並列処理と例外処理を統一的に扱う構造化言語

[1]島田 雄介　[2]八杉 昌宏　[2]瀧 和男

[1]神戸大学大学院自然科学研究科
[2]神戸大学工学部情報知能工学科

本論文では，オブジェクト指向並列言語 OPA の並列処理と例外処理について述べる．OPA では，並列処理を fork-join 型に構造化するため，同期などに誤りが入りにくく，不規則な並列性を含む問題を容易に記述することができる．例外処理については，fork-join ブロック内の並列実行されるスレッドで発生した例外を join 先で受け止めて処理するように記述できる．また，例外が発生したスレッドだけでなく，fork-join ブロック内の他のスレッドをすべて暗黙的に終了させている．このため，この例外処理を利用することで，探索問題などを効率良く実行させることできる．

A Structured Language for Parallel Processing and Exception Handling

[1]Yusuke SHIMADA　[2]Masahiro YASUGI　[2]Kazuo TAKI

[1]The Graduate School of Science and Technology, Kobe University
[2]Department of Computer and Systems Engineering Faculty of Engineering, Kobe University

In this paper we present parallel processing and exception handling in an object-oriented language OPA. In OPA, parallel processing is represented by fork-join structure; therefore,we can easily describe irregular parallel problems. An exception which is raised in fork-join parallel processing can be handled at join destination. In addition, not only the thread in which an exception is raised but also other threads which are executed in parallel are terminated implicitlly. A search problem can be executed efficiently by this exception handling mechanism.

1　はじめに

我々は現在，オブジェクト指向並列言語 OPA [1,2] の設計と実装を進めている．OPA では，分散メモリ型，共有メモリ型を問わず効率良くプログラム実行できるような処理系の実装を可能とするとともに，不規則な並列性を含む問題を容易に誤りなく記述できることを目指している．

本稿では，OPA の並列処理と例外処理について述べる．信頼性が高く構造化されたプログラムを作成するためには，通常とは異なる例外的な状況の発生を検知し，それに対処するための処理を記述する必要がある．このためユーザがこの例外処理について容易に誤りなく記述できるように，言語側で支援することが必要とされている．

以下，2 章でオブジェクト指向並列言語 OPA について述べる．3 章で OPA での例外処理について述べ，4 章で実装方法について述べる．

2　オブジェクト指向並列言語 OPA

OPA は Java [3] ライクなシンタックスを用いたオブジェクト指向並列言語である．メモリ番地としてのポインタは許さず，関数呼び出しの引数は値渡しであるが，オブジェクトや配列への参照が値として利用できる．

2.1　従来の並列処理

並列処理は，均質で規則的な問題を対象とするデータ並列と，均質さが小さく不規則な問題を対象とする非データ並列に分けることができる．

データ並列の問題に対しては長年研究されており，HPF など様々なデータ並列用の言語が開発されている．こういった言語では，ある解くべき問題に含まれる規則的な問題（行列の演算など）に対して，並列に処理することができ，forall 文などで容易にそれを記述できる．また，同期が暗黙にとられるため，そこに誤りが入りにくいものとなっている．しかし，不規則な並列性を含むような問題（非データ並列）に対しては，記述が困難となり，容易に並列処理できないといった問題がある．

一方，非データ並列の問題を記述するための言語として，ABCL/1 [4] など並列オブジェクトモデルに基づく言語が研究されている．この言語では，問題に内在する並列性を自然に引き出すため，不規則な並列性を含む問題を容易に記述できる．しかし，逆にユーザは実行時の全体としての処理の進行状況について把握しにくく，また，処理完了時の同期に関する誤りが容易に記述できてしまう問題点があった．

このように，従来の並列処理では，データ並列と非データ並列のどちらかを対象として言語を設計しているためそれぞれに欠点があった．このため，並列処理できる部分と逐次処理でなければならない部分が含まれる問題は，容易に記述できなかった．

