Javaを用いた広域並列計算システムNinflet上の通信クラスライブラリの実現

大久　光崇 †　高木　浩光 ††
松　岡　聡 †　小川　宏高 †

Javaを含むネットワーク可搬性が高い言語処理系を用いて、 夜間などに稼動していない遊休計算機資源を活用した広域並列計算システムが数多く研究されており、我々のNinfletシステムはその一つである。しかし、これらのシステムでは、単純なマスターワーカーモデルの並列計算でしか評価していないことが多く、また、 RMIやMPIなどの低レベルな通信プリミティブをそのまま使用し繁雑なプログラミングス タイルをとっている。本研究では、広域並列計算を容易にプログラムすることを支援するために、通信を含む並列処理のアルゴリズムを、デザインパターンを用いて抽象化したNinfletシステム用のクラスライブラリを実装し、従来のプログラミング手法と比較、検討する。

Implementation of Communication Library on Ninflet : A Java-based Global Parallel Computing System

Mitsutaka Oohisa ,† Hiromitsu Takagi ,††
Satoshi Matsuoka † and Hirotaka Ogawa †

There have been several recent proposals of high-performance distributed systems that utilize idle computing resources during the nights, etc. These systems typically employ highly portable programming language systems such as Java, and our Ninflet is one such system. However, evaluation of these systems have been mostly done with simple master-worker styles only, and more complex parallel programming styles have resorted to low-level communication primitives such as RMI and MPI. Instead, we design and encapsulate several high-level parallel programming patterns as class libraries for Ninflet using the so-called 'design patterns', and evaluate its effectiveness by comparing with traditional parallel Programming styles.

1. はじめに

インターネットの大衆化等により、ネットワークに接続された計算機は爆発的に増加し続けている。この中には夜間や休み時間などに稼動していない「遊休計算機」が数多く含まれているだろう。これらの遊休資源を広域分散環境で有効活用することを目指した「ハイスループット・コンピューティング」[1]の研究が活発化してきている。

広域分散環境での遊休計算機の活用のためには、ポータビリティとセキュリティが要求される。プログラムはその作者が利用権を持つ計算機以外で実行可能でなくてはならないため、任意のプロセッサアーキテクチャおよびOS上で動作可能である必要があり、また、どのような内容のプログラムが実行されようともその計算機の所有者に不利益を与えないための安全性の確保が必須となるからである。

このようなポータビリティとセキュリティを実現する手段の一つとしてとして、ベース言語としてJavaを採用すること挙げられる。 Javaのプログラムはバイトコードで配信され、実行時にJIT(Just-In-Time)コンパイラによってプロセッサのネイティブコードに変換されて高速実行される仕組みとなっているため、バイトコードレベルでポータビリティが保証される。また、バイトコードは起動時にはインタプリタ方式で解釈されるため、ローカル資源に対するI/O等を確実に禁止することが容易であり、確かなセキュリティを保証することができる。

このような背景から、 Java Appletをベースとした広域分散計算システムが数多く提案されており、暗号耐性検査等のアプリケーションに活用されている。これに対し我々のNinfletシステム[2][3]では、任意のユーザが、任意のアプリケーションのために、より安定的に継続して利用できるハイスループット・コンピューティングシステムとすることを目指して、非Appletベースの専用システムを構築している。

これらの一般的な広域分散計算システムでは、並列計算のモデルとして、 Master-Workerパターンを採用している場合が多い。広域ネットワークのように通信遅延が無視できない環境においては、通信遅延を隠蔽するため、また、均一でないタスク処理時間を分散させるために、 Master-Workerパターンはたしかに有効である。

これに対しNinfletシステムは、広域環境のみならず、 LAN上での「キャンパスワイド分散計算」の実現のため、また、ワークスステーション /PCクラスタ上

† 東京工業大学 Tokyo Institute of Technology
†† 電子技術総合研究所 Electrotechnical Laboratory

に並列処理環境を実現するための利用も想定しており、広域環境からクラスタまでをシームレスに統一された計算環境として利用することを目指している。例えば、単純な Master-Worker パターンでは並列化できない問題を、その問題に適した、より密な通信をする並列プログラムとして記述し、それ全体をひとつのジョブとして、広域環境のどこかに存在するクラスタシステムで実行させるといった処理を実現したい。

