プログラミング 11−4
(1997. 1. 23)

分散メモリ実装におけるデータフロー言語の構造データ管理方式

稲永 健太郎, 日下部 茂, 雨宮 真人
九州大学大学院システム情報科学研究科知能システム学専攻
〒816 福岡県春日市春日公園 6-1
E-mail:{inenaga,kusakabe,amamiya}@al.is.kyushu-u.ac.jp

あらまし プログラムの実行では, プログラムの動作保証がなされかつ有限なメモリ領域を効率よく利用する必要がある. 本稿では, non-strict なセマンティクスを持つデータフロー言語の構造データに焦点を当て, 構造データ領域の基本的な管理方式および再利用方式について述べる. 続いて, これら管理方式および再利用方式を分散メモリ型並列計算機上で実装する場合に実現すべき事項とその実現方法について述べる. この実現方法を例題に適用し分散メモリ型並列計算機富士通 AP1000 上で評価を行なった結果を示し, 再利用の効果および今後の課題について論じる.
キーワード データフロー言語, 構造データ, 分散メモリ実装

Structured Data Management in Implementation of Dataflow Languages on Distributed-Memory Machines

Kentaro Inenaga, Shigeru Kusakabe, Makoto Amamiya
Department of Intelligent Systems, Kyushu University
Kasuga Fukuoka 816, Japan
E-mail:{inenaga,kusakabe,amamiya}@al.is.kyushu-u.ac.jp

Abstract In this paper, we focus on structured data(SD) management in non-strict dataflow languages. First, we explain the basic management of SD and the reuse method of SD. Second, we disccuss the implementation technique of the management and the method on distributed-memory machines. We apply the technique to the program livermore loop12, and evaluate it on the distributed-memory machine, Fujitsu AP1000. We show the performance results, and disccuss the effectiveness and the future work.
key words Dataflow Language, Structured Data(SD), Distributed-Memory Machine

1 はじめに

データフロー言語はデータフローモデルに基づいた言語で，本質的に並列性を内在しその実行制御がデータ依存則に沿って行なわれる．このような言語は，明示的な並列実行制御の記述が不要であり，また抽象度の高い構造データ処理の記述が可能であるなど，並列処理記述の面で魅力的な特徴を持つ．しかし，それらの処理に関する具体的な記述を行なわない分を処理系がプログラムを解析して実現するか，あるいはランタイムシステムがサポートする必要がある．我々は，これらのうち実行効率を考慮して処理系の解析による実現方式を採用する．またデータフロー言語の中には，基本的に単一代入則を遵守し破壊的な代入を認めておらず，ループ処理において old や new のような変数の世代関係を明示するようプログラマに要求する言語があるが，その言語処理系では代入による副作用を考慮する必要がないものの，その反面メモリ領域の使用量が増大してしまう．

本稿では， non-strict なデータフロー言語の構造データに焦点を当て，構造データ領域の管理方式および再利用方式の分散メモリ実装について述べる．2 節では，データフロー言語の構造データ領域

の管理方式，特にプログラムの動作保証がなされる領域解放タイミングについて，また 3 節ではコンパイルの中間言語コードの解析による獲得領域の再利用方式について述べる．4 節では，2,3 節で述べた管理方式と再利用方式を分散メモリ型並列計算機上で実装する場合に実現すべき事項とその実現方法を述べる．5 節では，4 節で述べた実現方法を例題に適用し，分散メモリ型並列計算機 AP1000 上で評価および検討を行なう．6 節では，本稿で述べたメモリ領域管理についての関連研究について触れ，最後に 7 節でまとめを行なう．

2 データフロー言語の構造データ管理方式

関数呼び出しごとにある計算体が別の計算体を生成する場合，前者と後者の計算体の間に親子関係が存在するといい，基本的には全ての子の計算体の実行が終了した後に親の計算体の実行が終了する．また計算体は，自ら生成した構造データや親の計算体より引数渡しされる構造データへ操作を行なうことができる．構造データを引数渡しする方法として，領域管理が複雑であるが引数渡しのコストが小さくメモリ領域の使用量が少ないポインタ渡しと，領域管理がしやすいがメモリ使用量が多くなるコピー渡しがあるが，本稿ではメモリ領域の効率的利用を重視してポインタ渡しを原則とする．

