プログラミング-言語・基礎・実践- 8－11
(1992. 8. 20)

分散型関数型言語Cmexの構成とその応用について

廣田 周吾† 大久保 英嗣† 大野 豊† 白川 洋充††

† 立命館大学理工学部情報工学科

†† 近畿大学理工学部経営工学科

遅延評価型の関数型言語では，ユーザが引数を評価するタイミングを見い出すことができない．関数型言語を分散環境で実現するには，未評価の引数を他のノードに渡たすことになり，通信コストの増加によって並行性が損なわれることがある．この問題を解決するために分散型関数型言語Cmexでは，プロセスの概念を導入し，引数の先行評価を行っている．Cmexは，部分式の先行評価のみならず，プロセスを動的に生成可能であり，リストと等価なプロセス間通信のためのポートを用意している．本論文では，Cmexの構成と，並行／並列パラダイムの中でも重要なパイプライン処理への適用として並行B木探索の処理方式を示す．

A Distributed Functional Programming Language Cmex and Its Application to Concurrent B-tree Search

Shugo Hirota† Eiji Okubo† Yutaka Ohno† Hiromitsu Shirakawa††

† Department of Computer Science and Systems Engineering,
Faculty of Science and Engineering, Ritsumeikan University
56-1 Tojiin Kita-machi, Kita-ku, Kyoto 603, Japan

†† Department of Industrial Engineering,
Faculty of Science and Engineering, Kinki University
3-4-1 Kowakae, Higashi-Osaka, 577, Japan

The arguements of lazy functional language are evaluated only when they are truly needed. Sometimes this leads to the inefficiency in concurrency when the functional programming language is implemented in distributed environments. This is due to the fact that the unevaluated arguments may cause high communication costs for passing them to other nodes. Proposed distributed functional language Cmex is intended to be used in distributed environments. User can preset lazy evaluation, thus the arguments can be evaluated before passing to other nodes. This plays a crucial role in implementing pipeline processing or other efficient concurrent processing. As an application Cmex is used to describe pipelined concurrent B-tree search.

1 はじめに

一般に，関数型言語では，式の評価は遅延評価を基本としてなされる．即ち，式の評価は，その値が必要となるまで行われない．一方，関数型言語には副作用がなく，評価の結果はその評価順序には関係がないという性質を持っているので，マルチプロセッサに拡張した場合にも評価の結果は正しく求まる[1, 2]．しかし，大部分の遅延評価型の関数型言語では，ユーザが引数を評価するタイミングを見い出すことができない．従って，分散環境で関数型言語を実現するには，未評価の引数を他のノードに渡たすことになり，通信コストの増加を招き並行性を劣化させるという事態を招いている．この問題を解決するために，われわれはプロセスの概念を導入し，引数を先行評価する分散型関数型言語 Cmex(Concurrent Manipulation of Expressions) を開発している．Cmex は，次のような機能を有している．

- preset 句による部分式の先行評価が可能である．
- 評価の過程で，動的にプロセスを生成することが可能である．
- リストと同様に扱えるプロセス間通信を行うためのポートを用意している．

分散環境でプロセス生成が可能な関数型言語に Concurrent Clean [3] がある．Concurrent Clean では，プロセスは，通信チャンネル (communication channels) によって親ノード (root node) につながれたサブグラフとして扱われる．プロセス間の通信は，親プロセスと子プロセス間のみで行われ，子プロセス同士の通信を行うことはできない．従って，Concurrent Clean は多数のプロセスが通信し合う協調処理のパラダイムには適していない．これに対して，Cmex ではプロセス間通信の際にポートを用いることにより，親プロセスと子プロセス間の通信以外に子プロセス同士でも自由に通信を行うことができる．このポートは，Cmex において唯一副作用が存在する．しかしながら，これら Cmex のプロセスとポートの機能を用いることにより，より広い並行／並列処理のパラダイムへの適用が可能になる．その一つとして，並行／並列性のパラダイムの中でも重要なパイプライン処理が可能になる．本論文では，Cmex の構成を述べ，その適用例として，パイプライン処理を用いた B 木の並行探索を示し，Cmex の有用性を述べる．

