle は、スケーラビリティのあるデータベースの開発を目指して当研究室で開発されている分散データベースである。現在 Java と Haskell によりそれぞれの言語で開発されており、本研究で扱うのは Java 版である。本章では、分散データベース Jungle の構造と分散部分について触れる。

2.1　Jungle データベースの構造

Jungle は、当研究室で開発を行っている木構造の分散データベースで、Java を用いて実装されている。一般的なウェブサイトの構造は大体が木構造であるため、データ構造として木構造を採用している。

Jungle は名前付きの複数の木の集合からなり、木は複数のノードの集合でできている。ノードは自身の子のリストと属性名、属性値を持ち、データベースのレコードに相応する。通常のレコードと異なるのは、ノードに子供となる複数のノードが付くところである。

通常の RDB と異なり、Jungle は木構造をそのまま読み込むことができる。例えば、XML や Json で記述された構造を、データベースを設計することなく読み込むことが可能である。

また、この木を、そのままデータベースとして使用することも可能である。しかし、木の変更の手間は木の構造に依存する。特に非破壊木構造を採用している Jungle では、木構造の変更の手間は O(1) から O(n) となりえる。つまり、アプリケーションに合わせて木を設計しない限り、十分な性能を出すことはできない。逆に、正しい木の設計を行えば高速な処理が可能である。

Jungle はデータの変更を非破壊で行なっており、編集ごとのデータをバージョンとして TreeOperationLog に残している。Jungle の分散ノード間の通信は木の変更の TreeOperationLog を交換することによって、分散データベースを構成するよう設計されている。

2.2　分散機構

Jungle の分散設計は、Git や Mercurial といった分散バージョン管理システム (以下、分散管理システム) の機能を参考に作られている。分散管理システムとは、多人数によるソフトウェア開発において変更履歴を管理するシステムである。分散管理システムでは開発者それぞれがローカルにリポジトリのクローンを持ち、開発はこのリポジトリを通して行われる。ローカルのリポジトリは独立に存在し、サーバ上にあるリポジトリや他人のリ

ポジトリで行われた変更履歴を取り込みアップデートをかけることができる。また、逆にローカルリポジトリに開発者自身がかけたアップデートを他のリポジトリへと反映させることもできる。分散管理システムではどれかリポジトリが壊れたとしても、別のリポジトリからクローンを行うことができる。ネットワークに障害が発生しても、ローカルにある編集履歴をネットワーク復帰後に伝えることができる。このように、分散管理システムはデータ変更の履歴を自由に受け取ることが可能である。しかし、お互いが同じデータに対して編集を行なっている場合、変更履歴を自由に受け取ることができないことがある。これを衝突という。衝突は、分散管理システムを参考にしている Jungle においても起こる問題である。Jungle はリクエストが来た場合、現在持っているデータを返す。その為、データは最新のものであるかは保証されない。この場合、古いデータに編集が加えられ、それを更に最新のデータへ伝搬させる必要がある。このように他のリポジトリにより先にデータ編集が行われているとき、データの伝搬が素直にできない為、衝突が起きてしまう。この衝突に対し、分散管理システムでは Merge と呼ばれる作業を行う。Merge とは、相手のリポジトリのデータ編集履歴を受け取り、ローカルにあるリポジトリの編集と合わせる作業である。Jungle は、アプリケーションレベルでの Merge を実装することで、衝突を解決する。

Jungle の分散機構は、木構造、すなわちツリー型を想定したネットワークトポロジーを形成し、サーバー同士を接続することで通信を行なっている。ツリー型であれば、データの整合性をとる場合、一度トップまでデータを伝搬させることで行える。トップまたはトップまでの間にあるサーバーノードでデータを運搬中に衝突が発生したら Merge を行い、その結果を改めて伝搬すればよいからである (図 2.1)。

(図 2.1）の矢印の流れを以下に示す。

1. servernode 1, servernode 2 からきたデータが servernode 0 で衝突。
2. 衝突したデータの Merge が行われる。
3. Merge されたデータが servernode 1,servernode 2 へ伝搬
4. servernode1 から Merge されたデータが servernode 3、servernode 4 へ伝搬。全体でデータの整合性が取れる。

リング型 (図 2.2 ）やメッシュ型 (図 2.3 ）のトポロジーでは、データの編集結果を他のサーバーノードに流す際に、流したデータが自分自身に返ってくることでループが発生してしまう可能性がある。ツリー型であれば、閉路がない状態でサーバーノード同士を繋げることができる為、編集履歴の結果を他のサーバーノードに流すだけですみ、結果ループを防ぐことができる。

ネットワークトポロジーは、当研究室で開発している並列分散フレームワークである Alice が提供する、TopologyManager という機能を用いて構成されている。

また、データ分散の為には、どのデータをネットワークに流すのか決めなければならない。そこで、TreeOperationLog を使用する。前セクションでも述べたが、TreeOperationLog には、どの Node にどのような操作をしたのかという、データ編集の履歴情報が入っている。この TreeOperationLog を Alice を用いて他のサーバーノードに送り、データの編集をしてもらうことで、同じデータをもつことが可能になる。

