マルチメディア通信と分散処理 107－15
（２００２．３．２８）

Java言語における部分継続実行の実現手法の提案

佐藤 芳樹† 北形 元† 木下 哲男* 白鳥 則郎†
†東北大学情報科学研究科/電気通信研究所
E-mail: {yoshiki,minatsu,norio}@shiratori.riec.tohoku.ac.jp
*東北大学情報シナジーセンター
E-mail: kino@shiratori.riec.tohoku.ac.jp

本稿では，Java仮想計算機をまたぐスレッド移送機能を分散処理に利用するための既存のスレッド移送技術の問題点を指摘し，これを解決するための手法について述べる．具体的には，関数型言語などで研究されてきた部分継続と呼ばれる概念をJava言語に導入し，部分継続を一つの分散処理の単位とすることで柔軟な粒度での移送を実現する．本稿では，提案手法を実現するための構文の設計と，ソースコード変換による実装について述べる。

Proposal of a Scheme for Implementing Partial Continuations in Java

Yoshiki Sato†, Gen Kitagata†, Tetsuo Kinoshita* and Norio Shiratori†
†Graduate School of Information Science / Research Institute of Electrical Communication, Tohoku University
E-mail: {yoshiki,minatsu,norio}@shiratori.riec.tohoku.ac.jp
*Information Synergy Center, Tohoku University
E-mail: kino@shiratori.riec.tohoku.ac.jp

A scheme has been developed that enables a Java program to be migrated across computers while preserving its execution state. This scheme is called Thread Migration. We indicate the problem occured when applying thread migration on Distributed Processing and propose a solution for that. To be more precise adopting Partial Continuations studied in Functional Language and make that migration unit in Distributed Processing. Thereby we can migrate computation with flexible grain. This paper describes a design of additinal sentence structure to achieve our scheme, and implementing with source-code-level transformation.

1 はじめに

今日，ネットワーク技術の進歩とインターネットに接続されたコンピュータの爆発的な増加に伴い，その利用法は多様化しており，広域分散処理の基盤としてみなされるようになった．分散処理の形態は，ネットワーク環境の規模から見た場合，LANを前提とするクラスタコンピューティング，インターネットを基盤とするグローバルコンピューティングなどに分類できる．

Java言語及びJava仮想計算機は，ネットワーク安全なコード実行，非機種依存性，バイトコードの高速実行というネットワーク向け機能を同時に達成したことで，分散処理にとっても有用な機能を提供している．Javaを用いた分散処理としては，分散オブジェクトや分散共有メモリ，実行状態の移送などの研究が行なわれてきた．

実行状態の移送とは実行中のプログラムをその実行状態を保ったままコンピュータ間で移送する事であり，関数型言語Schemeにおける継続 (Continuation) の分散環境間の移送や分散OSにおけるプロセス移送として研究が行なわれてきた．

Javaにおける実行状態の移送手法の一つであるスレッド移送では，Java仮想計算機のセキュリティ機構により，バイトコードからの実行状態へのアクセスや操作をサポートしていないため，専用のJava仮想計算機の実装や，ソースコード変換による実現手法 [4] が提案されてきた．

スレッド移送を用いて実行状態に基づく分散処理を行なう場合，多地点における非同期分散処理 (処理の分割/委譲/同期) を言語レベルで柔軟に記述できないという問題があり，加えて，実行状態の移送に基づきネットワーク上に分散した計算機に処理を委譲する仕組みが存在しない．

具体的には，(P1) 処理の分割がスレッド単位であり，スレッド内処理を具元化し，移送する手段がなく，細粒度な分散処理ができない．(P2)1つのスレッドから，複数の道筋を持った処理を複数のホストに委譲できない．(P3) 移送とその同期が同期的であり，またインスタンス変数，ローカル変数等の状態の同期が行なわれない．という3つの問題点が挙げられる．

本研究では，これら3つの問題を解決するために，Java

に MPCs(Migratable Partial Continuations) 機能を導入した Java with MPCs を提案し, 試作システムを設計, 実装する. MPCs 機能は, 関数型言語 Scheme において研究されている部分継続の概念 [1][2] を Java 言語に導入するものであり, 分散環境間をまたいだ遠隔生成機能の事を指す.