2.2　OPA の並列処理

実行開始時は，逐次で処理されなければならないが，ある時点で並列に処理できる部分問題をもつような問題がある．また，その部分問題を処理した後，その結果を用いて，逐次で処理され続けるようなこともある．

前節で述べた言語では，このような問題の記述は容易ではない．OPA では，この問題に対して，fork-join を用いて容易に誤りなく記述できるようにする．

問題を処理している時，ある部分で並列に実行できる部分問題が複数ある場合は，必要個数のスレッドを生成 (fork) し，それに部分問題を解かせることで，並列処理を行なう．新たに生成されたスレッドも，さらにスレッドを生成することで，与えられた部分問題をより並列に処理でき．入れ子構造で並列処理を記述できる．fork-join ブロック内で生成されたスレッドは，ブロックの終了地点で同期 (join) をとる．(図 1)．

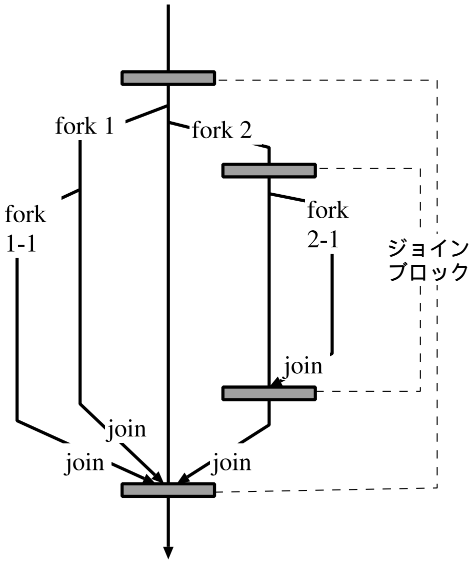

図 1: fork-join 処理

スレッドの fork と join には，キーワード par，join を用いる．fork させたいメソッド呼び出しに修飾子として par を付加すると，新たにスレッドが生成され，そのメソッドを新しいスレッドで実行することができる．また，join ブロック内で fork されたスレッドは，その join ブロック内で終了するため，処理が終了した後もスレッドが生き残ることはない．

```
join{
      par obj1.m();     // 並列実行
      par obj2.m();     // 並列実行
         ..
}                       // ここで同期
```

2.3　メソッド分類

オブジェクトのメソッドを，読み出し専用メソッド (RO) と読み書き両用メソッド (RW) に分けることで，オブジェクトの排他制御を少なくでき，処理の高速化を図ることができる．そこで，OPA では，メソッドの定義にキーワード instant を付加することで，文面上の読み書きに応じて，メソッドを RO メソッドと RW メソッドに分類する．

RO メソッドは，一貫性のあるオブジェクトの状態を読み出すことができれば，読み出した後のオブジェクトの排他制御は必要ない．RW メソッドについては，インスタンス変数の更新を，変数への書き込みがある度に行なうのではなく，メソッドのある特定の点で一括更新することによって，排他制御区間を最小限に抑えることができる．

2.4　多重状態

前節で述べた RO メソッドと RW メソッドといった instant メソッドの処理のために，それらが起動されるたびに，そのメソッド用にオブジェクトの状態の複製が作成される．これによって一つのオブジェクトが複数個の状態を同時に持つ多重状態が存在する．この状態を図 2に示す．

instant メソッドが起動されると，インスタンス変数などのオブジェクトのデータは，元となるオブジェクト (common state) からメソッド実行のための作業領域 (working state) に一括して読み込まれる．以後，instant メソッドは working state のデータに対して読み書きを行ない，RW メソッド実行の一括更新によって，working state から common state へデータが書き戻される．一括更新する前の RW メソッドの working state は，同時に一つしか存在を許していないため，複数の RW メソッド実行要求は逐次に処理される．RW メソッドと RO メソッドはデータの一括更新中を除くと並列に処理される．また，あるオブジェクトの instant メソッドを実行中に同じオブジェクトの RO メソッドを呼び出した場合は，手元の working state のデータを用いず，再度 common state からデータの読み込みを行なう．