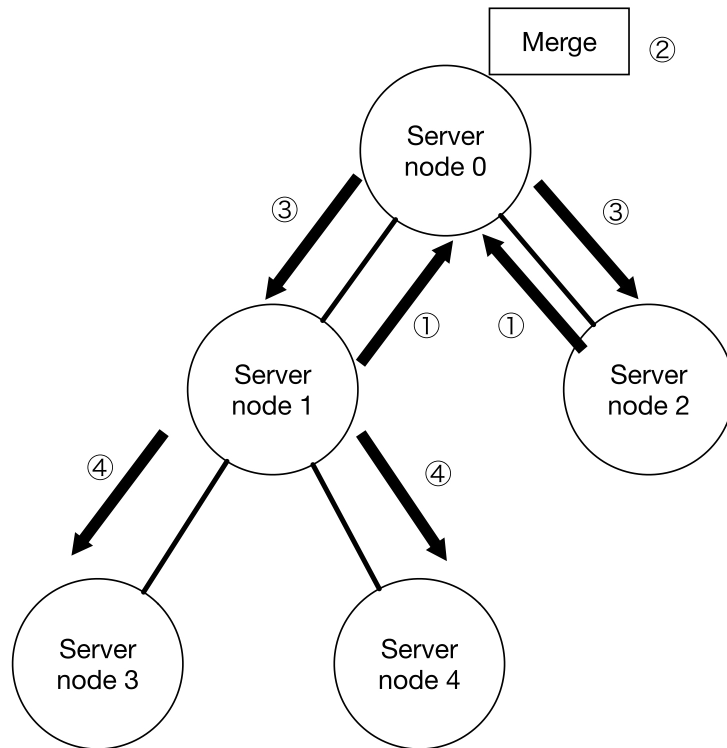


図 2.1: ツリー型のトポロジー

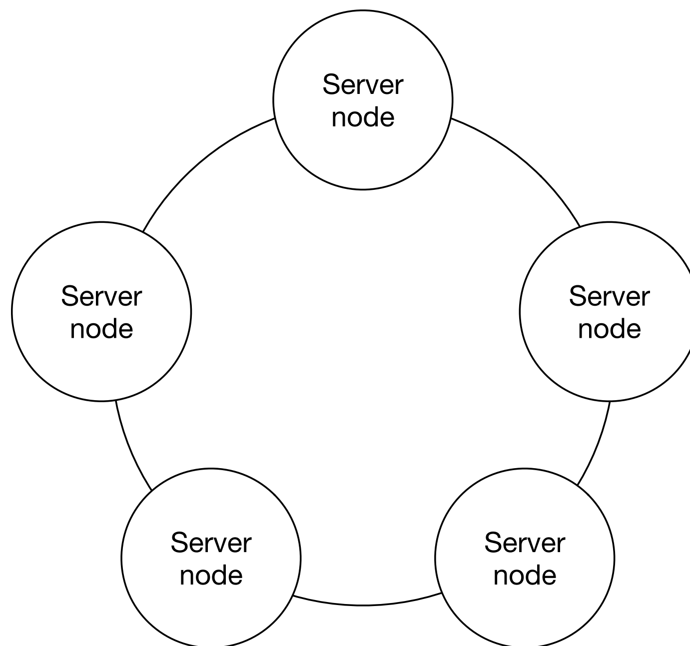


図 2.2: ring 型のトポロジー

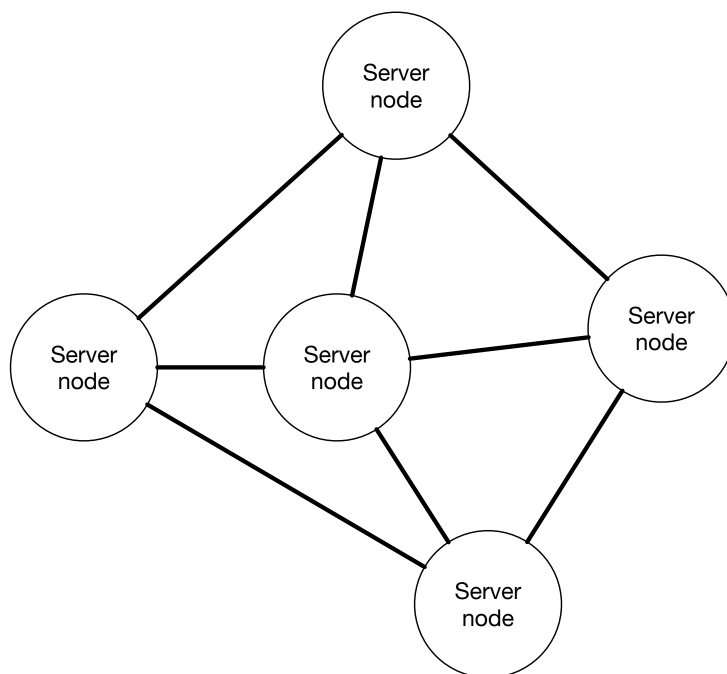


図 2.3: メッシュ型のトポロジー

第3章　評価実験

本研究は、Jungle の分散環境上での性能を正しく評価するための実験を行う。本章では実験の概要について述べる。まず、本研究の目的について述べ、次に、分散フレームワーク Alice による、本研究の分散機構を構成する方法について述べる。次に、木構造上に立ち上げた Jungle へ投入するタスクを制御するジョブスケジューラー、TORQUE について述べる。最後に、本実験の測定用プログラムについて述べる。

3.1　実験目的

Jungle は現在、Java で実装されたものと、Huskell で実装されたものがある。Java 版は、処理速度が早く、よりスケーラビリティの高いデータベースの実装を目的に開発された。対して Huskell 版は、モダンな型システムと、型推論と型安全という特徴を生かし、信頼性の高データベースの実装を目的に開発された。そして、これまでの研究で、Java 版と Haskell 版の Jungle の分散性能を測定する実験が行われている。分散性能測定実験では、それぞれ Jetty,Wrap という web サーバーをフロントエンドに用いた Web 掲示板サービスを使用している。Java 版と Huskell 版の Web 掲示板サービスをブレードサーバー上で実行される。計測方法は、掲示板に対して読み込みと書き込みを行い、ネットワークを介して weighhttp で負荷をかける。weighthttp の設定は、 1 スレッドあたり 100 並列のリクエストを、10 スレッド分投入し、合計 100 万のリクエストを処理させる。Java と Haskell の測定結果が表 3.1 のようになった。