2 部分継続

本章では, MPCs 機能のための予備知識として Scheme における継続, 及び部分継続の概念について述べる.

2.1 継続

継続 (continuation) とは「ある時点以降の残りの計算」を抽象化した概念である. 歴史的には手続き的プログラムの表示的意味を定義するために導入され, Lisp の一方言である関数型言語 Scheme においては, プログラム中で継続をデータとして陽に扱う事ができ, これにより繰り返し, ジャンプ, 非局所脱出, コルーチン, マルチタスクといった様々な制御構造を実現している.

例えば S 式(+ 1 (* 2 3)) における部分式(* 2 3) の評価時には, この部分式の評価が終れば, 結果に 1 を足すという継続が存在する. このように, 継続はプログラムの実行を制御するのに欠かせない概念であり, あらゆるプログラミング言語にこの概念は存在する. Scheme では継続の生成に call-with-current-continuation(以後, call/cc と表記) を用いる事によって, 任意の時点における継続を得る事ができる. 例えば,

```
(* 2
  (+ 1 (call/cc (lambda (k) (+ 3 (k 4))))))
```

という式を評価すると, 部分式 (call/cc ···) の継続を引数としてラムダ式が呼び出される. 結果として (+ 3 (k 4)) の部分式 (k 4) が評価された時点で継続が呼び出され, 式全体の評価値は 10 となる.

以上のように, Scheme における継続は関数として具現化される. このように言語の一級 (first-class) データとして表現された継続を, 一級継続 (first-class continuation) と呼ぶ.

2.2 部分継続

継続が, ある時点から残りすべての計算を抽象化したものであるのに対し, 部分継続 (partial continuation) は, ある決まった場所までの計算を抽象化した概念である.

部分継続のための言語機構としては, splitter[2], marker-call/pc[1] などが提案されている. 例えば,

```
(* 2 (marker
   (+ 1 (call/pc (lambda (k) (+ 3 (k 4)))))))
```

という式を評価すると, marker から部分式 (call/pc ···) までの部分継続を引数としてラムダ式が呼び出される. 結果として, (+ 3 (k 4)) の部分式 (k 4) が評価された時点で, 部分継続が呼び出され, 式全体の評価値は 10 となる. 継続との違いは, 部分継続は呼び出したコンテキストに制御が戻るという点である.

Scheme の call/cc の一般的な実装 [3] のように, 継続を無限エクステントを持った一級データとして扱うためには, 任意の時点における制御スタックのコピー (あるいは, それに相当する情報) を保存し, 呼び出し時に制御スタックに戻す必要がある. これは, 一般的に負荷の高い処理である. 部分継続の実装上の利点としては, コピーするスタックを指定した範囲までに抑えられるということが挙げられる.

3 Java with MPCs

3.1 Java with MPCs の提案

既存のスレッド移送技術を活用し, 実行状態の移送による効率的な分散処理を実現するために, Java に MPCs 機能を導入する. そのために本研究では, Java のスレッド移送技術における移送単位の細粒度化, および記述性の向上を実現する手法として, Java with MPCs を提案する.

Java with MPCs は, 前述した既存のスレッド移送技術の問題点を解決するために, (S1) 計算機資源の場所指定, 処理の部分指定を記述する構文, (S2)MPCs の移送/同期を記述する構文, (S3)MPCs のインクリメンタルな捕捉と移送, を Java 言語に新たに導入する. (S1),(S2) については 3.2 節, (S3) は 3.3 節で述べる.

3.2 言語設計

MPCs 機能を実現するために, まず Java 言語に部分継続の概念を導入する. 部分継続は「ある時点からある時点までの残りの計算」を表すものであり, 処理の一部分を表す. また, 部分継続は, ある時点における「結果待ちの計算」と「未実行計算」に具元化できる. すなわち MPCs 機能を Java に導入するためには, 図 1 で示す 3 つの機能を言語レベルで表現する必要があり, これらは具体的に次の 3 つの項目に対応する.