2 分散型関数型言語 Cmex

分散型関数型言語 Cmex は，Miranda をベースとして，部分式の先行評価，プロセスの概念，プロセス間通信のためのポートを導入し，並行／並列性を記述できるように拡張した言語である．Cmex では，浮動小数点演算，UNIX に関連するコマンドを除いた Miranda の持つすべての機能をサポートしている．

2.1 基本機能

プログラムは関数の定義式の集合であり，スクリプトと呼ばれる．where 句により，関数の定義式中に局所的な定義を行うことができる．関数定義のスコープを示すために，オフサイドを用いたインデントルールを用いている．

ガード式とパターンマッチング

ガード式とパターンマッチングは，ともに場合分けを処理するためのものである．ガード式は “式, 条件” の集合であり，条件が満たされた時に式が評価される．一方，パターンマッチングは，関数の引数のパターンによって定義式を書き分けるものである．

ドット式と ZF 式

ドット式と ZF 式は，あるパターンを持ったリストを簡潔に記述するためのものである．ドット式は等差数列を要素とするリストを [初項, 公差.. 終項] として記述する．ZF 式は，リストの内包的表記法を用いて要素を表現する．

型

言語は，静的に強く型付けされている．つまり，スクリプト中のすべての関数や部分式の型は，コンパイル時に決定される．型に関する宣言がない場合には，Milnerの方法に基づいた型推論機構により型が推論される．また，多相型を扱うことができ，型変数*,**を用いて任意の型を表現できる．

型定義の方法には，次の三種類がある．

(1) 型類義 (type synonym)
すでに存在する型に別名を付ける．"==" を用いて定義する．

(2) 代数データ型 (algebraic data type)
構築子 (大文字から始まる識別子) を用いて新しい型を定義することができる．"::=" を用いて定義する．

(3) 抽象データ型 (abstract data type)
型と型に対する操作とを合わせて，一つの型としたものである．abstypeによって型を定義し，with 以下にその操作を行う関数の型を宣言する．

2.2 拡張機能

preset 句による先行評価

Cmexでは，式の評価に遅延評価を用いている．しかし，並行／並列処理を行う場合には，次のような問題点がある．

- プロセス間通信の際に，受渡しされるデータが未評価の式の場合には通信にかかるコストが大きくなる．
- 値が必要になるまで，それを評価するプロセスが生成されない．

従って，上記のような場合においては，すべての式を遅延評価するよりも，部分的に式を先行評価した方が望ましい結果が得られる場合がある．そこで，Cmexでは，部分的に式を先行評価できるようにpreset句を導入している．

preset句は，where句と同様に一つの関数定義における局所定義を行うためのものである．preset句内の式(preset式と呼ぶ)は，"識別子=式" の書式をとる．preset式の右側に先行評価が必要な部分式を記述し，左側の識別子を関数の定義式中に部分式の代わりに記述することができる．preset句中のすべての式は先行評価される．すなわち，関数の式の評価に先だって，preset句の上から順に評価されることになる．preset句の構文を以下に示す．

```
preset 句 := preset  preset 文+
preset 文 := 識別子 = 式
           | ポートの宣言
           | array 文
           | preset 句
```

プロセス

プロセスは，Cmexにおける並行／並列処理の単位である．これは，評価の過程で動的に生成することが可能である．プロセスを生成するためには，どの部分を並行／並列して実行させるのかを，明示的に指定する必要がある．このように，並行／並列性の記述をユーザに任せることにより，余分なプロセス生成を制御でき，効率的な処理を行えるようになっている．プロセス生成は，preset句中にプロセス生成子 {p} を付加することによって行う．

```
{p} ::  * -> *
識別子 = {p} 式
```

式の前に，プロセス生成子 {p} を付加することにより，その式の評価を行うプロセスを新たに生成する．評価の結果，正規形が得られたならば識別子に評価の結果を返し，プロセスはその処理を終え，消滅する．評価の結果が返ってくる前に，その識別子を評価しようとしたプロセスは，結果が返るまで封鎖される．