測定	Haskell	Java
読み込み	16.31 s	53.13 s
書き込み	20.17 s	76.4 s

表 3.1: Haskell と Java の比較

Huskell 版は、 Java 版と比較して、読み込みで 3,25 倍、書き込みで 3.78 倍 の性能差がでている結果となってしまった。処理速度においては Huskell よりも高いことを予想されていたのにもかかわらず、Java 版が Huskell 版よりも遅くなってしまった原因は、測定時の Web 掲示板サービスのフロントエンドに、どちらも Web サーバーを用いているということが考えられる。しかも、その際は言語の問題から、異なる種類の Web サーバーを使用している。これでは、この性能結果が、異なる言語で実装された Jungle の性能差に

よるものなのか、Web サーバーの性能差によるものなのかがわからない。そこで、本研究では Java 版の Jungle において、Web サーバーを取り除いた、純粋な Jungle の分散性能を測定するプログラムを実装した。

3.2 実験概要

Jungle の分散性能を測定するにあたり、複数台の Jungle を通信させ、Jungle から Jungle に対する書き込みにかかる時間を計測する。複数台の Jungle を分散させる為に、学内共用の仮想マシンを 32 台使用した。分散した Jungle 同士の通信部分には、当研究室で開発している分散フレームワーク Alice の機能である TopologyManager を使用する。TopologyManager の起動には、仮想マシン 32 台のうちの 1 台を使用する。学科の仮想マシン 31 台上でそれぞれ 1 台ずつ Jungle を立ち上げ、ツリー型のトポロジーを構成する。そのうち 16 台の Jungle に対して 100 回ずつデータを書き込む。子ノードの Jungle は、他の Jungle で書き込まれたデータを自身に書かれたデータと merge していく。全ての子ノードの Jungle で merge されたデータは、最終的に親ノードの Jungle のデータへ merge されていく。本実験では、複数の子ノードに Jungle に書き込まれたデータが最終的にルートノードの Jungle のデータへ merge され、書き込まれた時間を計測し、平均を取る。図 3.1 は本実験を図で表したものであり、31 台中 16 台の Jungle から書き込まれた データがルートノードの Jungle へ書き込まれる、一回あたりの時間を計測する。

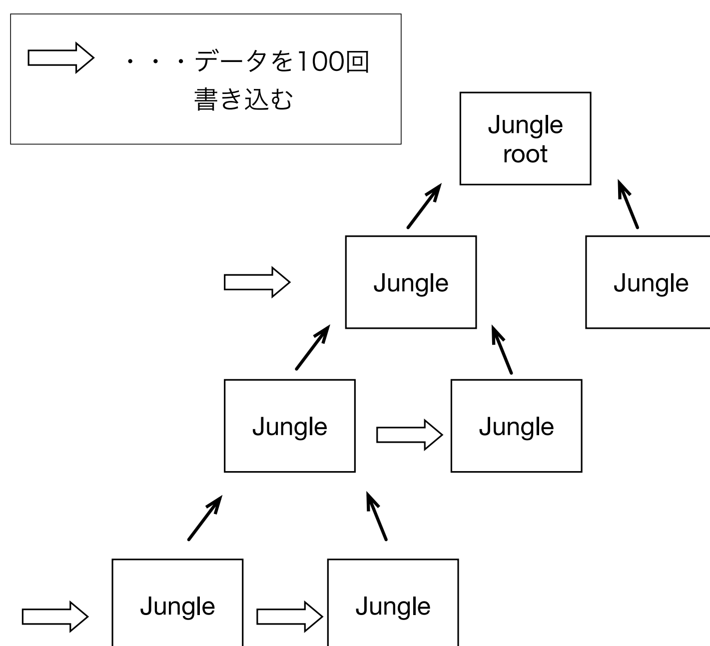


図 3.1: 複数の jungle に書き込まれたデータが root の jungle へ到達する時間を計測する

3.3 実験環境

学科の KVM 上の仮想マシンによる仮想クラスタ環境を用いて実験を行った。分散環境上での実験を行うにあたり、他の利用者とリソースが競合しないよう、TORQUE ジョブスケジューラーを利用している。KVM と仮想マシンの性能はそれぞれ表 3.2、表 3.3 である。

マシン台数	Haskell
CPU	16.31 s
物理コア数	20.17 s
論理コア数	
CPU キャッシュ	
Memory	

表 3.2: KVM の詳細

マシン台数	Haskell
CPU	16.31 s
物理コア数	20.17 s
仮想コア数	
CPU キャッシュ	
Memory	

表 3.3: 仮想クラスタの詳細

3.4 TORQUE Resource Manager

分散環境上での Jungle の性能を測定するにあたり、VM31 台に Jungle を起動させた後、16 台の Jungle に対し、データを書き込むプログラムを動作させる。プログラムを起動する順番やタイミングは、TORQUE Resource Manager というジョブスケジューラーによって管理する。

TORQUE Resource Manager は、ジョブを管理・投下・実行する 3 つのデーモンで構成されており、ジョブの管理・投下を担うデーモンが稼働しているヘッダーノードから、ジョブの実行を担うデーモンが稼働している計算ノードへジョブが投下される（図 3.2 ）。

ユーザーはジョブを記述したシェルスクリプトを用意し、スケジューラーに投入する。その際に、利用したいマシン数や CPU コア数を指定する。TORQUE は、ジョブに必要なマシンが揃い次第、受け取ったジョブを実行する。

