Code Segment と Data Segment によるプログラミング手法

河野真治[†]　杉本優[†]

データを Data Segment、タスクを Code Segment という単位に分割して記述する分散ネットフレームワーク Alice を作成した。Alice によるプログラミング例を示すと共に、本研究室で従来使ってきた Federated Linda との比較も示し、Java による実装について考察する。

Programming with Code Segment and Data Segment

Shinji Kono[†] and Yu Sugimoto[†]

We have developed an distributed programming frame Alice, which uses Data Segment and Code Segment as programming units. We show programming examples and comparisons with our old framework, Federated Linda. We show some consideration about our Java implementation.

1. 分散ネットフレームワーク Alice

Alice[8] は、本研究室で開発を行なっている並列タスク管理フレームワーク である。Cell 用の Open CL に似た Task 管理フレームワーク Cerium[4),10)] と、Linda[1)] を相互接続した分散フレームワークである Federated Linda[6)] の開発を通して得られた知見を生かされている。

Cerium では、Task を小さく分割して並列実行し、データ転送はパイプライン実行により隠される。Task には依存関係があり、その記述は煩雑になるが、実際にはデータの依存関係がそのまま Task の依存関係になることが多い。繰り返し使われるデータ構造の管理が重要であり、実行時にわかるデータ構造間の依存関係が Task を複雑にしている。

Federated Linda では、Linda サーバ内部に Meta Engine と呼ばれる Linda のタプル (データ構造) をやり取りする部分を作成した[7)]。Meta Engine では、タプルのやり取りによって起動する call back を使うが、call back による記述が分散してしまい、可読性を落としてしまう。また、複数のタプルの待ち合わせが重要だが、その待ち合わせは single threaded な Meta Engine 内部の状態に依存する。

これらが示しているのは、並列分散実行はコードの並列実行だけでなく、データの単位が重要だということである。そこで、 Alice は Data Segment と Code Segment という単位でデータと処理を細かく分割し、それぞれの依存関係を記述して分散プログラムを作成する。Code Segment は Continuation based C の実行単位[3),9)] であり、その双対が Data Segment である。

Data Segment は Code Segment と分離されたデータ構造であり、オブジェクトではない。オブジェクト指向プログラミングが状態を複雑に持ち、並列実行や分散実行に向かないことは徐々に理解されてきている。一方で、状態自体は有限状態遷移機械 (Finite State Machine/FSM) で記述するのが自然である。Code Segment は状態遷移記述そのものであり、その状態遷移は Data Segment の到着によってトリガーされる。

カプセル化されたデータをプロセスがやり取りするのは、DFD(Data Flow Diagram) の古典的な手法であり、それ自体は新しくはない。むしろ、メインフレーム上でのソフトウェア開発に良く使われてきた手法である。Alice では、それを再実装する。

Alice は Code Segment と Data Segment を Java と Message Pack で実装したフレームワークである。トポロジーマネージャーを持ち、Blade 上での分散プログラムの実験を容易に行うことができる。また、SEDA Architecture[5)] を採用しており、マルチコア上でのスループットの向上を期待している。

† 琉球大学
University of the Ryukyu

本論文では、Code Segment と Data Segment の Alice の API と、その設計方針を示し、それによって実装された水族館プログラムを示す。また、これまでの Federated Linda との性能評価も行う。

2. Data Segment API

Data Segment は数値や文字列などのデータを構造体的に保持するが、Data Segment の相互参照が問題になる。Alice では Data Segment をデータベースとして扱い、Data Segment は必ずキーを持つ。つまり、Data Segment を Key Value Store として考えることができる。通常のデータベースでは隠れているが、Key 毎のキューがあり、Key 毎に順に実行される。key 毎の追加と取得は、Linda に準じた設計になっている。

Data Segment を管理するのが Data Segment Manager である。ノード毎に local DS manager と remote DS manager がある。local manager は、ノードに固有の key Value Store と考えることができる。したがって Key はノード内部で unique な文字列である。

remote DS manager は他のノードの local DS manager の proxy である。Alice のトポロジーマネージャーが remote DS manager を自動的に構成する。つまり、remote DS manager は複数あって、それぞれ対応するノードが異なる。

Key Value Store へのアクセスはキューによって、ノード内部で逐次化される。それ以外は、すべて Java の Thread pool により並列実行される。Code Segment が実行される時には、Data Segment はすべて手元に揃っているので、Blocking が起きることはない。逆に、Blocking が必要な場合は、Code Segment を分割する必要がある。