1. 未実行計算の移送
2. 結果待ち計算の範囲の指定
3. 実行結果との同期

そこで本研究では, 上述の 3 項目を表現するために以下の構文を導入する.

1. **scatter 文**
2. **marker 文**

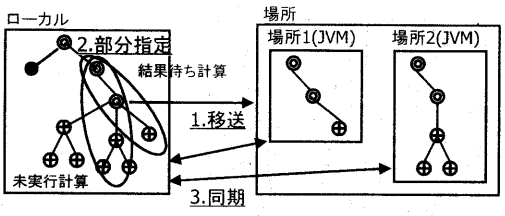

図 1: MPCs 機能のために必要な機能

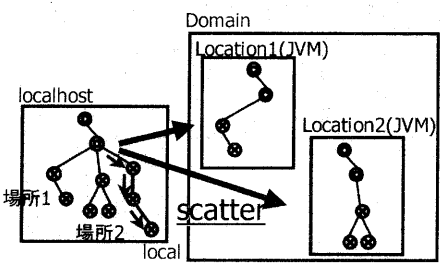

図 2: scatter 文

3. sync 文

以降の節で, それぞれの構文の詳細を述べる.

3.2.1 場所に関する設計

Java with MPCs では, MPCs の移送先を表すためにロケーションとドメインという二つの場所を導入する. ロケーションと MPCs の実行に対応した Java 仮想計算機の場所を表し, ドメインは残りの計算を行う権利を持つロケーションの集合を表す. また, ドメインに属するロケーションは結果待ちの計算を互いに共有する.

3.2.2 scatter 文

本研究では, 複数ホストへの部分継続の分散を実現するための構文として, scatter 文を導入する. なお, scatter 文では, 一級部分継続は実現せず, 『ばらまく』という動作をプリミティブに与える. scatter 文は, place, default, local の三つのラベルを持ち, 引数としてとるドメインに属するロケーション毎に継続を生成し, 呼び出しを行なう.

具体的には図 2 に示すように, scatter が呼び出された時点で, 複数の異なる「残りの計算」を並列にドメイン内の複数の計算機へ分散させる.

place ラベルは場所を指定し, 合致する場所にそれぞれが持つ式で表される未実行計算を持つ部分継続を生成する. default ラベルにより, ドメイン内に属するロケーションが示す JVM 全体に対して同じ部分継続を生成することが可能である. 加えて, ドメイン内で遊休状態にある計算機資源に期待し, タスクをドメイン全体に渡すという記述も可能である. また, local ラベルはローカルホストで実行する部分継続を生成する. これにより, 複数回 scatter で分散させることができる.

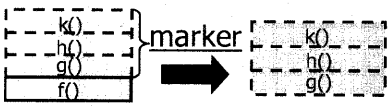

図 3: marker 文

3.2.3 marker 文

marker 文は, 処理の一部分を明示的にマークするものであり, 結果待ち計算の範囲, すなわち実行コンテキストの範囲を指定するものである. 概念的には, 図 3 に示す通り, 移送するフレームをマークするものである. また, marker は多段に用いる事が可能であり, 移送する保存スタックをプログラマが自由に指定することができる.

3.2.4 sync 文

sync 文は, 現在のローカル変数, インスタンス変数, クラス変数を含む実行状態の同期を指示するための構文である. sync 文は, 引数で指定されたロケーションにおいて, 以前に呼び出した部分継続を同期するために用いる. なお,sync 文により同期をとらない場合は, 部分継続の制御は戻らない. また, 複数計算機で実行された部分継続を指定して同期をとることが可能である.

また, sync 文は引数でドメインを指定することができ, その場合に同期されるロケーションは, ドメインに属するロケーション (サブドメインを含む) の中で最も早く計算が終了したロケーションとなる.

図 4, に追加した構文の記述例を示す.