今回作成した、ジョブに投入するためのシェルスクリプトを以下 (ソースコード 3.1) に示す。

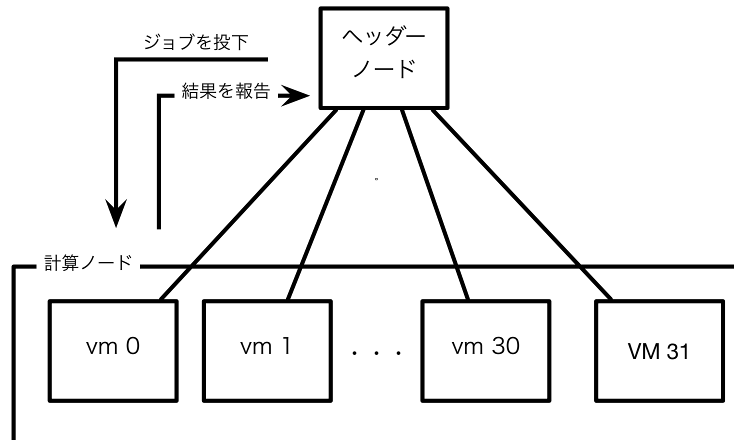


図 3.2: TORQUE の構成

ソースコード 3.1: 本実験で投入するジョブスクリプト

```
#!/bin/sh
#PBS -q jungle
#PBS -N LogUpdateTest
#PBS -l nodes=16,walltime=00:08:00

cd /mnt/data/jungle_workspace/Log
/usr/bin/perl /mnt/data/jungle_workspace/scripts/LogupdateTest.pl
```

6 行目で指定されたディレクトリに移動し、7 行目ではそのディレクトリで、指定した別の階層にある perl スクリプトを実行している。

3.5 分散フレームワーク Alice による分散環境の構築

分散させた Jungle の通信部分を担うのが、当研究室で開発している並列分散フレームワーク Alice[1] である。Alice は、ネットワーク上の複数のサーバーノードにトポロジーを形成させ、通信する機能を提供する。今回扱うサーバーノードに学科の仮想マシン (VM) を用いる。本研究では、分散環境上での Jungle の性能を確認する為、VM32 台分のサーバーノードを用意し、それぞれで 1 台ずつ Jungle 起動することで、分散させる。

Alice には、ネットワークのトポロジーを構成する TopologyManager[2] という機能が備わっている。TopologyManager は以下のソースコード 3.2 のように起動する。

ソースコード 3.2: TopologyManager の起動方法

```
% java -cp ../../build/libs/logupdateTest-1.1.jar alice.topology.manager.
    TopologyManager -conf ../../scripts/tree.dot -p 10000
```

-p オプションはトポロジーマネージャーが開くポートの番号、-conf オプションには トポロジーファイルである tree.dot のパスを渡している。トポロジーファイルとは、どのようにトポロジーノードをつなげるかを記述したファイルである。Alice の TopologyManager を使用する際は、どのようなトポロジーを形成したいかを決め、あらかじめトポロジーファイルを作成する必要がある。今回、31 台のサーバーノードでツリートポロジーを形成する dot ファイルを以下のソースコード 3.3 のように記述した。

ソースコード 3.3: 作成したトポロジーファイル

```
digraph test {
    node0 -> node1 [label="child1"]
    node0 -> node2 [label="child2"]
    node1 -> node0 [label="parent"]
    node1 -> node3 [label="child1"]
    node1 -> node4 [label="child2"]
    node2 -> node0 [label="parent"]
    node2 -> node5 [label="child1"]
    node2 -> node6 [label="child2"]
    node3 -> node1 [label="parent"]
    node3 -> node7 [label="child1"]
    node3 -> node8 [label="child2"]
    node4 -> node1 [label="parent"]
    node4 -> node9 [label="child1"]
    node4 -> node10 [label="child2"]
    node5 -> node2 [label="parent"]
    node5 -> node11 [label="child1"]
    node5 -> node12 [label="child2"]
    node6 -> node2 [label="parent"]
    node6 -> node13 [label="child1"]
```

```
21      node6 -> node14 [label="child2"]
22      node7 -> node3 [label="parent"]
23      node8 -> node3 [label="parent"]
24      node9 -> node4 [label="parent"]
25      node10 -> node4 [label="parent"]
26      node11 -> node5 [label="parent"]
27      node12 -> node5 [label="parent"]
28      node13 -> node6 [label="parent"]
29      node14 -> node6 [label="parent"]
30  }
```

また、31 台のサーバーノードで形成するトポロジーファイルを自動で生成するプログラムを作成した。以下のソースコード 3.4 に示す。

ソースコード 3.4: 本実験で使用するトポロジーファイルを生成するプログラム

```
1   def create_nodes(node_num)
2     (0..node_num - 1).map { |i|
3       i = "node" + i.to_s
4     }
5   end
6
7   def print_dot(connections)
8     puts "digraph test {"
9     connections.each { |connection|
10      print "\t"
11      print connection[0]
12      print " -> "
13      print connection[1]
14      print ' [label="' + connection[2] + '"]'
15      puts
16    }
17    puts "}"
18  end
19
20  node_num = ARGV[0].to_i
21  nodes = create_nodes(node_num)
22  connections = Array.new
23  nodes.each_with_index { |node, i|
24    parent = (i - 1) / 2;
25    child1 = 2 * i + 1;
26    child2 = 2 * i + 2;
27    if parent >= 0 then
28      connections << [nodes[i], nodes[parent], "parent"]
```

```
29    end
30    if child1 < node_num then
31      connections << [nodes[i], nodes[child1], "child1"]
32    end
33    if child2 < node_num then
34      connections << [nodes[i], nodes[child2], "child2"]
35    end
36  }
37  print_dot(connections)
```