この実現のために、Ninflet システムでは、Master-Worker 以外の様々な並列処理パターンをクラスライブラリとして提供していく。本稿ではそのひとつとして、リダクション処理を繰り返す並行プロセスの抽象化例を示す。

従来、こうしたアプリケーションは、MPI 等の低レベルの通信プリミティブを用いて書かれてきた。そこでは、プログラマ自身が通信の必要な個所に通信プリミティブをそのまま埋め込む方法がとられている。このような構造化されていない通信の記述スタイルは、プログラムのメンテナンスを難しくするだけでなく、Ninflet システムのようなシステムにおいては、プログラムの各部分を、実行時に利用可能な資源に応じて動的に、かつ任意に計算機に発行することが求められるが、それを難しくする。そこで、並列アルゴリズムを、オブジェクト指向デザインパターン[4] の枠組みにしたがって抽象化し、通信パターンを、個々の計算内容から分離することを試みる。

以下、2 章で、Ninflet システムの概要を紹介し、3 章で、オブジェクト指向デザインパターンを適用した並列プログラミング用ライブラリの構成法、それを用いた CG 法のプログラミング例を示す。そして 4 章で、従来の MPI によるプログラミングとの比較について述べ、5 章で関連研究について述べる。

2. Ninflet

2.1 Ninflet システムの概要

Ninflet システムの利用者には、計算機の貸し出しをする計算機提供者の立場と、計算の依頼を発行するユーザプログラマの立場がある。Ninflet システムは、2 つの daemon プログラム “Ninflet Server”(以下単に Server) と、“Ninflet Dispatcher”(以下単に Dispatcher) と、Ninflet プログラム (以下単に Ninflet) から構成される。Ninflet システムは、Java 専用の分散オブジェクト技術と独自の機能により、一種の並行オブジェクトである Ninflet をリモートにある Server 上で実行することで、広域並列計算システムを実現している。

- **ユーザプログラマ:** 実行させたい計算を 2 つのプログラム (Ninflet プログラム、クライアントプログラムとして実装する。
- **Ninflet:** Server 上で実行され、マイグレーションが可能な並行オブジェクト。
- **クライアントプログラム:** ユーザプログラマの所有しているホスト上で実行され、Ninflet の発行と中止を管理する役割をもつ。
- **Server:** 貸し出される計算機上で実行される daemon プログラムであり、URL で指定された Ninflet をロードして起動する役割と、その計算機所有者からの指示によって実行中の Ninflet を退去させる役割を担う。
- **Dispatcher:** Server からの参加登録要求に基づいて、現時点で使用可能な Server のリストを管理し、クライアントからの Ninflet 発行要求に応じて、Ninflet をどの Server 上で実行させるかのスケジューリングをおこなう。
- **HTTP/FTP サーバ:** ユーザプログラマはあらかじめ Ninflet(とそれが利用するクラス群) を HTTP/FTP サーバに置く。Server は HTTP/FTP サーバから Ninflet をロードする。

2.2 Ninflet システムの起動と利用手順

Ninflet システムの起動から、ユーザプログラムの起動までを順を追って説明する。

(1) Ninflet システムの管理者によって、Dispatcher daemon がどこかのホスト上で起動される。
(2) 計算機提供者は、その計算機上で Server を daemon プロセスとして稼動させる。このとき 1 つの Dispatcher を URL で指定する。Server daemon は指定された Server に自分自身を登録する。
(3) ユーザプログラマは Ninflet を HTTP/FTP サーバ上に置き、クライアントプログラムを実行する。クライアントプログラムは、起動したい Ninflet のクラス名と URL を Codebases で指定して、Dispatcher を呼び出す。
(4) ユーザプログラマはクライアントプログラム内で、Ninflet のインスタンス化を Dispatcher に依頼する。
(5) Dispatcher は、利用可能な Server のリストから、あるスケジューリングアルゴリズムに基づいて、ひとつの Server を選択し、その Server に、Ninflet のインスタンス化の依頼を委譲する。このとき Codebase を渡す。
(6) Server は、この Codebase をもとに HTTP/FTP サーバから Ninflet をロードし Ninflet のインスタンスを生成する。このインスタンスへのリモートリファレンスは Dispatcher へ、そしてクライアントプログラムへと返される。
(7) クライアントプログラムはこのリモートリファレンスをもとに Server 上の Ninflet インスタンスを直接呼び出す。