基本的な構造データ領域管理には，以下に挙げる 2 種類の操作がある．

- 構造データ領域の確保
- 構造データ領域の解放

特に構造データ領域の解放操作のコードを処理系が挿入する場合には最低限プログラムの動作保証がなされることが必要であり，この要件を満たす基本的な構造データ解放タイミングについて次に述べる．

[基本的な解放タイミング] 対象となる構造データを操作する計算体のうち，最も祖先の計算体が実行を終了した時点で対象の構造データ領域の解放を行なう．

例として図1を示す．この図では 1 つの楕円が 1 つの計算体を表わしており，実線で結ばれた 2 つの計算体は，上の方が親の計算体，下の方が子の計算体であることを表わしている．また，構造データへの操作の順序を左肩に示す．ただし，言語の性質上全ての操作の順番が静的に確定してはいないため，例えば 1-2 という表記は 1 番目の操作と 2 番目の操作の間に実行されることを表している．この図では，網かけの計算体が構造データ A を操作する最も祖先の計算体であり，この計算体が実行を終了した時点で構造データ A の領域を解放する．

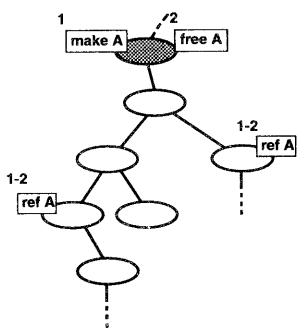

図 1: 構造データ領域の基本的な解放タイミングの例

3 構造データ領域の再利用方式

一度確保した構造データ領域を解放せずに再利用することにより，以下のようなメリットが得られる．

- 領域の効率的利用
- 領域確保コストの削減

獲得領域の再利用を行なうには，解放可能な領域が存在していることが前提となる．そこで，再利用可能な領域の候補を増やし再利用の可能性を高めることを考える．

3.1 再利用のための構造データコピー渡し

2 節で述べた解放タイミングに従った場合，祖先の計算体で解放される構造データ領域は不要となった後解放されるまでの時間的間隔が長くなる傾向がある．そこで，より子孫の計算体において構造データ領域を解放できる状態 (再利用可能な状態) とし，再利用する構造データの生成に先行して領域確保を行なうことを考える．本稿では，構造データが複数の子の計算体に引数渡しされる場合に限り構造データのコピー渡しを採用し，再利用可能な領域の候補を増やす．このコピー渡しにより複製された構造データは，別々に基本的な解放タイミングに従って解放される．

次に，構造データが複数の子の計算体にコピー渡しされる場合の，それぞれのパスにおいてより子孫の計算体で解放を行なうタイミングについて述べる．例として図 2 を示す．構造データ A は，2 つの網かけの計算体にコピー渡しされている．そして，網かけの計算体が構造データ B，C を参照する計算体のうち最も祖先の計算体であることから，その網かけの計算体がそれぞれ構造データ B，C の領域を解放する．ただし，複数の子の計算体へコピー渡しする場合，メモリ領域の効率的利用を考慮して図 2 では，1 つの子の計算体のみポインタ渡しを行なうことができる．

これに対し図 1 の場合は，操作 1. を行なう最も祖先の計算体が構造データ A の解放を行なうことになり，操作 1. を行なう計算体より子孫の計算体で新たに生成される構造データはそれぞれ別に領域を確保しなければならない．

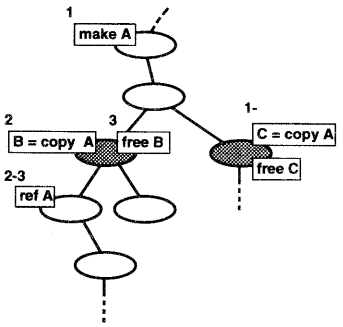

図 2: 再利用のための構造データコピー渡しの例

3.2 構造データ領域の再利用可能条件

一度確保した構造データ領域を再利用する場合，以下に示す条件を満たす構造データ領域のみ再利用の可能性が残されていることになる．

[構造データ領域の再利用可能条件]

1. 再利用の対象領域に存在していた “古い” 構造データへのアクセスがない．
2. “古い” 構造データの領域を再利用する “新しい” 構造データの候補が，処理系の解析により確定される．
3. “古い” 構造データと “新しい” 構造データとの間に「同一性」が存在する．