通常スレッドがオブジェクトを占有しているのは，RW メソッド実行の一括更新前の間だけである．このため，一括更新後も占有し続けていたい時は，synchronized を用いる．この synchronized についても多重状態を用いて実現する．synchronized の処理開始時に，RW メソッドの時同様，common state から SYNC working にコピーし，コピーされたデータに対して，読み書きを行なう．この間，synchronized の処理を行なっているスレッドは図の右側を見ており，他のスレッドは左側を見るものとする．但し，オブジェクトの一貫性を保つため，synchronized 状態のオブジェクトが存在している間は，そのオブジェクトに対しての RW メソッドの実行を許さない．データの書き戻しについては，synchronized の処理終了時に行なわれる．

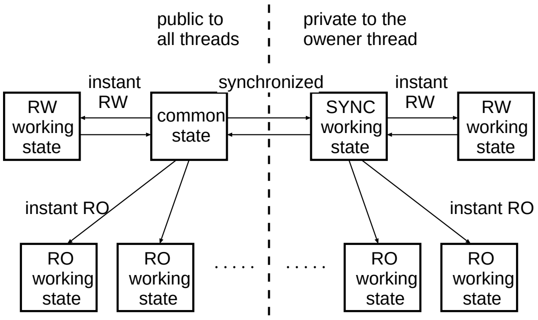

図 2: 多重状態

3　例外処理

3.1　設計方針

OPA の並列処理は，ある大きな問題に含まれる並列処理可能な部分問題を記述しやすく設計されている．この部分問題において，並列に実行されているスレッドは，それぞれ同じ目的を持って実行されていることが多い．このため，これらのスレッドについて同期を容易にとることができるよう，OPA では fork-join ブロックを用いている．

例外処理についても同様に考えると，各スレッドごとに例外が発生した時の処理を記述するよりは，並列に部分問題を解く上で発生した例外を一括して処理するように記述できた方が自然である．そこで，OPA の例外処理は，fork-join ブロック内の並列処理で発生する例外を join 先で受け止めることができるようにする．

3.2 文法

OPA での例外処理の文法は，基本的には Java と同じである．例外が発生する範囲の指定と，例外処理の記述には try-catch ブロックを用いており，try ブロックの内で発生した例外を catch で受け止め，そのときの動作を catch 節に記述する．

```
try {
  ...
  join{
   // スレッド生成・実行
     par obj1.method();
     par obj2.method();
  }
     ...
}catch(ExceptionName1 expObj){
// 例外 1 発生時実行
  ...
  throw expObj;  // 例外を外へ投げる
}catch(ExceptionName2 expObj){
// 例外 2 発生時実行
  ...
}finally{
// 必ず実行
  ...
}
```

try ブロックの中で発生する複数の種類の例外を個別に処理するときは，個々の例外に対する catch 節をそれぞれ記述する．発生した例外をより外側の catch 節に伝播させるときは throw 文を用いる．例外発生の有無に関わらずに必ず実行したい処理があれば finally 節に記述する．また，try-catch ブロックは入れ子に書くことが可能である．

一つのスレッド内で解決する例外処理については，java の例外処理と同様に考える．並列処理での例外について，OPA では，join ブロックを try ブロックの中に記述でき，その join ブロック内のスレッドが投げた例外を catch 節で受け止めることができる．

3.3 動作

あるスレッドにおいて例外が発生した場合，そのスレッド内で例外処理が完了する時は，java の例外処理と同様に動作する．最も内側の catch 節で例外を受け止めて処理し，そこから外側の try-catch 節に例外を投げることもできる．メソッド内の catch 節で受け止めることができない時は，メソッド呼出側に例外が伝播していく．

例外が発生したスレッドで例外処理が完了しない時は，スレッドの join 先に例外が投げられる．そして，その join ブロックを囲んでいる最も内側で定義された catch 節で例外を受け止める．この時，例外を受け止めた try-catch ブロックに含まれる fork-join ブロック内のスレッドは，すべて強制終了させる (図 3)．