以下が用意されている Data Segment API である。これらを用いてデータの送受信を行う。

- `void put(String key, Value val)`
- `void update(String key, Value val)`
- `void peek(Receiver receiver, String key`
- `void take(Receiver receiver, String key)`

2.1 put

`put` はデータを追加するための API である。Key Value Store のキューに追加される。(図 1)

2.2 update

`update` はデータを置き換えるための API である。キューの先頭を置き換える特急メッセージのように動作する。(図 2)

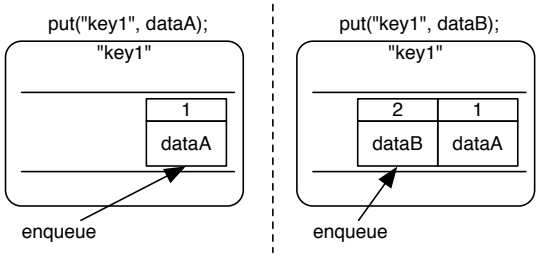

図 1 put はデータを追加する

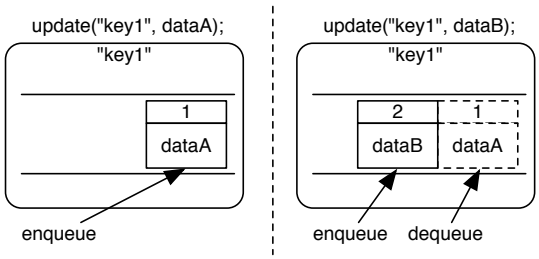

図 2 update はキューの先頭を書き換える

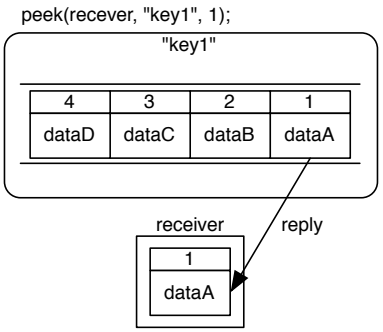

図 3 peek はデータを調べる

2.3 peek

`peek` はデータを調べる

最新の Data Segment がなければ、Code Segment の待ち合わせ (Blocking) が起きる。(図 4)

`put` や `update` により Data Segment の更新があれば、`peek` が直ちに実行される。つまり、Data Segment を作成した Code Segment が active queue に移される。

2.4 take

`take` もデータを読み込むための API である。読み込まれたデータは Key Value Store のキューから取り除かれる。これは、Linda の in() に相当する。(図 5) 必要な待ち合わせが行われる。

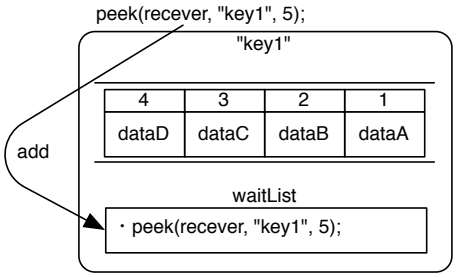

図 4 希望のデータが無いときは保留する

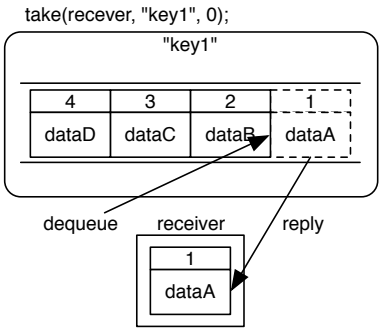

図 5 take はデータを読み込む

3. Data Segment の実装

Data Segment のデータの表現には MessagePack を利用している。MessagePack に関して Java におけるデータ表現は以下の 3 段階あり、これらのデータ表現は制限を伴うが互いに変換かのである。

- 一般的な `Java` のクラスオブジェクト
- `MessagePack for Java` の `Value` オブジェクト)
- `byte[]` で表現されたバイナリ

Data Segment API では、この MessagePack for Java の Value オブジェクトを用いてデータが表現されている。MessagePack は Java のように静的に型付けされたオブジェクトではなく、自己記述なデータ形式である。MessagePack for Java の Value オブジェクトは MessagePack のバイナリにシリアライズできる型のみで構成された Java のオブジェクトである。そのため、Value も自己記述式のデータ形式になっている。