Ninflet システムでは、分散オブジェクトを実現する低位のミドルウェアとして RMI を使用している。また、独自のクラスローダ、専用のセキュリティマネージャを持ち、オブジェクトのマイグレーションをサポートする、RMI を拡張した ORB を持つ。

3. デザインパターンによる並列アルゴリズムのためのクラスライブラリの構築

Ninflet システムでは、Master-Worker 以外にも、様々な並列処理のパターンをクラスライブラリとして提供していく。ここではそのひとつとして、リダクション処理を繰り返す並行プロセスのパターン化の例を示す。

3.1 オブジェクト指向デザインパターン

Ninflet システムでは、ユーザプログラマは予め用意されているクラスのサブクラスを作成することによって、目的のプログラムを構築する。文献[3] で示した Master-Worker パターンのクラスライブラリでは、オブジェクト指向デザインパターン[4] における、Template Method パターンを適用し、ユーザプログ

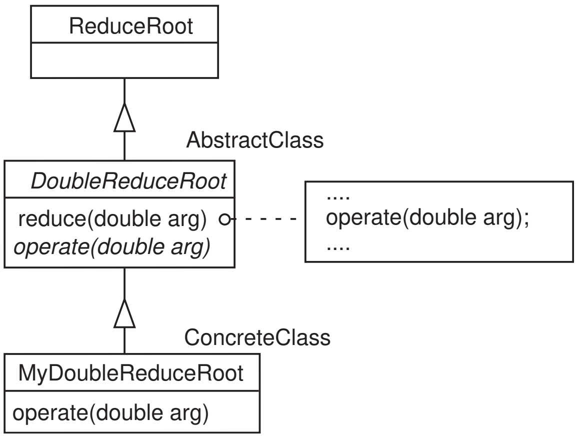

図 1　Template Method パターンによるユーザ定義リダクションの記述

ラマは、自分が記述したい問題に固有の、差分だけを記述するものとした。

Ninflet システムでは、流体問題などのためのより高レベルなクラスライブラリを用意する予定である。このような高レベルなクラスライブラリを効率よく実装するためには、部品となる中小レベルのクラスライブラリが必要となる。これらを、メンテナンスが容易であると同時に効率的に実装するために、より広範にオブジェクト指向デザインパターンを用いることにした。

今回示すライブラリは、 Template Method パターンの他に、 Strategy パターンも適用した構成となっている。まず以下に、 Template Method パターンおよび Strategy パターンの一般的な特質について説明する。

3.1.1　Template Method パターン

Template Method パターンは、複数のアルゴリズムを共通のインタフェースで利用する場合などに利用される。 1 つのメソッドに複数のアルゴリズムから共通するステップを抽出しスケルトンとして定義する。いくつかの異なるステップを抽象メソッドとし、サブクラスにその実装を任せることにする。これにより、抽象メソッドをサブクラスごとで定義するだけで、アルゴリズムを完全に実現した、サブクラスごとに特有なメソッドを定義することができる。 Template Method パターンは以下のクラスより構成される。図 1はリダクション処理に Template Method パターンを適用した例である。

- **AbstractClass** クラス
  各サブクラスで特有なメソッドを、 primitive operation と呼ばれる抽象メソッドとして定義する。 primitive operation を呼び出すことで、 template method をアルゴリズムのスケルトンとして実装する。図 1では、リダクション処理の演算部分を抽象化しその他の部分のステップを記述した Template Method を reduce() と定義し、 primitive operation としてリダクション処理の演算部分を operate() と定義している。
- **ConcreteClass** クラス
  アルゴリズムのステップを特有の形で実行するために、 primitive operation を実装する。図 4 では総和をとるための演算を実装している。

図 1では AbstractClass クラスを DoubleReduceRoot クラス、 ConcreteClass クラスを MyDoubleReduceRoot としている。ユーザは、 primiteve operation を MyDoubleReduceRoot クラスで実装するだけで、様々なリダクション処理を実現することが可能となる。