プロセス間通信

プロセス間通信を行うために，Cmex ではポートの機能が用意されている．通信は，送信側のプロセスがポートに値を出力し，受信側のプロセスが同じポートから値を入力することによって成立する．このポートも，評価の過程で動的に生成することが可能である．

Cmex では，ポートはリストと等価なものとして扱われる．すなわち，“ポートはそのポートが受け取った値のリストである” として扱われる．つまり，ポートに値を出力することは，ポートのリストの一番最後にその値を追加することであり，ポートから値を入力することは，ポートに出力された値のリストを受け取ることである．したがって，ポートから値を入力するプロセス側から見ると，ポートからの入力をリストと全く同じものとして扱うことができ，プロセス間通信を意識せずに記述することが可能である．

ポートの生成は，ポート宣言子 useport によって行う．

```
useport ポート名+
        [in {common | private | once}]
```

ポート宣言子 useport では，生成するポート名とそのポートの種類を指定する．ポートの種類としては以下のものがある．

(1) 共有ポート

共有ポートとは，そのポートに出力されている値をすべてのプロセスが共有するポートのことである．従って，あるポートから入力操作を行ったプロセスは，すべて同じ値のリストを得ることになる．

(2) 非共有ポート

非共有ポートでは，先に入力操作を行ったプロセスがポートの値を占有使用する．例えば，三つのプロセス A，B，C が，この順に非共有ポートに対して入力操作を行ったとすると，最初の値をプロセス A が，二番目の値をプロセス B が，三番目の値をプロセス C が受け取ることになる．

(3) 同期ポート

同期ポートとは，出力できる値が一つに限定されたポートのことである．プロセスが同期ポートに対して出力操作を行うと，そのポートに既に値が出力されている場合には，ポートに出力されず，既に出力されている値のリストが返る．値が出力されていない場合は，値をポートに出力し，その値が入力操作によって取り出されるまで送信側のプロセスは封鎖される．受信側によって値が取り出された後，送信側には [ ] が返る．入力操作を行うと，ポートに出力された値のリストが返る．ポートに値がない場合でも，受信側のプロセスは封鎖されず，[ ] が返る．

ポートはポート型を持つ．ポート型は port * で表され，* はそのポートに対して入出力される値の型を示す．一つのポートには一つの型の値のみ入出力を行うことができる．

ポートへの入出力操作を行うために次の二つの命令が用意されている．

(1) ポートに値を出力する

```
命令:  {>}
型   :  port * → * → [*]
記述:  {> ポート名} 値
```

ポート名で示されるポートに値を出力する．同期ポートは前述のようにポートの状態に応じたリストが，それ以外のポートは出力した値を要素として一つだけ持つリストが返る．すなわち，副作用を利用してポートに値を出力しており，これは，Cmex で唯一の副作用を持つ命令である．

(2) ポートの値のリストを得る

```
命令:  {<}
型   :  port * → [*]
記述:  {< ポート名}
```

ポート名で示されるポートが受け取る値をリストとして返す．

3 応用例 ～並行B木探索～

ここでは，Cmexのプロセスとポートを用いることにより，並行B木探索[4]のアルゴリズムをCmexで記述することを考える．

B木探索とは，平衡m分木探索の一種であり，データの探索，挿入，削除を最悪の場合でも一定時間内で行うことができる探索法である．このアルゴリズムは，データベースにおける辞書の並行操作に相当する．

B木は，次の制約条件を満たす．

(1) 根は葉であるか，または二つ以上の子供を持つ．

(2) 各節点(根と葉を除く)の子供の数をxとすると，$\lceil m/2 \rceil \leq x \leq m$である．

(3) 根から葉までの木の深さは，すべての葉に対して等しい．

ここで，データは葉だけにあり，節点はデータを探索する際に用いるキーのみしか持たない．さらに，データやキーは昇順で格納されている．

3.1 プロセス構成

並行B木探索を行うために，B木の節点のそれぞれに一つプロセスを割り当てる．各プロセスは隣接する上下左右の節点に位置するプロセスと通信することにより，データの探索，挿入，削除の処理を行う．プロセスは，挿入操作により新たに節点が必要になった時に生成され，削除操作により節点が必要なくなると消滅する．図1にプロセスの構成を示す．