まず，条件 1. については，2 節で述べた構造データ領域の解放タイミングの決定と同様に解析できる．また，条件 2. については，グラフ形式の中間言語コードを用いた解析を必要とする．これについては 3.2.1 節で詳しく述べる．また条件 3. における「同一性」は，新旧 2 つの構造データがある実マシンの同一のメモリ領域を実際に利用できるかどうかを示すものであり，各実マシンでの構造データの実装方法により異なる．本稿では，4 節で分散メモリ型並列計算機上への実装した場合の新旧の構造データ間の「同一性」について述べる．

3.2.1 領域を再利用する構造データ候補確定に関する解析

構造データ領域の再利用可能条件の 2. に関する解析では，コンパイル時に生成されるグラフ形式の中間言語コードを用いて以下の手順に従い解析を行なう．この中間言語コードとは，ソースプログラムより字句解析・意味解析・データ依存解析を経て得られ，グラフ表現可能なマルチスレッドコントロールフローコードである．このコードは，同一の実行コンテクストを共有する 1 つ以上の命令列である命令スレッドとそれらスレッドの実行順序関係を表す継続スレッド指定 (グラフではアークで表現) から構成されている．

解析手順

1. 関数ごとに次に示す構造データを指すポインタの集合を求める．ここで用いるポインタは，ポインタ名とそのポインタ内容から構成されている．以下現れるポインタの表し方は，特に説明のない限りポインタ内容を指している．
   - $P_{def}(fn)$: 関数 fn 内で定義される構造データを指すポインタ p_{def} の集合
   - $P_{receive}(fn)$: 関数 fn 内の引数として受け取る構造データを指すポインタ $p(fn, slot)$ の集合．ここで slot は，引数として受け取った構造データを指すポインタの ID 番号となる．
   - $P_{rlink}(fn)$: 関数 fn 内で関数適用により呼び出した関数 rlink_fn から返り値として受け取る構造データを指すポインタ $p(fn, rlink\_fn)$ の集合

- 関数適用により $P_{return}(fn)$：関数 fn を呼び出した関数 return_fn へ返される構造データを指すポインタ $p(fn, return\_fn)$ の集合
- $P_{link}(fn)$：関数 fn 内の 関数適用により呼び出される関数 link_fn へ引数渡しされる構造データを指すポインタ $p(fn, link\_fn, slot)$ の集合．ここで slot は，引数渡しされる構造データを指すポインタの ID 番号となる．
- $P_{no\_ref}(fn)$：$P_{link}(fn)$ に属するポインタのうち，関数 fn 内で参照命令に用いられないポインタの集合

2. (構造データの基本的解放タイミング)

   基本的な解放タイミングに従って関数 fn において解放される構造データを指すポインタの集合 $P_{free\_sd}(fn)$ を求める．

   $$P_{free\_sd}(fn) \stackrel{def}{=} (P_{def}(fn) \cup P_{rlink}(fn)) - P_{return}(fn)$$

3. (再利用のための構造データコピー渡し後の解放タイミング)

   関数 fn において，ポインタ $p \in P_{free\_sd}(fn)$ と同じポインタ名を持つポインタ p' が $P_{no\_ref}(fn)$ に存在すれば次の 2 つの操作を行なう．

   - $P_{free\_sd}(fn) = P_{free\_sd}(fn) - p$
   - $P_{free\_sd}(link\_fn) = P_{free\_sd}(link\_fn) \cup p'$

   すべての関数において，条件を満たすポインタ p，p' がなくなるまでこの操作を続ける．

4. ("古い" 構造データの領域を再利用する "新しい" 構造データの候補の確定)

   関数 fn において $P_{output}(fn) \stackrel{def}{=} P_{link}(fn) \cup P_{return}(fn)$ を求め，"新しい" 構造データを指すポインタ $p_{def} \in P_{def}(fn)$ ごとに以下の手順に従って解析を行なう．

   (a) 関数 fn において，ポインタ $p_{def} \in P_{def}(fn)$ と同じポインタ名を持つポインタ p' が $P_{output}(fn)$ に存在すれば (b) へ．存在しなければ，$p'' = p$ として (e) へ．

   (b) $p'(= p(fn, link\_fn, slot)) \in P_{link}(fn)$ ならば，$p'' = p(link\_fn, slot)(\in P_{receive}(link\_fn))$ について (c) へ．