TopologyManager は、参加表明をしたサーバーノード (以下 TopologyNode) を、トポロジーファイルの内容に従ってトポロジーを構成する。TopologyManager への参加表明は、TopologyNode起動時に、TopologyManager の IP アドレスとポート番号を指定すれば良い。

TopologyNode は TopologyManager に、誰に接続を行えばよいかを尋ねる。TopologyManager は尋ねてきた TopologyNode に順番に、接続先の TopologyNode の IP アドレス、ポート番号、接続名を送り、受け取った TopologyNode はそれらに従って接続する。この時、TopologyManager 自身は VM0 を用いて立ち上げる。よって、TopologyManager は Jungle をのせた VM1 から VM32、計 VM31 台分のサーバーノードを、木構造を形成するように采配する（図 3.3 ）。

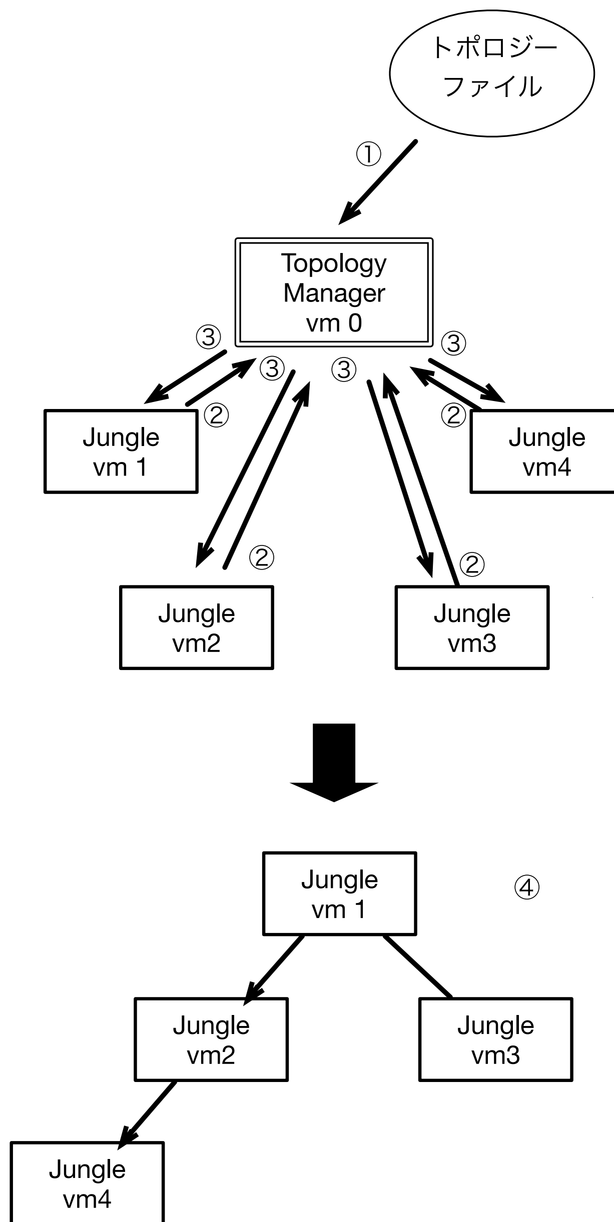


図 3.3: Alice による Jungle の木構造トポロジーの形成

図 3.3 の矢印の流れを以下に示す。

1. TopologyManager がトポロジーファイルを読み込む。
2. TopologyNode が TopologyManager に接続先を尋ねる。
3. 接続先の TopologyNode の IP アドレス、ポート番号、接続名を送り返す。
4. 受け取った接続先の情報を元に、トポロジーを形成する。

各ノードは自身を示す nodeName[3] を持ち、この nodeName を指定することで他ノードとの通信を行う。nodeName は自身のサーバーノードがデータを受け取る際に指定する必要がある。

たとえば、servernode0,servernode1,servernode ２により、図3.4のように木構造が構成されたとする。

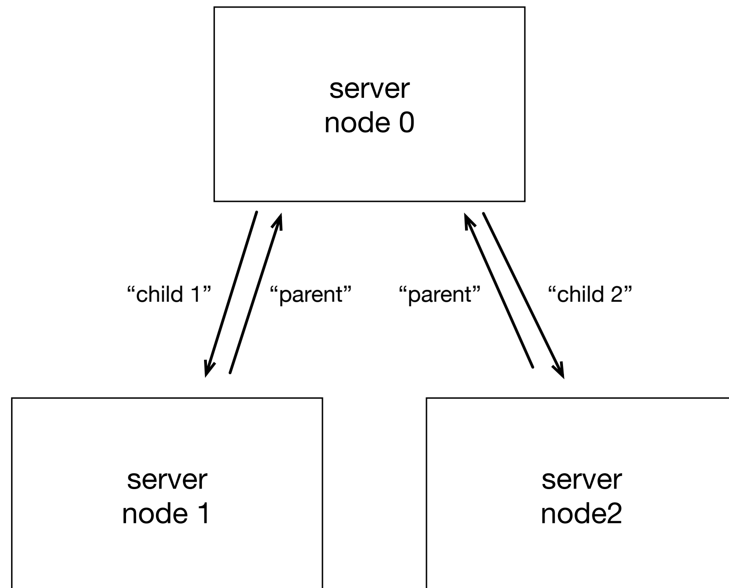


図 3.4: トポロジーの形成

この時、servernode0はservernode １、servernode2に対して親にあたる。逆に、servernode1,servernode ２はservernode0に対して子にあたる。よって、図3.4に矢印の隣にかかれている文字列”parent”,”child 1”,”child 2”のようにキーを指定している。servernode0からservernode1へデータを送りたい場合、”child 1”という nodeName を追加すればいい。このように、データアクセスしたいサーバーノードの nodeName を追加することで、そのサーバノードの DataSegment へデータアクセスすることができる。他のサーバーノードの DataSegment へデータアクセスする際には、アクセス先のサーバーノードの nodeName を追加すればいい。