3.2.5 マルチレベル分散と投機分散

scatter 文と place,default,local ラベル, 多段に指定可能な marker 文により, 部分継続を柔軟に指定する事ができ, かつ, それらを並列にドメイン内のロケーションへマルチレベルに分散させる事ができる. また, local ラベルにより処理を分散させ, 結果が必要な時のみ sync 文で同期をとるという投機分散も可能となる. 図 5 に示すように, Java with MPCs で追加した構文を用いる事で, プログラマは柔軟に様々な分散処理を記述, 実現できる.

3.3 ソースコード変換

既存の Java 言語には, 継続を扱う機能は存在しない. 継続は「プログラムの残りの計算」を表すものであり,

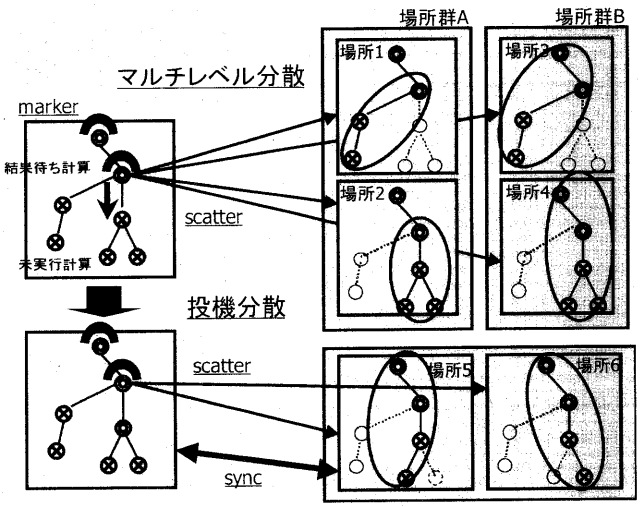

図 5: マルチレベル分散と投機分散

```
public void markMethod(){
  marker(dom){
    scatterMethod();
  }
  sync dom;
}
public void scatterMethod(){
  scatter(dom){
  local : speculativeMethod("a");
  place B : speculativeMethod("b");
  place C : speculativeMethod("c");
  }
}
public void speculativeMethod(String x){}
```

図 4: サンプルコード

結果待ち計算と未実行計算で具元化される. Javaにおいて, これらはある時点における実行コンテキストとバイトコードの残りの部分に対応するため, Java with MPCsを実現するためには, それらの情報が必要となる.

実行コンテキストの保存

実行コンテキストを保存するために, 各スタックフレーム毎にその環境を保存するクラスをソースコード変換により静的に生成する. 具体的には, 次のようなクラスを生成する.

例)

```
class StateTesttest
  int pc; // プログラムカウンタ
  int n; // ローカル変数
  int tmp; // テンポラリに使用する変数
           // (オペランドスタック上の変数)
```

実行コンテキスト全体の保存については後述する.

未実行コード

ソースコード変換時に, 実行コンテキストの保存と同様に, 未実行のコードは, 移送元で実行される保存用メソッドと, 移送先で実行される復帰用メソッドを生成し, 残りの計算を表現する.

3.3.1 Eager CMR

次に, 実行コンテキスト保存用のスタックフレーム毎のクラスを生成し, 実行コンテキスト全体を保存する手法について述べる.

一般に, 継続実行の機能は, 処理系に実装される [3]. しかし, JVMはそのセキュリティ機構により, バイトコードからの実行コンテキストへのアクセス, 操作をサポートしていない. また, コスト的判断により, 一般的に, 実行コンテキストの捕捉が起きない時の実行効率と空間使用量を劣化させないよう実装される. そこで, JavaGo[4] などでは, 例外処理機能を用いてスレッド移送を実現している. これをここではLazy CM(Capturing

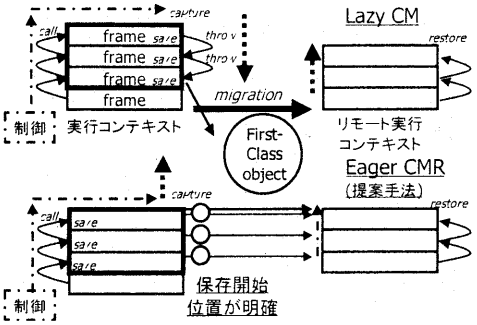

図 6: Lazy CM と Eager CMR

and Migration) と呼ぶ.