   $p'(= p(fn, return\_fn)) \in P_{return}(fn)$ ならば，$p'' = p(return\_fn, fn)(\in P_{rlink}(return\_fn))$ について (d) へ．

   (c) p''と同じポインタ名を持つポインタが $P_{output}(link\_fn)$ に存在すれば $p' = p''$として (b) へ．存在しなければ (e) へ．

   (d) p''と同じポインタ名を持つポインタ p'が $P_{output}(return\_fn)$ に存在すれば (b) へ．存在しなければ (e) へ．

   (e) このポインタ p'' が "古い" 構造データを指すポインタとなり，p''の指す構造データの領域を再利用する候補として，p_{def}の指す "新しい" 構造データを確定する．

ただし，ここでの解析により確定された "新しい" 構造データの候補による再利用が必ずできるということを保証しているわけではない．そこで，実際に再利用できるかどうかを判断するため，前にも述べた新旧の構造データ間の「同一性」を調べる必要がある．

4 構造データ管理の分散メモリ実装

2,3 節で述べた構造データ管理方式と構造データ領域の再利用方式を分散メモリ型並列計算機で実装する場合に実現すべき事項およびその実現方法を述べる．本節では構造データの具体例として配列[1]を用いる．

4.1 配列領域管理の実現すべき事項

まず，配列の特徴として以下のような点が挙げられる．

- 言語の性質上，配列への操作も非同期的に行なわれる．
- 配列に関する情報を保持するディスクリプタが用意されている．

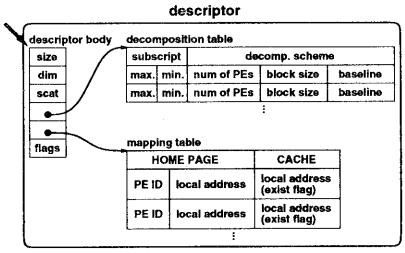

図 3: ディスクリプタの例

図 3にディスクリプタの例を示す．このディスクリプタは以下に挙げる 3 つの部分から構成され，配列の生成ごとに 1 つ生成される．

[1] この配列はいくつかの断片に分割され各 PE に配置される．本稿ではこの分散情報をプログラマが指定すると仮定する

- ボディ部

  配列全体の情報，例えば全要素数 (size) や次元数 (dim)，部分配列[2]数 (scat) などが含まれている．

- 分割テーブル部

  配列の分割に関する情報，例えば各次元ごとに割り当てられた PE 数 (num of PEs) や部分配列の各次元の大きさ (block size) などが含まれている．

- 配置テーブル部

  各部分配列領域に関する情報，例えば各部分配列が割り当てられた PE 番号 (PE ID) やその割り当てられた PE 上での部分配列領域の先頭アドレス (local address) などが含まれている．

分散メモリ実装における配列の領域管理および領域の再利用では，このディスクリプタを用いる．また，分散メモリ実装ではポインタの役目をこのディスクリプタが担っているため，配列の引数渡しはこのディスクリプタをコピー渡しすることで実現される．

以上述べた配列の特徴を考慮して，分散メモリ実装における配列の領域管理に関して実現すべき事項としては

- PE 間通信を用いた配列領域の管理
- 非同期的な配列操作を考慮した配列領域の管理
- ディスクリプタのメモリ領域管理

が挙げられる．4.2 節ではこれらメモリ領域管理についての実現方法を述べ，4.3 節では，配列領域の再利用の実現方法について述べる．

4.2 配列領域管理の実現方法

4.2.1 PE 間通信を用いた配列領域の管理

配列領域の解放を行なう場合は以下のような手続きにより行なわれる．

1. 対象の配列の領域解放を行なう計算体は，その配列のディスクリプタに保持されている情報のうち
   - 部分配列が割り当てられている PE 番号 (配置テーブル内の PE ID)
   - 各部分配列が存在する PE 上での部分配列領域の先頭アドレス (配置テーブル内の local address)

   を用いて，PE 番号の指す PE に対してそれぞれ local address を含む解放要求メッセージを送信する．(自 PE に存在する部分配列領域は local address を用いて直接領域の解放を行なう)