3.1.2　Strategy パターン

Strategy パターンは、 Template Method パターンのように複数のアルゴリズムを共通のインタフェースで利用する場合などに利用されるが、アルゴリズム全体の実装がサブクラスに委ねられている点で異なる。個々のアルゴリズムを共通のインタフェースを持つように定義することにより、それらを利用するオブジェクトとは独立に、それぞれのアルゴリズムを全く異なる仕様に実装することが可能となる。 Strategy パターンは以下のクラスより構成される。図 2はプログラムのメインループ内に Strategy パターンを適用した例である。

- **Strategy** クラス
  個々のアルゴリズムに共通のインターフェースを宣言する。図 2で Calculator クラスはインタフェースとして init()、 run()、 set() を持つ。
- **ConcreteStrategy** クラス
  Strategy クラスのインタフェースについて、個々のアルゴリズムを実装する。後述の CG 法の例では、メインループ内の計算を 4 つに分け、それぞれを ConcreteStrategy クラスとして実装し、 run() に計算の内容を記述する。
- **Context** クラス
  個々のアルゴリズムを使い分けるクラスであり、利用するアルゴリズムの実装された ConcreteStrategy オブジェクトを保持し、その参照の型は Strategy クラスとする。また、 Strategy クラスが Context クラスのデータにアクセスするためのインタフェースを定義してもよい。図 2 では Block クラスが CalcAlpha オブジェクト、または CalcBeta オブジェクトを Calculator クラスのオブジェクトとして保持している。

3.2　リダクションライブラリ

リダクション処理を含む並列処理では、計算とリダクション処理の繰り返しとなるパターンが多い。よって、この並列処理パターンを、 Template Method および Strategy パターンを適用して汎用ライブラリ化する。

まず、メインループを Iterator クラスとする。 Iterator クラスは、ループ構造を持ち、ループ内は Block クラスで表される。 Block は計算とリダクションのペアであり、それぞれ Calculator クラスと Reducer クラスである。 Calculator と Reducer には Strategy パターンを適用し、ユーザプログラマは、 Calculator をサブクラス化して目的の問題を記述し、これのインスタンスを Block に登録するようにする。 Reducer については、予め用意されている Reducer のサブクラスの中から、記述したい問題に利用できるものがあればそれを選び、なければ必要に応じ新たに作成して、そのインスタンスを Block に登録するようにする。

- **Calculator:** Ninflet のメインループの計算の一部を担当するオブジェクト。インタフェースとして、このオブジェクトの変数を設定するメソッド init()、このオブジェクトの実行結果を Ninflet オブジェクトに反映させる set() を持ち、このオブジェクトを実行させるメソッド run() を持つ。