Value オブジェクトは通信に関わるときには、シリアライズ・デシリアライズを高速に行うことができる。また、ユーザーはメソッドを用いてオブジェクト内部のデータを閲覧、編集することができる。

ユーザーが一般的なクラスを IDL(Interface Definition Language) のように用いてデータを表現することができる。この場合、クラス宣言時に@Message というアノテーションをつける必要がある。(ソースコード 1) もちろん、MessagePack で扱うことのできるデータのみをフィールドに入れなければならない。

Listing 1 一般的なクラスを IDL のように使用

```
1  import org.msgpack.annotation.Message
2
3  @Message
4  public class MessagePackTest {
5      public String key;
6      public int val;
7  }
```

4. Code Segment

Code Segment はタスクのことである。Code Segment をユーザーが記述するときに、Code Segment 内で使用する Data Segment の作成を記述する。Code Segment には、Input Data Segment と Output Data Segment を作る API が存在する。

Input Data Segment で作成された Data segment は、remote か local かと、key を指定する必要がある。Input Data Segment がすべて揃わないと Code Segment は active にならない。

Output Data Segment で作成された Data segment にも、remote か local かと、key を指定する必要がある。Input/Output が Code Segment 間の依存関係を自動的に記述することになる。

`ids,ods` により、Input/Output を選択して Data Segment を作成する。Output には put 時にキーを指定する (図 2)。Input は setKey を使ってキーを指定する。もちろん、`cs.width` のようにアクセスする

Listing 2 Data Segment の例

```
1  public class SendWidth extends
       CodeSegment {
2      Receiver width = ids.create(CommandType.
           PEEK);
3      @Override
4      public void run() {
5          int width = this.width.asInteger();
6          ods.put("parent", "widths", width);
7          System.out.println("send␣widths:␣" +
               width);
8          SendWidth cs = new SendWidth();
9          cs.width.setKey("local", "width", this.
               width.index);
10     }
11 }
```

Listing 3　Start Code Segment を実行させる方法

```
1 public class TestLocalAlice {
2     public static void main(String args[]) {
3         new StartCodeSegment().execute();
4     }
5 }
```

のは Java 的には正しくない書き方であり避けるべきである。`SendWidth` は Code Segment であり、Data Segment が揃った時に、 `Runnable` のように実行される。`SendWidth` 内部で setKey する方が Java 的には望ましい。

どの時点でキーとノードを指定するか、どのような API を用意するべきかは、まだ、議論の余地がある。

5. Code Segment の実行方法

Alice には、Start Code Segment (ソースコード 4) という C の main に相当するような最初に実行される Code Segment がある。Start Code Segment はどの Data Segment にも依存しない。つまり Input Data Segment を持たない。この Code Segment を main メソッド内で new し、execute メソッドを呼ぶことで実行を開始させることができる。(ソースコード 3)

6. Code Segment の記述方法

Code Segment をユーザーが記述する際には Code Segment を継承して記述する。(ソースコード 5) その CodeSegment は InputDataSegmentManager と OutputDataSegmentManager を利用することができる。

InputDataSegmentManager は Code Segment の `ids` というフィールドを用いてアクセスする。

- `Receiver create(CommandType type)`

create でコマンドが実行された際に取得される Data Segment が格納される受け皿を作る。引数には CommandType が取られ、指定できる CommandType は `PEEK` または `TAKE` である。

- `void setKey(String managerKey, String key)`

setKey メソッドにより、どこの Data Segment のある key に対して peek または take コマンドを実行させるかを指定することができる。コマンドの結果がレスポンスとして届き次第 Code Segment は実行される。

OutputDataSegmentManager は Code Segment の `ods` というフィールドを用いてアクセスする。OutPutDataSegmentManager は `put` または `update` を実行することができる。

Listing 4　StartCodeSegment の例

```
1  public class StartCodeSegment extends
       CodeSegment {
2
3      @Override
4          public void run() {
5          System.out.println("run␣
               StartCodeSegment");
6
7          TestCodeSegment cs = new
               TestCodeSegment();
8          cs.input1.setKey("local", "key1");
9
10         System.out.println("create␣
               TestCodeSegment");
11
12         ods.update("local", "key1", "String␣
               data");
13     }
14
15 }
```