[2] 配列の各 PE ごとに割り当てられる配列領域を指す

2. 部分配列領域の解放要求メッセージを受信した PE は，そのメッセージに含まれる local address を用いてその領域を解放する．

4.2.2 非同期的な配列操作を考慮した配列領域の管理

データフロー言語では，配列への操作も非同期的に行なわれ並列度の高い実行が可能である．これは，データ (配列要素) 側で生成と参照の同期を取ることにより実現されている．このようにデータ側で生成と参照の同期を取る配列を生成する場合には，配列生成を行なう計算体よりもその親の計算体の方が早く実行が終了することがあり得る．これは，2 節で述べたデータフロー言語の実行の，全ての子の計算体の実行が終了した後に親の計算体の実行が終了することに反する可能性が出てくる．

そこで，新たな配列の生成を行なう計算体へ配列を引数渡しする場合は，ポインタを渡すのではなく，配列全体をコピー渡しすることでプログラムの動作保証を行なう．そしてコピー渡しされた配列は，新たな配列生成の計算体ごとに従来の解放タイミングに従って解放される．

4.2.3 ディスクリプタのメモリ領域管理

4.1節でも述べたように，分散メモリ実装において，ポインタの役割をするこのディスクリプタは各計算体にコピー渡しされており，渡されたディスクリプタは 1 つの計算体内で用いられる．したがって，各計算体は保持しているすべてのディスクリプタのメモリ領域を解放する．

4.3 配列領域の再利用の実現方法

配列領域を再利用するには，3 節で述べた解析により選ばれた新旧 2 つの配列の「同一性」を調べる必要がある．この 2 つの配列間の「同一性」は，具体的に以下に挙げる項目について同じであるかどうかを，まず静的な解析によりチェックする．もし，静的なチェックにより再利用できないと判断した場合でも，今度はディスクリプタを用いて実行時にチェックするためのコードを処理系が挿入する．

- 処理系による静的チェック
  1. 各次元の添字範囲
  2. 次元数
  3. プログラマにより指定された部分配列の配置情報

     プログラマが指定した分散情報のうち，どの部分配列がどの PE 上に配置するかを 2 つの配列の間でマッチングを取る．
- ディスクリプタを用いた実行時チェック
  - 部分配列の個数 (scat) が等しい．

2. 各部分配列内の要素数 (size/scat) が等しい.

3. 配列の配置関数が同じである.

   ここでいう配置関数とは，部分配列 ID を引数に取り PE ID を返す関数である．これは各部分配列がどの PE 上に配置されているのかを示すものであり，各部分配列ごとにある配置テーブル部の PE 番号のマッチングを取る.

また配列領域を再利用する場合には，再利用する領域を初期化する必要がある．これは，以下のような手続きにより行なわれる.

1. “古い” 配列のディスクリプタを “新しい” 配列を生成する計算体にコピー渡し，“新しい” 配列のディスクリプタとする.

2. “新しい” 配列を生成する計算体は，解放の部分配列領域の解放要求の場合と同様に，部分配列が割り当てられた PE 番号 (PE ID) へ部分配列の先頭アドレス (local address) を含んだ各部分配列内の全要素の初期化要求メッセージを送信する.

3. 初期化要求メッセージを受け取った PE では，メッセージ内に含まれている部分配列の先頭アドレス (local address) を用いて部分配列の全要素を初期化し，その後に初期化終了を知らせる ack を返す.

   要素の初期化の方法は，配列の持つ構造により異なるが，本稿で用いている配列の要素には，非同期的な配列操作を実現するために各配列要素に I-structure[3] という同期機構が備えられている．この I-structure を備えた配列要素の初期化は，要素の状態を初期状態に戻す iclear 操作により行なわれる.

4. 配列初期化要求メッセージを送信した計算体が，初期化の ack を全て受け取った後，“新しい” 配列のディスクリプタを他の計算体に渡す.

またこの初期化に関して全ての要素が初期化されているかどうかを確認するため PE 間で同期を取る必要があり，この点に関して領域の解放の場合とは異なる.

5 評価

本節では，4 節で述べた分散メモリ実装における配列の管理方式および再利用方式をプログラム livermore loop12 に適用する．そして，構造データ領域を再利用する／再利用しない場合の

- 構造データ領域の延べ使用量
- プログラムの全実行時間

を比較し，構造データ領域の延べ使用量の削減および再利用における時間的コストの削減がなされているかどうかについての評価を行なう．今回の評価では，各 PE での部分配列の存在するメモリ領域 (heap 領域) 全体は，フリーリストと呼ばれる，双方向循環リストを構成する未使用連続領域からなるブロックを要素に持つリストを用いて管理されている.