強制終了されるスレッドは，終了前に finally 節を実行し，終了時にそのスレッドが確保していたオブジェクトをすべて解放する．このとき，一括更新を行なっていないオブジェクトに対しては，更新を行なわない．しかし，例外が発生したスレッドの場合は，ユーザが throwing を伴う return 文を明示することで更新させることができる．

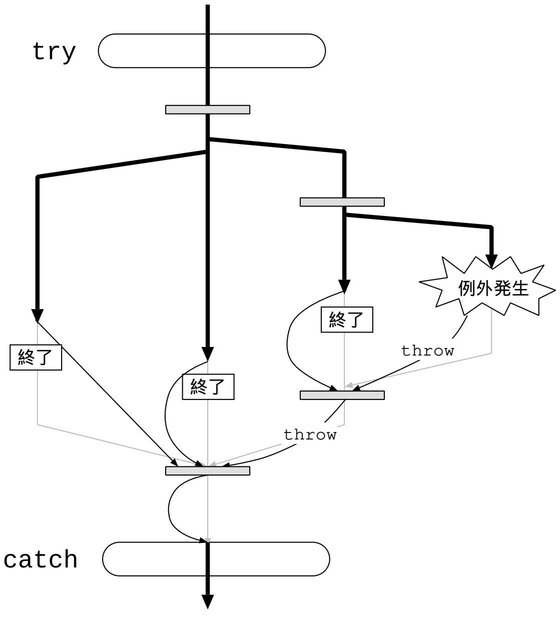

図 3: 例外処理

3.4 応用例

複数解をもつ問題を解く時，全解ではなく一つもしくはいくつかの解が分かれば良いことがある．この時，複数のスレッドが並列に解を探索する方法が一般的であり，この探索終了条件は必要個数の解を見つけた時となる．しかし，従来の言語では並列探索を行なっているすべてのスレッドの実行停止といっ

た制御の記述が必要であるためプログラミングが複雑になる．

前節で述べたように，OPA の例外処理を用いると，並列実行しているスレッドの終了を容易に行なうことができる．このため，例外処理を用いて，この種の探索問題を，効率良く実行できるプログラムを記述することができる．

以下に解を二つ探索する問題の例を示す．

```
// メインクラス
//  （例外を受け止める）
public class SearchStart {
   static void main(String argv[]) {
     try{
       join{
         Search node1 = new Search(),
                node2 = new Search();
         Table table = new Table();

        // 並列探索開始
         par node1.calc(table, 1);
         par node2.calc(table, 2);
       }
       System.out.println("UnFounded");
     }catch(Founded excp){   // 例外処理
       System.out.println("Founded");
     }
   }
}

// 解探索を行なうクラス
class Search {
   void calc(Table table, int p)
        throws Founded {
      ...
     if (解発見)
        table.add(answer);
     else
        join {
          Search node1 = new Search(),
                 node2 = new Search();
          par node1.calc(table, 2*p+1);
          par node2.calc(table, 2*p+2);
        };
}

// 解を保存するクラス
//  （例外が発生する）
class Table {
   int answer1, answer2, num = 0;
   instant void add(int ans)
            throws Founded {
     if (num == 0) {
        answer1 = ans;
        num++;
     }else{
        answer2 = ans;
        num++;
     // 例外を発生させる
     //   この時一括更新も明示して行なう
        return throwing new Founded();
     }
   }
}
```

3.5 関連研究

3.5.1 Java

Java の例外処理は，基本的に例外が発生したスレッド内でその例外を解決するように設計されている．例外処理の記述には try-catch ブロックを用いている．

OPA は，並列の記述以外は Java との互換性をできるだけ保つようにしているため，例外処理の記述法は Java と同じ try-catch ブロックを用いており，スレッド内での例外処理は，Java の例外処理と同様に扱われる．しかし OPA ではさらに，スレッドで発生した例外を join 先に投げて fork-join ブロック内のスレッドを終了させることができる．