Listing 5　CodeSegment の例

```
1  public class TestCodeSegment extends
       CodeSegment {
2      Receiver input1 = ids.create(CommandType
           .PEEK);
3
4      @Override public void run() {
5          System.out.println("index␣=␣" +
               input1.index);
6          System.out.println("data␣=␣" + input1.
               val);
7
8          if (input1.index == 10) System.exit(0);
9
10         TestCodeSegment cs = new
               TestCodeSegment();
11         cs.input1.setKey("local", "key1",
               input1.index);
12         ods.update("local", "key1", "String␣
               data");
13     }
14 }
```

- `void put(String managerKey, String key, Value val)`
- `void update(String managerKey, String key, Value val)`

7. Topology Manager

Alice は複数のノードで構成され、相互に接続される。通信するノードは、URL などにより直接指定するのではなく、TopologyManager によって管理される。

TopologyManager 関連の通信処理は Code Segment で実装してある。TopologyManager はトポロジーファイルを読み込み、参加を表明したクライアント (以下、Topology Node) に接続するべきクライアントの IP アドレスやポート番号、接続名を送り、ト

Listing 6　3 台でリングを組んだ時の例

```
1 digraph test {
2         node0 −> node1 [label="right"]
3         node0 −> node2 [label="left"]
4         node1 −> node2 [label="right"]
5         node1 −> node0 [label="left"]
6         node2 −> node0 [label="right"]
7         node2 −> node1 [label="left"]
8 }
```

ポロジーファイルに記述された通りにトポロジーを作成する。

Code Segment 内部で remote DS manager にアクセスする場合は、Topology Manager によって指定されたノード内部だけで有効な label(文字列) を使う。これにより、特定の URL が Code Segment 内部に記述されることを防いでいる。

7.1　Topology Manager の設定ファイル

Topology Manager はトポロジーファイルを読み込むが、トポロジーファイル自体は DOT Language[2] という言語で記述される。DOT Language とはプレーンテキストを用いて、データ構造としてのグラフを表現するための、データ記述言語の一種である。この DOT Language のグラフを利用して、クライアント間の接続を表現する。DOT Language ファイルは dot コマンドを用いて、グラフの画像ファイルを出力することができるので、記述したトポロジーが正しいことを可視化して確認することができる。

クライアント間の接続には label を用いて名前が割り振られており、この接続名を用いてユーザーは Data Segment Manager にアクセスすることができる。前述した Receiver に setKey を行う際、ods で put または update する際の引数の managerKey がこれにあたる (図 6)。

Alice の Node を起動する際にコマンドライン引数として Topology Manager の IP アドレスとポート番号を指定をする。そして main 関数内で TopologyNode を new を行えば良い。TopologyNode の第一引数は Alice デーモンの設定オブジェクト、第二引数は Start Code Segment である。ここで指定した、Start Code Segment がトポロジーが完成した後実行される。

8. 水族館の例題

今回作成した例題は水族館である。複数のクライアントのディスプレイを複数の魚が移動していくものである。魚は画面の端まで移動すると自分の画面上からは消え、別のクライアントの画面の端から魚が出てくる。また、魚のうち一匹はクライアントが直接操作すること

- `dot -T png ring.dot -o ring.png`

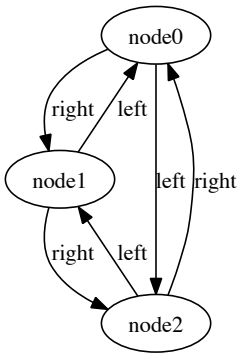

図 6　dot コマンドで作成された 3 台で構成されたリングのグラフ

ができる。トポロジーは TopologyManager によりツリー状に構成してある。

(1) ユーザーが魚を操作するまたは Code Segment により魚の座標が更新される。
(2) 画面に表示させるための SetLocation (Code Segment) が実行され実際に魚のオブジェクトにセットされ画面に反映される。
(3) Update(Code Segment) に FishPosition(魚の座標データ) が渡される。
(4) Update に list(送信者リスト) が渡される。
(5) Update が実行され、list を元にデータが送信される。ただし、この時に FishPosition には送信元情報が付加されているので、送信元には送信されない。
(6) 各 client で 2 - 4 が実行される。