各プロセスは，自節点への要求を受け取るポート，親節点，左右の節点に位置するプロセスと通信するためのポート，自節点の右側に節点がある場合には，その節点が持っている探索キーのうち最小の検索キーを持つ．また，プロセスのB木上での位置によって次のような情報も併せ持つ．

(a) **根**　$2 \sim m$個の子節点に位置するプロセスと通信するためのポートと，それと同数の探索のキーを持つ．

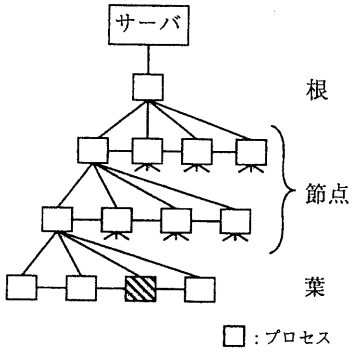

図1: プロセスの構成

(b) **節点**　$\lceil m/2 \rceil \sim m$個の子節点に位置するプロセスと通信するためのポートと，それと同数の探索のキーを持つ．

(c) **葉**　$\lceil m/2 \rceil \sim m$個のデータと，それと同数の探索のキー，処理の結果を返すためにサーバと通信するためのポートを持つ．

各プロセスは，探索，挿入，削除等の処理要求を受け付け，その要求に応じて処理を行い，その結果を子節点のプロセスに伝える．その際，必要ならば親節点，左右の節点のプロセスと通信を行う．処理の結果は，葉に位置するプロセスからサーバプロセスに直接返される．全体の処理の流れを図2に示す．

3.2 操作

B木に対しては，次に示すような探索，挿入，削除の三つの操作を並行に行うことができる．

探索操作

サーバプロセスが，根プロセスに探索すべきキーを送る．根プロセスは自分が持っているキーを用いて，そのキーに対応する次のレベルの節点を選択し，そのプロセスに探索キーを含む探索要

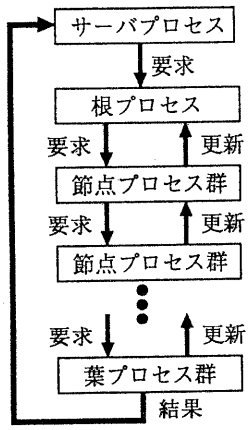

図 2: 処理の流れ

求を送る．節点プロセスは，まず，送られてきたキーと自節点が持つ最小キーと右節点が持つ最小キーとを用いて，次に選択すべき節点が自分の子節点であるかどうか調べる．自分の子節点でない場合には，送られてきたキーによって左右の節点に位置するプロセスに探索要求を送る．自分の子節点である場合にはそのプロセスに探索要求を送る．このようにするのは，挿入や削除操作によって加えられた更新は，葉から根に向かって波及し，かつ各操作は並行に行われるために，更新が親節点まで及んでおらず，親節点が間違った節点を選択した場合でも正しく探索が行われるようにするためである．

この探索を葉に達するまで繰り返す．葉においてキーに対応するデータが得られたら，それを葉プロセスがサーバプロセスに直接送る．

挿入操作

検索と同様の方法で，データを挿入すべき葉を見つけ出す．葉のデータの数が $m-1$ 個以下なら問題なく挿入できる．既に m 個のデータを持っている場合には，その葉の右横に新しく葉のプロセスを生成し，データを $\lfloor (m+1)/2 \rfloor$ 個と $\lceil (m+1)/2 \rceil$ 個に分割する．新たに生成した葉プロセスの挿入を，親節点に当るプロセスに挿入更新要求として送る．その親節点も m 個の子供を持っていたなら，その右横にプロセスを新たに生成し分割を行い，そのまた親節点に挿入更新要求を送る．これを分割が起きなくなるまで繰り返す．もし，根を分割する必要が生じたときは，今まで根であったプロセスが新しく根になるプロセスも生成する．