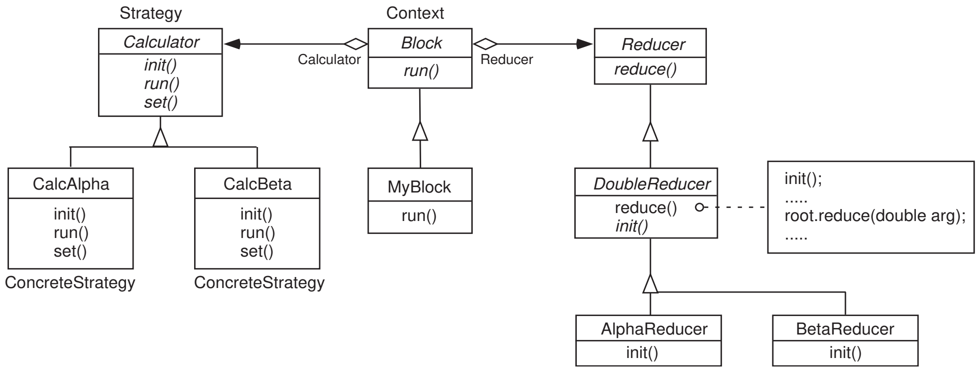

図 2 Strategy パターンによるメインループ内の記述

Strategy パターンを用い、各ブロックの計算をカプセル化する。

- **Reducer:** Ninflet のメインループ内のリダクションの実行を ReduceRoot オブジェクトに依頼するオブジェクト。様々なリダクションアルゴリズムを提供するためにここで Template Method パターンを用いている。primitive operation として、Ninflet のどの変数をリダクションするかという初期化のためのメソッド init() とリダクションの結果を Ninflet に反映させるためのメソッド set()(reduceScatter を用いた時) を持ち、これらの実装は IntReducer、ObjectReducer などのクラスのサブクラスで記述される。インタフェースとして reduce()、reduceScatter() を持つ。
  Java ではプリミティブ型 (int, double など) をオブジェクトして扱うことができないため、それぞれ用の Reducer として IntReducer、DoubleReducer を用意した。プリミティブ型の値をラッパーオブジェクトに包んで扱うことも可能であるが、利用の煩雑さと実行効率の悪さを避けるためにこのようにした。配列に対するリダクションのためには、ObjectReducer を必要に応じてサブクラス化して独自の演算を定義して使う。
- **Block:** メインループ内の計算 (Calculator) とリダクション (Reducer) をペアにしたオブジェクト。メソッド run() によりこれらを実行する。
- **Iterator:** メインループ内の各 Block のオブジェクトを配列として持ち、メソッド run() の実行により、この配列が指し示す各 Block を実行する。
- **ReduceRoot:** 各 Ninflet の Reducer オブジェクトからの依頼を集中的に管理するオブジェクト。このクラスでも Template Method パターンを用いている。このオブジェクトは独立した Ninflet であり、プリミティブ毎に IntReduceRoot などが用意されている。リダクションの演算は primitive operation とし IntReduceRoot などのクラスのサブクラスでメソッド operate(int arg) などとして実装する。

ここで、Reducer クラスと ReduceRoot クラスを例にしてデザインパターンのクラスライブラリへの適用を説明する。図 3 は Template Method パターンが適用された DoubleReducer クラスの一部であり、メソッド reduce() は DoubleReducer オブジェクトから呼び出されるリモートメソッドである。リダクションの演算を primitive operation であるメソッド operate() に任せサブクラスで実装する。

```
public void reduce(double arg) {
  synchronized (this) {
    operate(arg);
    count++;
    if (count == numOfTasks) {
      reducedValue = tmp;
      tmp = 0;
      count = 0;
      release();
    }
  }
}

public abstract void operate(double arg);
```

図 3 DoubleRoot クラスのメソッド reduce()

```
public void operate(double arg) {
  tmp += arg;
}
```

図 4 DoubleRoot クラスのサブクラスのメソッド operate()

3.3 CG 法を実装する例

このようにして用意したライブラリの使用例として、CG 法を実装してみる。実装した CG 法のアルゴリズムを図 5 に示す。このアルゴリズムでは、3 箇所でリダクション処理を行う。

まず、メインループ内の (1) から (10) までのルーチンを Calculator クラスと Reducer クラスのサブクラスとしてそれぞれ実装する。(1) を Calculator クラスのサブクラス (図 6)、(2) を Reducer クラスのサブクラスとする。次に、ブロックを決定する。ここでは (1) と (2)、(3) と (4)、(5) から (9)、そして (10) と 4 つのブロックを考え、それぞれ Block クラスのサブクラスとして定義する。最後のブロックのように、Calculator オブジェクトのみのブロックについては、何もしない DummyReducer オブジェクトを Block クラスのコンストラクタで追加する。こうしてできた

$x_0 =$ 初期推定ベクトル; $\quad r_0 = b - Ax_0$
$p_{-1} = \mathbf{o}$; $\quad \beta_{-1} = 0$
$\rho_0 = (r_0, r_0)$
$i = 0, 1, 2, \cdots$ に対して
(1) $\quad p_i = r_i + \beta_{i-1} p_{i-1}$
(2) $\quad p$ についてリダクション
(3) $\quad q_i = Ap_i$
(4) $\quad \alpha$を求めるために(p_i, q_i) の値をリダクション
(5) $\quad \alpha_i = \rho_i / (p_i, q_i)$
(6) $\quad x_{i+1} = x_i + \alpha_i p_i$
(7) $\quad r_{i+1} = r_i - \alpha_i q_i$
(8) $\quad \rho_{i+1} = (r_{i+1}, r_{i+1})$
(9) $\quad \rho$ についてリダクション
(10) $\quad \beta_i = \rho_{i+1} / \rho_i$
(11) $\quad x_{i+1}$ が正確であれば終了
ループ終了