9. 実　　験

Ring 上のトポロジーを構築して、100 周回時間を測定してみた。マシン 48 台,CPU Intel(R) Xeon(R) X5650 @ 2.67GHz, 仮想コア数 4,CPU キャッシュ 12MB。Blade 上の仮想マシン上での測定となっている。従来の Federated Linda よいも若干遅い結果になっている。一部の異常なデータがあるが、データ量が増えると差は縮まっている。これは、コピーの影響よりも、個々の通信の手間の影響が大きいことを示している。00

10. 評価と考察

今回の実装は、Java により Code Segment と Data Segment に必要な API を洗い出すためのものであった。この実装でもいくつかの問題が明らかになっている。

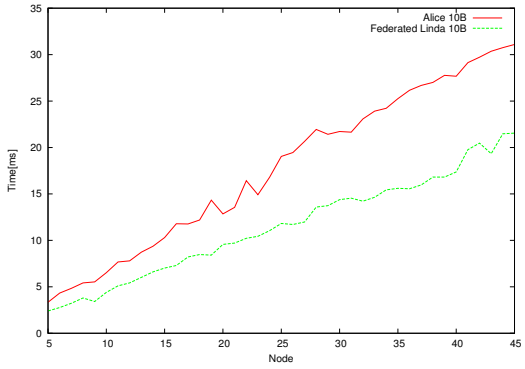

図 7 10 bytes のデータを 100 周させたときの 1 周にかかる平均時間

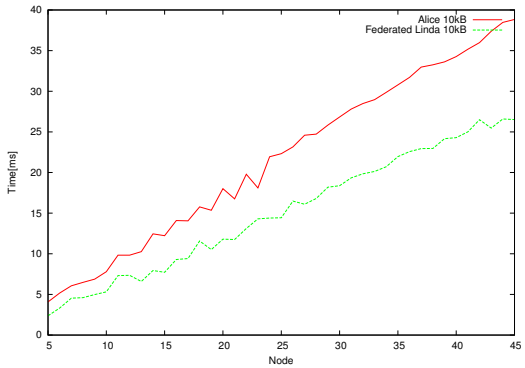

図 8 10 Kbytes のデータを 100 周させたときの 1 周にかかる平均時間

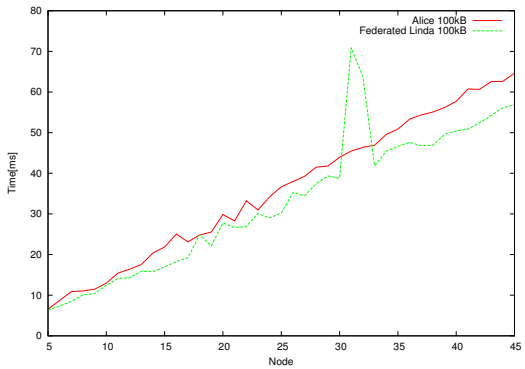

図 9 100 Kbytes のデータを 100 周させたときの 1 周にかかる平均時間

API Class を継承したり、Input Data segment や Output Data segment の作成に factory object を使うのは Java を使う際の技術的なものであり、Alice の API 自体は Java に固有である必要はない。むしろ、Java の Object 指向な記述が全体を煩雑にしている部分がある。`update` は、Data Segment の競合的な更新に使われるべきだと思われる。

SEDA Federated Linad に比べて、通信のレスポンスが遅い原因の一つは SEDA architecture のせいだと思われる。SEDA はスループット重視の実装であり、多段のパイプラインのせいでレスポンスは遅れてしまう。実際、スレッドプールを使用しないほうが、Ring の結果は良くなる (図 10)。

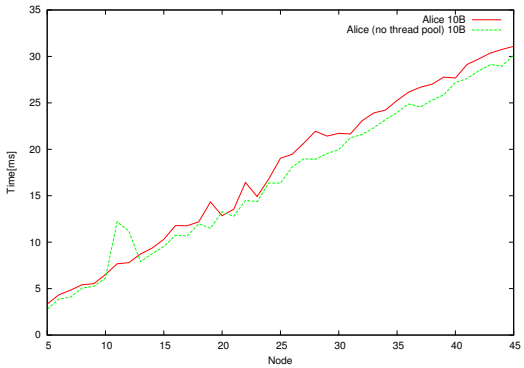

図 10 Code Segment のスレッドプールを使用せずに、 10 bytes のデータを回した時の実験結果

レスポンスが要求される部分のスケジューラーを別にするなどの工夫が必要だと思われる。それを記述そのものに入れるのが良いかどうかには議論の余地がある。

MessagePack 今回の実装では単純な Message の転送の場合でも MessagePack の decode/encode が必要になる。これは単純に overhead になる。encode/decode 抜きに直接処理できる方が望ましい。また、Data Segment の一部の修正に Data Segment を再構成するのは望ましくない。Cerium では Input Data Segment と Output Data Segment を swap する API があり、若干状況は複雑になるが良好な結果を得ている。この辺りは、ユーザから見えない最適化として実装する方が望ましいが、なんらかの制御方法も必要だと思われる。