挿入操作によって新たにプロセスが生成される様子を図 3 に示す．この場合は，まずプロセス A_5のデータ挿入の際に分割が起き，プロセス A_6 が生成される．そして，それが上位レベルへ波及し根にあたるプロセス B_1が分割されたため，新たな根としてプロセス C_1が生成される．

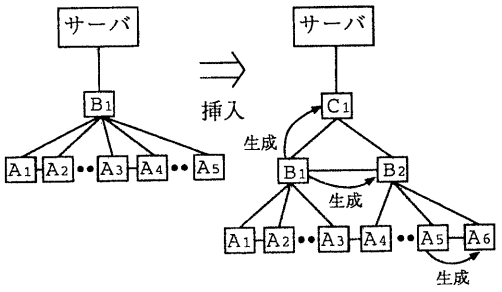

図 3: プロセスの生成

B 木におけるプロセスが図 3 の右側のようになっていたとすると，B 木における節点の親子関係とは別に，プロセスの生成順によって，各プロセスには次の親子関係が存在することになる．

- プロセス A_1は A_2の親プロセス，プロセス A_2は A_3の親プロセス…である．これは，プロセス B_iについても同様である．
- さらに，プロセス A_1は B_1の親プロセス，プロセス B_1は C_1の親プロセスである．

例えば，プロセス B_1に注目すれば，これはプロセスの生成順でいうと自分の親プロセスである A_1(B 木では子ノード)，兄弟プロセスである

A_2(子ノード) と C_1 (親ノード), 子プロセスである B_2(兄弟ノード), 兄弟プロセスの子孫である A_3(子ノード) と通信しなければならない. これらは, Cmex の子プロセス同士の通信を行えるポートを用いることにより達成される.

削除操作

検索と同様の方法で, データを削除すべき葉を見つけ出す. 葉のデータの数が $\lceil m/2+1 \rceil$ 個以上なら問題なく削除できる. $\lceil m/2 \rceil$ 個のデータを持っている場合には, その葉の左横の葉プロセスの持つデータと合わせて, それを二等分し再分配する. それでもデータの数が m にしかならない場合には, プロセスを B 木から削除する. そして, 削除したプロセスを親節点から削除するために削除更新要求を送る. 親節点のプロセスでも再分配, またはプロセスの削除が必要な場合には, 同様の処理を行う. これを再分配やプロセスの削除が起きなくなるまで繰り返す. もし, 削除により根プロセスの子供が一つになった場合には, 根プロセスは木から削除され, その子プロセスが新たな根プロセスとなる.

3.3 スクリプト記述

B 木の構造を次のように btree 型で宣言する. 引数は, 葉に格納するデータの型である.

```
btree * ::= Branch [(*, link *)]
          | Leaf [(num,*)] | Nul
```

ここで, 型 link は節点と節点とをつなぐ構造であり, 次のようにポートを用いて宣言される.

```
link * ::= Port (port (request *)) | Nil
```

各プロセスは次のような処理要求を受け取る.

SERCH	探索要求	UPD_K	キー更新要求
INS	挿入要求	TRANS	同期要求
DEL	削除要求	DEL_TD	再分配要求
UPD_I	挿入更新要求	UPD_L	左節点更新要求
UPD_D	削除更新要求	RPLY	結果

各プロセスでは, その B 木上での位置と受信した要求によって対応する関数を評価し, それぞれの処理を行う. 付録に, 並行 B 木探索において, 葉に位置するプロセス (図 1 の斜線部のプロセス) の探索・挿入処理を Cmex で記述した例を示す. 他のノードに位置するプロセスも同様のアルゴリズムを用いて記述することができる.

4 おわりに

本論文では, プロセスの概念を導入し, その引数である部分式を先行評価できるようにした分散型関数型言語 Cmex の構成を述べた. また, Cmex を, 並行／並列性のパラダイムの中でも重要なパイプライン処理へ適用し, その具体例として並行 B 木探索を示した. Cmex は, ポートを用いることにより, 多数のプロセスと, それらが子プロセス同士であっても自由に通信し処理を行えるため, 協調処理のパラダイムに適している.