しかし, 部分継続の場合, 捕捉するスタックフレームの範囲が明示的に指定されるため, 実行コンテキストの保存を, スタックに積む毎にインクリメンタルに行なう事が可能である. 本研究では、このようなインクリメンタルな実行状態の保存を Eager CMR(Capturing and Migration and Restore) と呼ぶ. Eager CMR の利点としては, ローカル環境に捕捉が起きた時の実行状態を持った部分継続を生成できることがあげられる. これにより, 遠隔ホストに処理を投機分散し後で同期することや, 複数段階に scatter を呼び出すことが可能となる. Lazy CM と Eager CMR を模式的に示した図を図 6 に示す.

3.4 Java with MPCs システムの試作

Java with MPCs システムは, MPCs 機能を実現するための Java のクラスライブラリと, 拡張コードから Java コードを生成するコンパイラからなる. 実装は Java 言語により行ない, コンパイラは Java 用プリプロセッサ構築フレームワークである EPP[5] を用いて実装した.

4 関連研究との比較

分散処理への利用という観点で, 既存のスレッド移送技術と本研究を比較検討する. 比較表を表 1 に示す. ①は提案する Java with MPCs で, ②はソースコード変換で例外処理を用いた JavaGo, ③は JIT コンパイラにより実現した MOBA である.

JavaGo は undock という拡張構文で部分継続を実現するが, その同期機構を持たず, 処理を委譲するというよりは, モバイルエージェントの強い移送のためのものである. JavaGo, MOBA は go メソッドにより計算の主体の移動を与えるが, これに対して, 提案手法では, scatter 文で異なる部分継続を複数計算機資源へ分散する. また提案手法では, sync 文により任意の状態に同期するか, またはドメインに対して同期を行ない, 計算終了順に同期するかをプログラマが選択可能である. さらに, インクリメンタルな保存により, ローカルホストにも部分継続を生成でき, これにより, 投機分散を実現できるという利点を持つ.

5 性能測定と考察

提案手法の有効性を示すために, 試作した Java with MPCs システムを用いていくつかの実験を行ない, 考察を述べる.

5.1 実験環境

実験環境は, 以下に示す通りである.

実験環境	マシン A	マシン B
アーキテクチャ	Sun Ultra60 UPA/PCI	
CPU	UltraSPARC-II 360MHz	
メモリ	640MB	
OS	SunOS5.8	SunOS5.7
Java	Sun Java2 SDK v1.3.1	
ネットワーク	100Base-T	

5.2 オーバーヘッドの測定

ネットワークを介してマシン A からマシン B へ MPCs を分散させ, その時のスレッド移送に伴うオーバーヘッドを, 既存手法である Lazy CM と提案手法である Eager CMR とで比較する. また, インクリメンタルに保存だけを行なう Eager Capture を用いた場合も併せて示す. 実験では, fibonacci プログラムにおいてフィボナッチ数列を計算する再帰関数 fib(n) で fib(35) を計算し, n が 0 になった時点でマシン A からマシン B へ MPCs を移送することとする. marker はスタックの底からマークすることとし, そのためスタックフレームは 35 個移送される. また, 一つのスタックフレームは, 数十バイト程度である. 測定結果を表 2 に示す. 実験結果より, インクリメンタルな実現によるスレッド移送のオーバーヘッドは, 既存技術と同程度であり, 実用上十分な実行速度が得られる事が確認された.

5.3 考察と応用例

本研究では, スレッド移送に基づく分散処理の単位として MPCs(部分継続) を用いる事で, Eager CMR を実現し, 実験より既存技術と同程度のオーバーヘッドで実行可能である事を確認した. また, Eager CMR により, マルチレベル分散や投機分散など様々な分散処理を記述, 実現可能となる.