3.5.2 ABCL/1

ABCL/1 の例外処理 [5] は，例外メッセージに対する処理として定義する．システムは，例外発生時に例外メッセージを作成しオブジェクトに送信する．例外メッセージの送信先には，例外が発生したメソッドのタイプや引数に応じて，そのメソッドを呼び出したオブジェクトや返答メッセージを受け取るオブジェクトが指定される．

ABCL/1 はアクタをベースとした並列オブジェクト指向言語であるため，例外処理は，例外発生時のオブジェクトの振舞いを個々に定義することで対処している．それに対し，OPA はスレッドをベースとしているため，例外処理は，try-catch で記述できる．

3.5.3 KL1

KL1 は並列論理型言語であり，例外処理には荘園 [6] を用いている．荘園とは，ある複数のプロセスのグループのことであり，グループはあるプロセスとそのサブプロセスすべてで形成されている．また，荘園は入れ子構造にできる．プロセスが発生させた例外は，プロセス単位でなく，そのプロセスが属する荘園単位で扱われる．荘園の内外との通信のために，報告ストリームと制御ストリームが用意されており，これを用いて荘園内部で発生した例外を外部に伝えることができ，また，荘園内部に制御命令を送ることができる．

KL1 と OPA では，並列処理における例外処理の考え方は似ている．しかし KL1 の荘園は，入出力をもつ独立したプロセスとみることができ，そのプロセスの例外を扱うと考えることができる．一方 OPA では，ある大きな処理全体の中の一部の処理中で発生する例外を扱うと考えている．

4 実装方法

4.1 スレッド停止のタイミング

あるスレッドは他のスレッドが起こした例外によって，強制終了しなければならないことがある．このため，スレッドは実行中に例外発生のフラグが立っていないかチェックする必要がある．このチェックを一命令実行ごとに行なえば，フラグが立つとすぐに終了することができるが，実行速度は大きく低下すると考えられる．そこで今回の実装では，自スレッドで例外が発生した時と yield() が実行された時にフラグのチェックを行なう．

4.2 スレッド停止方法

OPA では，fork-join ブロックは join frame を用いて実装しており，これを利用して各スレッドを終了させる．

join frame は fork-join ブロック開始時に，スレッドの join の完了を検出するために作られ，fork-join ブロック内で生成されたスレッドはこの join frame へのポインタを持つ．そこで，join 先に投げられる例外が発生した場合，そのスレッドの join 先の join frame に，例外が発生したことを示すフラグを立てるようにする．fork-join 内のスレッドは前節で述べたタイミングでこのフラグをチェックすることで，スレッドを終了させることができる．例外処理は，スレッドが終了した後に行なう．

5 結論と今後の課題

本論文では，オブジェクト指向並列言語 OPA の並列処理と例外処理について述べた．OPA では，fork-join 型で構造化された並列処理中に発生した例外を join 先で受け止めることができ，join ブロック内のすべてのスレッドを終了させることができる．今後は，例外処理の実装を進めていく予定である．

参考文献

[1] 八杉昌宏, 瀧和男. 並列処理のためのオブジェクト指向言語 OPA の設計と実装. 情報処理学会研究報告, Vol. 96, No. 82, pp. 157–162, 1996.

[2] 八杉昌宏, 瀧和男. 実用的な並列処理のためのオブジェクト指向言語 OPA の設計. 第 13 回オブジェクト指向計算ワークショップ (WOOC'97), Mar 1997.

[3] K. Arnold and J. Gosling, editors. The Java Programming Language. Addison-Wesley Publishing Company, 1996.

[4] A. Yonezawa, editor. *ABCL:An Object-Oriented Concurrent System - Theory, Language, Programming, Implementation and Application.* MIT Press, 1990.

[5] Yuuji Ichisugi and Akinori Yonezawa. Exception Handling and Real Time Features in an Object-Oriented Concurrent Language. In *Concurrency: Theory, Languages and Architecture*, Vol. 491 of *LNCS*, pp. 92–109. Springer-Verlag, 1990.

[6] 瀧和男（編）. 第五世代コンピュータの並列処理. 共立出版, 1993.