5.1 構造データ領域の延べ使用量

ここでは配列サイズ・使用 PE 数を固定し loop 回数を変化させた場合の構造データ領域の延べ使用量の相対比を示す.

[1. 構造データ領域の延べ使用量 (相対比)]

loop 回数	再利用する／再利用しない
2	0.667
4	0.4
8	0.222
16	0.118
32	0.061
64	0.031
128	0.016

この結果より，構造データ領域の延べメモリ使用量の相対比はいずれの場合も 1 より小さく，構造データ領域の再利用による延べメモリ使用量の削減が確認できる．また，延べメモリ使用量の相対比がほぼ loop 回数に反比例していることもわかる．これは，配列領域を再利用した場合に延べメモリ使用量が一定であるのに対し，再利用しない場合には loop 回数に比例して延べメモリ使用量が増加していることを意味しており，特に例題のような loop の実行において再利用を行なう場合には，loop 回数が増加するほどより高い効果が得られることが考えられる.

[2. プログラム全体の実行時間の比較]

(a) loop 回数：変化，配列サイズ・使用 PE 数：固定

(b) 配列サイズ：変化，loop 回数：使用 PE 数：固定

の 2 つの場合について，それぞれ構造データ領域を再利用しない場合と再利用する場合のプログラム全体の実行時間を図 4，5に示す.

この図よりプログラム全体の実行時間は loop 回数や配列サイズに比例して増加している．再利用した方が loop 回数および配列サイズに比例して領域確保および解放分の時間的コストが減少すると考えられたが，予想に反しプログラムの実行時間にはほとんど差は見られなかった．これは，次のような原因によるものと考えられる.

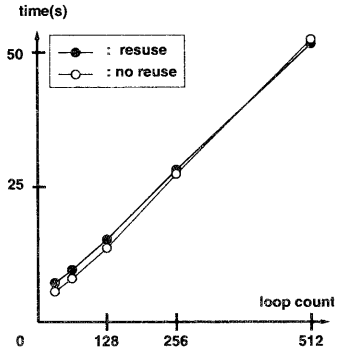

図 4: 2-(a) loop 回数を変化させた場合

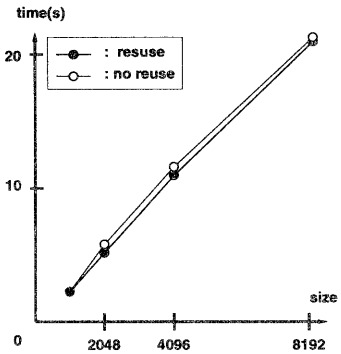

図 5: 2-(b) 配列サイズを変化させた場合

まず構造データ領域を再利用するには，前の繰り返しから再利用する構造データ領域へのポインタ (今回のプログラムでは配列のディスクリプタを指す) を次の繰り返しへ渡す必要がある．その際ポインタを他の PE 上に存在する計算体へ渡すのに PE 間通信を用いる場合がある．今回の評価では再利用しない場合に比べて PE 間の通信回数が loop 回数に比例して増加している．今回用いた例題のように規模の小さな問題を AP1000 上で実行した場合は，実行時間に占める通信の固定コストの割合が大きいため [5]，増加した分の通信の固定コストが領域確保および解放のカットによる時間的なコスト削減の効果を打ち消していると考えられる．また別の原因として，再利用に用いるポインタを受け取るまで新たな構造データを生成できないという点も挙げられる．これは，再利用する構造データの生成を行なうタイミングが再利用しない場合より遅くなりパイプライン的処理の並列性が抑制され，並列実行により隠蔽可能な領域確保および解放による時間的コストが隠蔽できなくなったことを意味しており，再利用による時間的コストの削減効果を打ち消していると考えられる．