Java with MPCs により, 単純な分散処理を容易に記述する事ができ, また, N クイーンやパズル問題のような OR 並列型探索問題などにも有効であり, 投機的並列実行を行なう分散インタプリタ, WWW 投機的情報検

表 1: 関連研究との比較

システム	実現手法	移送単位	移送方法	同期	実行コンテキスト捕捉方法
①Java with MPCs (本研究)	ソースコード変換	部分継続	scatter で複数分散	sync で同期任意に状態	Eager Capture
②JavaGo	ソースコード変換	スレッド or 部分継続	go で単一移送	呼び出しと同期 or 同期しない	Lazy Capture
③MOBA	JVM の拡張	スレッド	go で単一移送	呼び出しと同期	専用 JIT で捕捉

表 2: スレッド移送に伴うオーバーヘッド

測定 プログラム [msec]	逐次	Lazy CM	Eager Capturing	Eager CMR
fib(35)	6977	10479	10593	10526
分割移送のコスト	-	3502	3616	3549

索などを記述, 実現するのにも有効であると考えられる.

6 むすび

6.1 まとめ

本論文では, 実行状態の移送に基づく Java 分散処理の実現を目的とし, 多地点での非同期分散処理を行なうために MPCs 機能を導入した Java with MPCs を提案し, その設計, 実装を行なった. また, 試作したシステムのオーバーヘッドを測定し, 応用例を論じ, その有効性を検証した.

6.2 課題

セキュリティに関する課題

現在の提案システムでは, MPCs 移送のためのセキュリティ機構を RMI に依存している. しかし, 実用を考えた場合, Ninflet[6] や Java/LR[7] のような独自のセキュリティ機構を備える必要があると考えられる.

動的負荷分散に関する課題

本来, 実行状態の移送は, 動的負荷分散という点で他の分散モデルに対して優位性を持つが, Java の持つセキュリティ機構により, バイトコードからの実行コンテキストへアクセス, 操作はおろか, あるスレッドの実行中における他のスレッドの安全な停止などに課題がある. . そこで, javassist[8] のようなバイトコード編集ツールによるバイトコード変換と拡張クラスローダによる動的コード変換による, 動的負荷分散の実現なども今後検討したい.

参考文献

[1] Moreau, L. and Queinnec, C. *Partial Continuations as the Difference of Continuations - A Duumvirate of Control Operators,* Lecture Notes in Computer Science, Vol.844, SpringerVerlag,pp.182-197(1994)

[2] Queinnec, C. and Serpette, B. *A Dynamic Extent Control Operator for Partial Continuations,* Proc. Eighteenth Annual ACM SIGACTSIGPLAN Symposium on Principles of Programming Languages, pp.174-184(1991)

[3] Clinger, W., Hartheimer, A.H. and Ost, E.M. *Implementation Strategies for Continuations,* Proc. 1988 ACM Conference on Lisp and Functional Programming, pp.124-131(1988)

[4] T. Sekiguchi and A. Yonezawa. *A calculus with code mobility* IN FMOODS '97(1997)

[5] 一杉裕志. 高いモジュラリティと拡張性を持つ構文解析器. 情報処理学会論文誌: Vol39, No.SIG1(PRO 1), pp.61-69(1998)

[6] 高木 浩光, 松岡 聡, 中田 秀基, 関口 智嗣, 佐藤 光久, 長嶋 雲兵. *Java* による大域的並列計算環境 *Ninflet*(特集:並列処理), 情報処理学会論文誌: Vol40, No.05(1999)

[7] 渡辺 昌寛, 伊藤 貴康. 場所の概念を備えた *Java* 言語とその処理系. 情報処理学会論文誌: プログラミング, Vol40, No.SIG07(1999)

[8] Shigeru Chiba, *Load-time Structural Reflection in Java.* In Proceeding of ECOOP 2000, LNCS 1850, Springer Verlag,pp.313-336,2000