図 5 CG 法のアルゴリズム

```
public void init() {
  // run() で使用するフィールドの初期化を行う
}

public void run() {
  init();
  for (int i = 0; i < partialSize; i++) {
    p[i] = r[i] + beta * p[i];
  }
  set();
}

public void set() {
  // p を Ninflet オブジェクトのフィールドに代入
する
}
```

図 6 Calculator クラスのサブクラスの実装

```
public void run() {
  calculator.run();
  reducer.reduceScatter();
}
```

図 7 Block クラスのサブクラスの実装

Block のサブクラス (図 7) は Iterator クラスのフィールドに配列として格納され、メソッド run()(図 8) によって順次実行される。(11) のチェックについては Iterator クラスのサブクラスで実装する。

一方、リダクション処理を集中的に管理するために、α、ρ 用の double 型の変数のリダクション処理を行うオブジェクト、p 用の配列のリダクション処理を行うオブジェクトのクラスを、それぞれ DoubleReduceRoot、ObjectReduceRoot のサブクラスとして実装する (図 3)。これらのクラスの

```
public void run() {
  for (int i = 0; i < block.length; i++) {
    block[i].run();
  }
}
```

図 8 Iterator クラスのメソッド run()

primitive operation であるメソッド operate() でリダクションの演算を実装する。(図 4)

4. MPI による実装との比較

CG 法を例に、今回示したデザインパターンを適用したクラスライブラリを用いる場合と、MPI を用いる場合とで、その実装方法の特質を比較する。

3.2 節で示した CG 法のアルゴリズムは、MPI を用いて実装した場合には、そのメインループは図 9 のようになる。ループの各イタレーションで、3 種類のリダクションと 4 種類の計算がなされており、それぞれはループボディ中に並置される。

プログラミングおよびメンテナンスの容易さについては、この程度の小規模の問題では、両者の違いは小さく、それぞれのスタイルに対する習熟度の違いといえてしまうかもしれない。しかし、広域分散環境における実行効率の面で次の違いがある。

MPI によるこの実装は、起動時に固定的に決定された本数のプロセス群によって始まりから終りまで実行されることを前提としている。このような、伝統的な、並行プロセス上のメッセージパッシングモデルによる並列アルゴリズムの実装は、たしかに、ワークステーション/PC クラスタのように、均質で安定的な環境においては実行効率上十分なものとなり得るが、Ninflet システムが前提としている、非均質で非安定的な環境においては、リダクションの同期点にプロセスが到着する時刻のばらつきによる負荷の偏在化が無視できなくなること、また、利用可能な計算機数が動的に変化することなどから、Master- Worker パターンで一般的に用いられている self-guided スケジューリングによって負荷分散を図ることが、任意のアルゴリズムにおいて重要となる。

同じ問題を self-guided スケジューリングを行うよう、MPI で記述するには、上に示したものとは全く異なる構造で書かなくてはならない。今回示したデザインパターンを適用したクラスライブラリは、通信のパターンを計算そのものから分離しているため、これらの記述とは独立に、スケジューリング戦略を選択的に適用することができる。

ただし、一般に、このようなオブジェクト指向のプログラミングスタイルでは、メソッド呼び出しが頻発して実効性能が低下することが懸念される。これについては、JIT コンパイラによってインライン展開されて効率的に実行されることが期待できる他にも、広域分散環境ではその通信効率からそもそも計算の粒度は十分に大きく設定されるものであり、今回のクラスライブラリでは通信の発生する部分にしかデザインパターンを適用していないため、そのようなオーバーヘッドは無視できると考えられる。

```
while (true) {
  /* pを求める */
  MPI_Reduce_scatter(&p, &totalp, length,
                     MPI_DOUBLE, &arraycombine,
                     MPI_COMM_WORLD);
  /* qを求め変数mytmpに(p, q)の値を代入す
る */
  MPI_Reduce_scatter(&mytmp, &tmp, 1,
                     MPI_DOUBLE, MPI_SUM,
                     MPI_COMM_WORLD)
  /* tmpの値を用いてαの値を求める */
  /* xとrを求めmytmpに(r, r)の値を代入す
る */
  MPI_Reduce_scatter(&mytmp, &rho, 1,
                     MPI_DOUBLE, MPI_SUM,
                     MPI_COMM_WORLD)
  /* βの値を求める */
  /* xが十分正しければブレイクする */
}
```