トポロジー構成後、Jungle 間の通信でのデータ形式には TreeOperationLog を利用する。TreeOperationLog は、Jungle によるノードの編集の履歴などの情報が入っている。TreeOperationLog は、Alice の DataSegment でも扱えるようシリアライズ化されたデータである。各サーバーノードには、データを受け取る部分である DataSegment が備わっている。よって、Alice によって構成されたネットワークトポロジーのサーバノード間でのデータのアクセスが可能になっている。TreeOperationLog を Alice によって他の Jungle へ送る。送信先の Jungle では、送られてきた TreeOperationLog を参照して送信元の Jungle と同じノード編集を行う。こうして、Jungle 間でのデータの同期を可能にしている。Alice のトポロジー形成と他のサーバのデータへアクセスする機構を用いるためには、Alice が提供するプログラミングスタイルに沿わなければならない。それは DataSegment と CodeSegment である。よって、Jungle のログの実態である TreeOperationLog は、DataSegment で扱えるようシリアライズされている。Alice はタスクを行う CodeSegment と、CodeSegment で使用

するデータを扱う DataSegment によってプログラムを行うスタイルを取る。CodeSegment は DataSegment が必要なデータを受け取り次第、タスクを行う。DataSegment がデータを受け取る為には、その DataSegment を示すキーが必要である。

3.6 データ書き込みプログラムの実装

これまで、本実験の概要、測定環境について説明し、次に TORQUE による Jungle へのプログラム投下方法と、Alice による分散通信部分の構築方法について説明した。ここでは、本実験の手順と合わせて、今回実装した部分である、Jungle にデータを書き込むための機能について解説する。本実験において、木構造を形成した 32 台のうち、16 台の Jungle へデータを 100 回書き込むプログラムを作成した。以下のソースコード 3.5 は、実装したテストプログラムの起動部分である。

ソースコード 3.5: 測定用プログラムの起動部分

```
1  mysystem("ssh $nodes[0] \"cd $logFile/$logNum;java -cp ../../build/libs/l
      ogupdateTest-1.1.jar alice.topology.manager.TopologyManager -conf
      ../../scripts/tree.dot -p 10000 --showTime --noKeepAlive >  $logNum\"
      ",1);
2  $logNum++;
3  sleep 10;
4  for my $i (1..($#nodes-1)) {
5    mkdir "$logFile/$logNum";
6    mysystem("ssh $nodes[$i] \"cd $logFile/$logNum;java -jar ../../build/li
      bs/logupdateTest-1.1.jar -host $nodes[0] -p 10003 -port 10000  --noKee
      pAlive > $logNum\" & ");
7    $logNum++;
8  }
9  mkdir "$logFile/$logNum";
10 mysystem("ssh $nodes[$#nodes] \"cd $logFile/$logNum;java -jar ../../buil
      d/libs/logupdateTest-1.1.jar -host $nodes[0] -p 10003 -port 10000  -wr
      ite -count 10 --noKeepAlive > $logNum\" &" );
11 for (wait) wait;
```

1 行目で本実験のネットワークトポロジーを形成するため topokogymanager の起動を行なっている。$nodes は VM0 VM31 台の、合計 32 台の仮想マシンを表し、TopologyManager は VM0 に起動する。

-p オプションは TopologyManager が開くポートの番号、-conf オプションには dot ファイルのパスを渡している。ポート番号は Alice のより記述された並列分散プログラムの起動時に渡す必要がある。

dot ファイルには、トポロジーをどのように構成するかが書かれている。dot ファイルを読み込んだ Alice の TopologyManager に対して、サーバーノードは誰に接続を行えばよいかを尋ねる。TopologyManager は尋ねてきたサーバーノードに対してノード番号を割り振り、dot ファイルに記述している通りにサーバーノードが接続を行うように指示をだす。

-showTime オプションは、今回 Alice に実装した機能である。–showTime オプションをつけることで、出力される結果に、子ノードの Jungle からの書き込みが root ノードの Jungle へ到達し、書き込みが終了したときの時間が表示されるようになる。

4 行目では、Jungle を起動している。このとき、-host で TopologyManager の IP アドレスを渡し、-port で TopologyManager のポート番号をしている。TopologyManager の IP アドレスとポート番号を渡すことで、TopologyManager へ参加表明をしている。

10 行目では、4 行目と同様 Jungle を起動しているが、今回実装した-write オプションと-count オプションをつけている。

-write オプションは、Jungle にデータを書き込む機能をつけることができる。

-count オプションは、何回データを書き込むかを指定することができる。隣に引数をつけることで、回数を設定できる。本実験では、16 台の Jungle で、100 回データを書き込むよう設定する。すなわち、TopologyManager に 1 台、write モードで立ち上げる Jungle に 16 台使た後、残りの 15 台はそのまま Jungle を起動させている。