Key 本実装では Data Segment 相互の参照は Key 経由となる。Linda や分散実装では、それは妥当だが、並列実装では、すべての Data Segment を Key Value Store に格納するのは性能的な問題を引き起こす。一方で、分散記述と並列記述がかけ離れてしまうのも好ましくない。現状は Key Value Store は Java の Concurrent Hash map を用いているが、今のベンチマークでは、そこがネックになっているわけではないと思われる。本来は、この Key Value Store は持続性を持つべきだと思われるが今回は実装していない。

Java Ring の実験での異常なデータは、Java の

分散プログラミングでは良く現れる。一つは Java の GC の影響だと思われる。Alice では、すべての Data Segment は Key Value に格納され、実行時の Data Segment は Code Segment が active な時のみにメモリ上にある。この最大値を見積ることは、Active Task の量を見積もれば良い。したがって、Alice には Garbage Collection は必要ない。一方で、Key Value Store 上のデータは決して Garbage Collection の対象にならない。しかし、それは Garbage Collector には負荷をかけてしまう。つまり、Alice 自体は Java で実装するのには向いていない。

拡張性分散アプリケーションでのプロトコルは常に変更されるものであり、Alice もそれに対応する必要がある。Alice 上で走るプロトコルは、Data Segment と Code Segment によって決まる。Key とトポロージーマネージャーをプロトコル毎に別に用意すれば、複数のプロトコルを同時に走らせることが可能である。プロトコル間の互換性はいろいろあり得る。Data Segment と Code Segment の結びつきは弱いので、Data Segment に余計な値がある場合、あるいは、値が足りない場合に適切な値を設定することにより、古い Code Segment を変更せずにプロトコルを拡張できると考えている。実際、水族館の例題で、2 次元と 3 次元版を両立させることは容易である。

11. まとめと今後の課題

今回、Code Segment と Data Segment による並列分散フレームワークの Java による実装を示した。前の設計[8] と異なる部分は、実装でしか得られない治験である。

Java による実装は、ラピッドプロトタイピングとしては適切であり、例題の記述と基本性能の確認に向いている。一方で、Java が Alice の実装に不向きであることもわかってきた。Code Segment / Data Segment を見たコンパイラ的アプローチ、あるいは実行時最適化などが有効であると思われる。あるいは、CbC[3] による実装が効果的だと考えている。

特に、今回はノード内の並列実行や GPGPU による並列実行などは考慮していないので、将来的には、それを含めた実装をしていきたい。

参 考 文 献

1) Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, Aug. 1986.
2) AT&T Research. Graphviz - graph visualization software. `http://www.graphviz.org/Documentation.php`.
3) Shinji KONO. CbC. `http://sourceforge.jp/projects/cbc/`, March 2008.
4) Shinji KONO. Cerium. `http://sourceforge.jp/projects/cerium/`, March 2008.
5) Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, Vol.35, No.5, pp. 230–243, 2001.
6) 安村恭一, 河野真治. 大域 ID を持たない連邦型タプルスペース Federated Linda. 情報処理学会システムソフトウェアとオペレーティング・システム研究会, May 2005.
7) 赤嶺一樹, 河野真治. Meta Engine を用いた Federated Linda の実験. 日本ソフトウェア科学会第 27 回大会 (2010 年度) 論文集, Sep 2010.
8) 赤嶺一樹, 河野真治. Data Segment API を用いた分散フレームワークの設計. 日本ソフトウェア科学会第 28 回大会 (2011 年度) 論文集, Sep 2011.
9) 河野真治, 島袋仁. C with Continuation と、その PlayStation への応用 . In *IPSJ OS, CPSY*, May 2000.
10) 多賀野海人, 小林佑亮, 宮國渡, 河野真治 (琉球大). Cell Task Manager Cerium の SPU 内データ管理. 情報処理学会システムソフトウェアとオペレーティング・システム研究会, April 2009.