これら原因に対しては，まず通信に関しては同じ計算体に送信するメッセージをまとめることにより通信の固定コストを減らす手法 [4] を適用することが考えられ，またポインタ渡しによるパイプライン的処理の並列性に関しては，親の計算体がすでに保持している再利用可能な配列領域の情報を先行して子の計算体に渡すことにより処理の並列性の削減を抑制できると考えられる．またメモリ領域管理に関して時間的コストを削減する方法として，heap 領域全体の管理メカニズムを検討する必要もある．今回の評価ではフリーリストを含む heap 領域管理メカニズムを用いて構造データ領域の確保を行なっているため，このフリーリストから適切な未使用連続領域を探索するのに必要以上の時間的コストがかっていることが考えられる．この heap 領域管理メカニズムを含めてこれらの時間的コストに関する問題にいては今後さらに検討を行なっていく．

6 関連研究

配列をはじめとする構造データの領域管理に関しては，各種言語において異なる形態をとっている．例えば，我々が開発を行なっている超並列 V 言語 [7] は，明示的にメモリ領域操作を記述せずに処理系がこれらのメモリ領域操作に関するコードを挿入するが，これに対し逐次 C や 超並列 C 言語 NCX[6] のようにプログラマに明示的にメモリ領域操作を記述させるという点で V 言語をはじめとするデータフロー言語とは異なる．また，本稿で述べた管理方式と同じくメモリ領域操作に関するコードを処理系が自動的に挿入というスタンスを採っている言語として SISAL が挙げられる [2]．SISAL も単一代入則を遵守しているため書き換えによる副作用が生じないという特徴を持つ．ただし，SISAL ではプログラマに明示的に old という記述を構造データの変数名の前に付けさせることで，暗にメモリ領域操作の方針をプログラマに要求し書き換えによる副作用を生じないようにしている．これに対し V 言語の場合では，同じデータフロー言語でも同様の記述をプログラマに要求せずに，書き換えの副作用を生じないようなメモリ領域の再利用を行なっているという点においてこの言語とは異なるスタンスを採っている．その他のメモリ管理方式において，オブジェクト指向言語における GC が挙げられる [1]．これはオブジェクトが用いる領域の自動管理機構であり，プログラム本体の実行の流れからは独立した形で不必要なメモリ領域を回収していく．これは，本稿で述べたプログラムの実行中のどのタイミングで領域の解放を行なっていくかを処理系の解析により決定し，自動的にコードを挿入するという領域管理方式とは異なる方法を採っている．

7 おわりに

本稿では，データフロー言語の構造データ管理方式と構造データ領域の再利用方式について述べ，それらを分散メモリマシン上で実装する場合に実現すべき事項およびその実現方法を述べた．また，これらの管理方式および再利用方式を例題に適用し，分散メモリ型並列計算機 AP1000 上において評価を行なった結果，構造データ領域の延べ使用量の削減が確認できた．ただし時間的コストに関しては，我々の予想に反し再利用を行なう／行なわないに関わらずほぼ同じ実行時間を要した．しかし検討の結果，再利用を行なった上での時間的コスト関してはさらなる向上が見込めるため，今後はメモリ領域の管理メカニズムも含めて再利用を行なった上での実行効率の向上のための最適化を検討していく．

参考文献

[1] K. A. M. Ali and S. Haridi. "Global Garbage Collection for Distributed Heap Storage Systems". *International Journal of Parallel Programming*, Vol. 15, No. 5, pp. 339–387, 1986.

[2] David C. Cann and Paraskevas Evripidou. "Advanced Array Optimizations for High Performance Functional Languages". *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 3, pp. 229–239, March 1995.

[3] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eichken. "TAM – A Compiler Controlled Threaded Abstract Machine". *Journal of Parallel and Distributed Computing*, pp. 347–370, 1993.

[4] P.Banajee, J.A.Chandy, M.Gupta, J.G.Holm, A.Lain, D.J.Palermo, S.Ramaswamy, and E.Su. "The PARADIGM Compiler for Distributed-Memory Message Passing Multiprocessors". In *In the First International Workshop on Parallel Processing*, 1993.

[5] 清水俊幸, 堀江健志, 石畑宏明. "AP1000 の性能評価". Technical report, (株) 富士通研究所.

[6] 湯淺太一他. "超並列 C 言語 NCX 言語仕様書 (Version 3)". 重点領域研究「超並列原理に基づく情報処理基本体系」 第 4 回シンポジウム予稿集, pp. 2–8 – 2–92, 1994.

[7] 日下部茂, 雨宮真人. "超並列 V 言語". 重点領域研究「超並列原理に基づく情報処理基本体系」 第 4 回シンポジウム予稿集, pp. 2–143 – 2–190, 1994.