このスクリプトは、TORQUE によって各サーバーノードへ投入され、実行される。

3.7 時間計測プログラムの実装

前節ではツリートポロジーを形成した子ノードの Jungle へデータを書き込む部分、及び実験手順について解説した。複数の子ノードにデータをそれぞれ書き込み、最終的に root ノードへデータを書き込んでいく時間を計測する為の機能を、新たに Alice に実装した。ここでは Alice に実装した時間計測プログラムについて解説する。

以下のソースコード 3.6 は、実装したテストプログラムの起動部分である。

ソースコード 3.6: Alice に実装した時間計測プログラムのプログラムの起動部分

```
1 ystem("cd $jungleDir; ruby scripts/tree.rb $nodeNum > scripts/tree.dot");
2 mkdir "$logFile";
3 mysystem("ssh $nodes[0] \"cd $logFile;java -cp ../../build/libs/logupdate
     Test-1.1.jar alice.topology.manager.TopologyManager -conf ../../script
     s/tree.dot -p 10000 --showTime --noKeepAlive > $logNum\" ",1);
```

ソースコード 3.7

ソースコード 3.7: Alice に実装した timestamp 部分

```
1  package alice.topology.manager;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.io.FileReader;
6  import java.util.ArrayList;
7  import java.util.HashMap;
8  import java.util.LinkedList;
9
10 import org.apache.log4j.Logger;
11
```

```
12 import alice.codesegment.CodeSegment;
13 import alice.topology.HostMessage;
14 import alice.topology.fix.ReceiveDisconnectMessage;
15
16 import com.alexmerz.graphviz.ParseException;
17 import com.alexmerz.graphviz.Parser;
18 import com.alexmerz.graphviz.objects.Edge;
19 import com.alexmerz.graphviz.objects.Graph;
20 import com.alexmerz.graphviz.objects.Node;
21
22 public class StartTopologyManager extends CodeSegment {
23
24     TopologyManagerConfig conf;
25     Logger logger = Logger.getLogger(StartTopologyManager.class);
26
27     public StartTopologyManager(TopologyManagerConfig conf) {
28         this.conf = conf;
29     }
30
31     Overridepublic void run() new
    CheckComingHost();ods.put("absCookieTable", new HashMap<String,
    String>());ods.put("config", conf );if (!conf.dynamic)
    LinkedList<String> nodeNames = new
    LinkedList<String>();HashMap<String, LinkedList<NodeInfo>> topology =
    new HashMap<String,LinkedList<NodeInfo>>();int nodeNum = 0;try
    FileReader reader = new FileReader(new File(conf.confFilePath));Parser
    parser = new Parser();parser.parse(reader);ArrayList<Graph> graphs =
    parser.getGraphs();for (Graph graph :  graphs) ArrayList<Node> nodes =
    graph.getNodes(false);nodeNum = nodes.size();for (Node node :  nodes)
    String nodeName =
    node.getId().getId();nodeNames.add(nodeName);topology.put(nodeName,
    new LinkedList<NodeInfo>());ArrayList<Edge> edges =
    graph.getEdges();HashMap<String, NodeInfo> hash = new HashMap<String,
    NodeInfo>();for (Edge edge :  edges) String connection =
    edge.getAttribute("label");String source =
    edge.getSource().getNode().getId().getId();String target =
    edge.getTarget().getNode().getId().getId();LinkedList<NodeInfo>
    sources = topology.get(target);NodeInfo nodeInfo = new
    NodeInfo(source, connection);sources.add(nodeInfo);hash.put(source +
    "," + target, nodeInfo);for (Edge edge :  edges) String connection =
    edge.getAttribute("label");String source =
    edge.getSource().getNode().getId().getId();String target =
    edge.getTarget().getNode().getId().getId();NodeInfo nodeInfo =
    hash.get(target + "," + source);if (nodeInfo != null)
    nodeInfo.reverseName = connection; catch (FileNotFoundException e)
    logger.error("File not found:  " +
    conf.confFilePath);e.printStackTrace(); catch (ParseException e)
    logger.error("File format error:  " +
    conf.confFilePath);e.printStackTrace();// for recode topology
    information// cookie Listods.put("running",
    false);ods.put("resultParse", topology);ods.put("nodeNames",
    nodeNames);new IncomingHosts();ConfigWaiter cs3 = new
```

```
ConfigWaiter(nodeNum);cs3.done.setKey("local", "done"); else
ods.put("running", true);HashMap<String, HostMessage> nameTable = new
HashMap<String, HostMessage>();if (conf.type == TopologyType.Tree) int
cominghostCount = 0;ParentManager manager = new
ParentManager(conf.hasChild);ods.put("parentManager",
manager);ods.put("nameTable", nameTable);ods.put("hostCount",
cominghostCount);new ComingServiceHosts();new
ReceiveDisconnectMessage();ods.put("topology", new HashMap<String,
LinkedList<HostMessage>>());ods.put("createdList", new
LinkedList<String>());new CreateHash();TopologyFinish cs2 = new
TopologyFinish();cs2.finish.setKey("local",
"finish");cs2.config.setKey("config");cs2.startTime.setKey("startTime");
```

ソースコード 3.8

ソースコード 3.8: Alice に実装した timestamp 部分

```
1  package alice.topology.manager;
2
3  import alice.codesegment.CodeSegment;
4  import alice.datasegment.CommandType;
5  import alice.datasegment.Receiver;
6
7  public class TopologyFinish extends CodeSegment {
8      public Receiver finish = ids.create(CommandType.TAKE);
9      public Receiver config = ids.create(CommandType.PEEK);
10     public Receiver startTime = ids.create(CommandType.TAKE);
11     Overridepublic void run() TopologyManagerConfig conf =
   config.asClass(TopologyManagerConfig.class);long start =
   startTime.asClass(Long.class);if (conf.showTime)
   System.out.println("TopologymanagerTime = "+
   (System.currentTimeMillis()-start));System.exit(0);
```