今後は, 分散環境上での実行効率, より効率的な CPU へのプロセスの割り当ての検討などが必要であると考えている.

参考文献

[1] J. Darlington and M. Reeve, *ALICE, a Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages*, Conference on Functional Programming Languages and Computer Architecture, pp.65–76, October 1981.

[2] L. Augustsson and T. Johnsson, *Parallel Graph Reduction with the ⟨ν-G⟩machine*, Conference on Functional Programming Languages and Computer Architecture, pp.202–213, September 1989.

[3] M. van Eekelen, E. Nöcker, R. Plasmeijer and S. Smetsers, *Concurrent Clean*, Technical Report No. 90–20, Department of Informatics, Faculty of Mathematics and Informatics, University of Nijmegen, November 1990.

[4] A. Colbrook, E.A. Brewer, C.N. Dellarocas and W.E. Weihl, *Algorithms for Search Trees on Message-Passing Architectures*, Technical Report TR–517, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1991.

付録

```
|| B木の定義
lbtree * ::= Leaf [(num,*)] | Branch [(*, link *)] | Server | Nul

|| プロセス間のリンクの定義
link * ::= Port (port (request *)) | Nil

|| プロセス間通信の要求定義
request * ::= SERCH  num (link *)| INS num * (link *)| DEL num (link *)|
              UPD_I num (link *)| UPD_D num | UPD_K num num | TRANS (link*)|
              DEL_TD (link *) (lbtree *) (link *) (link *) num |
              UPD_L (link *) (link *) | RPLY num [*] | Nop
destination ::= Left | Right | Up | Dwn

|| 探索操作
search (key,upnode) (self,up,sv,l,r,sl,sr,m,rm,Leaf dat)
    = (self,upnode,sv,l,r,sl,sr,m,rm,nd)
      preset rply = search_lf key dat
             send = {> sv} rply

search_lf key []              = RPLY key []
search_lf key ((k1,v1):data) = RPLY key [v1],      key = k1
                             = search_lf key data, otherwise

|| 挿入操作
insert b (key,d,n) (self,up,sv,l,r,sl,sr,m,rm,Leaf dat)
    = ins_upd b (self,n,sv,l,r,sl,sr,m,rm, Leaf dat'), #dat' > b
    = (self,n,sv,l,r,sl,sr,m,rm, Leaf dat'),           otherwise
      where dat' = Leaf (ins_data (key,d) dat)

ins_data (key,val) [] = [(key,val)]
ins_data (key,val) ((k1,v1):data) = (key,val):(k1,v1):data,           key < k1
                                  = (k1,v1):(ins_data (key,val) data),otherwise

|| 挿入操作によりノードの分割が必要な場合に，それを処理する
ins_upd b cntxt
    = ins_upd_d b sync cntxt', (is_node_div b cotxt') = True
    = un_sync sync cntxt',     otherwise
      preset useport sync in private
             cntxt' = do_sync b r_sync (TRANS sync) cntxt

ins_upd_d b sync (self,up,sv,l,r,sl,sr,m,rm,nd)
    = (self,up',sv,l,r',sl,sr',m,rm,nd1)
       preset
         useport r' in private
         useport sr' in once
         (nd1,nd2)= div_node n (((len nd)+1) div 2) nd
         p = {p}process b (r',up',sv,self,r,sr',sr,minkey nd2,rm,nd2)
         updright = {> sync} (UPD_L r' sr')
         updup = {> up'}(UPD_I (minkey nd2,r'))

|| 他のプロセスとの同期操作
do_sync b dis req cntxt
  = cntxt,                                  res = []
  = do_sync b req (proc b cntxt (hd res)), otherwise
    preset res = dis cntxt req

r_sync (self,up,sv,l,r,sl,sr,m,rm,nd) req = {> sr} req

un_sync pt cntxt = cntxt
                   preset send = {> pt} Nop
```