図9 MPIによるCG法の実装

5. 関連研究

並列プログラミングにおいて、コードの再利用を目的とした研究に事例ベース並列プログラミングシステム[5]がある。このシステムでは、まず、ユーザは記述しようとする並列プログラムから特徴を示すインデックスを作成する。そして、このインデックスを用いて事例ベースから最も類似した並列プログラムを検索すると、類似した並列プログラムの骨組みであるスケルトンが提示される。ユーザは、このスケルトンを再利用することによってプログラミングの負担を軽減することができるとしている。

事例ベース並列プログラミングシステムの優れている点は、ユーザがインデックスを作成するだけで、自動的に目的の並列プログラムのスケルトン得ることができることである。しかし、スケルトンからの修正はユーザが自力で行わなければならず、複雑な並列プログラムを行う場合にはユーザの負担が十分に軽便されるとは言い難い。

これに対し本研究では、並列プログラムのスケルトンをデザインパターンを用いたクラスライブラリとしてユーザに提供することでコードの再利用性を実現している。ユーザは、目的にあったクラスライブラリのサブクラスを作成し、クラスライブラリの抽象メソッドをサブクラスで実装する。この弊害として、メソッド呼び出しが多くなりそのオーバーヘッドが危惧されるが、それはJust-In-Timeコンパイラによりインライン展開され最適化されることを期待する。

また、Javaで利用できる通信ライブラリにはPVMをJavaにバインディングしたJPVM[6]、JavaPVMがある。JPVMは100% Pure Javaで実装されており、Javaバーチャルマシンの動作する任意の計算機で利用できる点が特徴である。JavaPVMはネイティブメソッドを用いて既存のPVMシステムを利用する形で実装されており、ほかのプログラミング言語で記述されたPVMのプログラムとも通信可能である。しかし、これらはPVMのAPIを忠実に再現しているだけで、オブジェクト指向プログラミングを提供するものではない。

6. おわりに

Ninfletシステム用のクラスライブラリとして、オブジェクト指向デザインパターンを適用した並列処理のアルゴリズムを提供する方法について示した。クラスライブラリの作成にデザインパターンを用いることにより、より高レベルなクラスライブラリの作成が可能である。デザインパターンを用いたクラスライブラリによるプログラミングとMPIを用いたプログラミングを比較した場合、ヘテロな環境、マイグレーションなどの広域環境による影響から、クラスライブラリによるプログラミングの方がよいと考える。

本研究では、リダクション処理を用いた並列プログラムをパターン化したが、他にもブロードキャスト、バリアなどを用いた並列プログラムをパターン化しライブラリを充実させNinfletの実用性を高める予定である。

参考文献

1) Bricker, A., Litzkow, M. and Livny, M.: CONDOR TECHNICAK SUMMARY, *Computer Sciences Department University of Wisconsin* (1991).

2) Takagi, H., Matsuoka, S., Nakada, H., Sekiguchi, S., Stoh, M. and Nagasima, U.: Ninflet: a Migratable Parallel Objects Framework using Java, *ACM 1998 Workshop on Java for High-Performance Network Computin* (1998).

3) 高木浩光, 松岡聡, 中田秀基, 関口智嗣, 佐藤三久, 長島雲兵: Javaによる大域的並列計算環境Ninflet, 並列処理シンポジウムJSPP'98論文集, pp. 135–142 (1998).

4) Gamma, E., R. Helm, R. J., and Vlissides, J.: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994).

5) 山崎勝弘, 安藤彰一, 朝倉啓之: 事例ベース並列プログラミングシステム, 並列処理シンポジウムJSPP'97論文集, pp. 117–224 (1997).

6) Ferrari, A. J.: JPVM: Network Parallel Computing in Java, *ACM 1998 Workshop on Java for High-Performance Network Computin* (1998).