ソースコード 3.7 は TopologyManager を開始するコードであり、ソースコード 3.8 は TopologyManager の終了部分のコードである。本実験において、子ノードの Jungle から root ノードの Jungle へデータの merge が終了する時間を計るために、TopologyManager の開始時、終了時の時間を取得している。ソースコード 3.8 の最後では、取得した終了時の時間から、開始時の時間を差し引いた時間を出力している。前の説で解説した、データ書き込みプログラム () が投入されると、TopologyManager が起動し、Jungle へのデータの書き込みが始まる。そして root ノードの Jungle へデータが書き込み終わると共に TopologyManager が終了するので、TopologyManager の終了時から開始時を差し引いた時間が、今回の測定範囲となる。

ソースコード 3.8 の最後にある、currentTimeMillis() は TopologyManager が終了した現時点の時間である。TopologyManager の開始時の時間は、以下のソースコード 3.9 で示す。

ソースコード 3.9: Alice に実装した timestamp 部分

```
1  package alice.topology.manager;
```

```
2
3 import org.msgpack.type.ValueFactory;
4
5 import alice.codesegment.CodeSegment;
6 import alice.datasegment.CommandType;
7 import alice.datasegment.Receiver;
8
9 public class ConfigWaiter extends CodeSegment {
10
11     public Receiver done = ids.create(CommandType.TAKE);
12     public int count;
13
14     public ConfigWaiter(int nodeNum) {
15         this.count = nodeNum;
16     }
17
18     Overridepublic void run() count--;if (count == 0) ods.put("local",
    "start",
    ValueFactory.createNilValue());ods.put("startTime",System.currentTimeMillis());o
    true);return;ConfigWaiter cs3 = new
    ConfigWaiter(count);cs3.done.setKey("local", "done");
```

ソースコード 3.9 では、Jungle の起動時につけた-count オプションの引数を取得している。ここでは、DataSegment である startTime に、現在の時間を入れている。この時間が、TopologyManager の開始時のデータとなる。

ソースコード 3.10: Alice に実装した timestamp 部分

```
1 package alice.topology.manager;
2
3 import alice.daemon.Config;
4
5 public class TopologyManagerConfig extends Config {
6
7     public boolean showTime = false;
8     public String confFilePath;
9     public boolean dynamic = false;
10    public TopologyType type = TopologyType.Tree;
11    public int hasChild = 2;
12
13    public TopologyManagerConfig(String[] args) {
14        super(args);
15        for (int i = 0; i < args.length; i++) {
16            if ("-conf".equals(args[i])) {
17                confFilePath = args[++i];
```

```
18             } else if ("--Topology".equals(args[i])) {
19                 String typeName = args[++i];
20                 if ("tree".equals(typeName)) {
21                     type = TopologyType.Tree;
22                 }
23             } else if ("--Child".equals(args[i])) {
24                 hasChild = Integer.parseInt(args[++i]);
25             } else if ("--showTime".equals(args[i])) {
26                 showTime = true;
27             }
28         }
29 
30         if (confFilePath == null)
31             dynamic = true;
32     }
33 
34 }
```

ソースコード 3.10 では、データ書き込みプログラムを投入時、TopologyManager の起動部分でつけた -showTime オプションの有無を判断している。オプションが存在していれば、実行結果に root の Jungle に書き込まれた時間が表示される。

第4章　性能評価

4.1　java版jungleの分散性能の評価

4.2　性能測定方法の評価

第5章　結論

5.1　まとめ

本研究では、Jungle の純粋な性能を測定するためのプログラムを Jungle,Alice に実装した。また、それらの機能を使用し、実際に Jungle の性能評価を行なった。

Jungle への書き込みを行う機能である -write オプションと、書き込みの回数を指定できる -count オプションの実装を行なった。

ツリートポロジーを構成した Jungle の分散環境上で、子ノードの Jungle に書き込まれたデータが、root ノードの Jungle に到達し、書き込みが終了するまでの時間を表示する、–showtime オプションの実装を行なった。

実際に、Jungle の性能を測定するため、分散環境を構築した上で、Jungle の書き込み時間を測定した。

今回 Jungle の分散性能の評価を行い、() がわかった。

5.2　今後の課題

今後の課題として、() を行う。また、Huskell 版 Jungle の、Web サーバーを介しない測定方法を考察したい。

参考文献

[1] 杉本 優 : 分散フレームワーク Alice 上の Meta Computation と応用,

[2] 大城 信康 : 分散 Database Jungle に関する研究,

[3] 金川 竜己 : 非破壊的木構造データベース Jungle とその評価

[4] 大城 信康, 杉本 優, 河野真治 : Data Segment の分散データベースへの応用, 日本ソフトウェア科学会 (2013).

[5] 當間 大千 : 関数型言語 Haskell による並列データベースの実装,

[6] 杉本 優 : 分散ネットフレームワーク Alice による例題の作成

謝辞

本研究を行うにあたり, 日頃より多くの助言, ご指導いただきました河野真治准教授に心より感謝申し上げます。
また, 本実験の測定にあたり, Alice のプログラミングについてご指導くださった照屋のぞみ先輩、伊波 立樹先輩, torque の環境構築に協力してくださった前城健太郎先輩, また、たくさんの温かい励ましをくださった照屋のぞみ先輩、新里幸恵先輩、赤嶺研の泉川真里奈さん、長田研の豊美玲さん、並列信頼研究室の全てのメンバーに深く感謝いたします。最後に、物心両面で支えてくれた両親に深く感謝いたします。