Bulletin of Aichi Univ. of Education, 60（Natural Sciences）, pp. 35–39, March, 2011

GCD による Lisp の組み込み関数 member の並列化

安本　太一

情報教育講座

Parallelization of the Built-in Function member in Lisp by GCD

Taichi YASUMOTO

Department of Information Sciences, Aichi University of Education, Kariya 448-8542, Japan

1　はじめに

Lisp の組込関数の並列化について報告する．これまでに行われた Lisp 言語及びその処理系における並列化の研究は，マルチプロセッサを対象としたもので，並列実行されるコンテキストを明示的につくる，粒度の大きい並列化を対象にしていた [1, 2].

今日，マルチコア CPU を搭載したコンピュータが身近なものとなってきている．Dual-Core CPU を搭載したパーソナルコンピュータは 6 万円台から入手可能であり，Quad-Core CPU，6-Core CPU を搭載したパーソナルコンピュータも発売されている．今日のこれらのコンピュータは，単に複数のコアを一つのチップに載せただけではなく，キャッシュを共有するようになっている．複数のコアの間でデータのやりとりをする時や，複数のコアが同一のデータにアクセスする時のコストが低くなっている．

並列処理のフレームワークとして，Pthreads といったマルチスレッドの実装が普及している．Pthreads は，関数を並列実行する．一方，Apple 社は，Pthreads より軽量なマルチスレッドとして，Grand Central Dispatch (以下 GCD という) を提案している [3, 4, 5, 6]．GCD は，ブロック (複文) を並列実行する．GCD の実現に関わって，Apple 社は，C 言語の複文 (compound statement) をデータとする Blocks という構文を提案している．

このようにハードウェアとソフトウェアの両面から，粒度の小さい並列処理へマルチスレッドを適用しても，実行効率向上が期待できる環境が整えられてきている．本稿では，Lisp 言語の組み込み関数 `member` の実装にマルチスレッドを用いて高速化を試みたので，その結果を報告する．

2　Grand Central Dispatch の概要

本研究に関係したことに的を絞って，Grand Central Disptach (GCD) の概要を説明する．GCD は，C 言語の複文 (compound statement) を Blocks というデータとし，この Blocks を並列実行する機能を提供する．並列実行したい (並列実行できることを期待する)Blooks をキューに投入すると，GCD のランタイム環境が，実行環境のリソースの状況 (CPU のコア数や CPU の利用状況) に応じて，これらの Blocks を可能な限り並列実行する．

図 1 に反復構造の例を示す．C 言語の複文に ^ を前置して作成した Blocks を，関数 `dispatch_group_async` に渡すと，この
Blocks を主スレッド (`dispatch_group_async` を呼び出したスレッド) と並列に実行する事ができる． この例では，意味論上は 10 個の Blocks が主スレッドとは非同期に実行されるが，実際に同時に並列実行される Blocks の数は，実行環境のリソースの状況による．

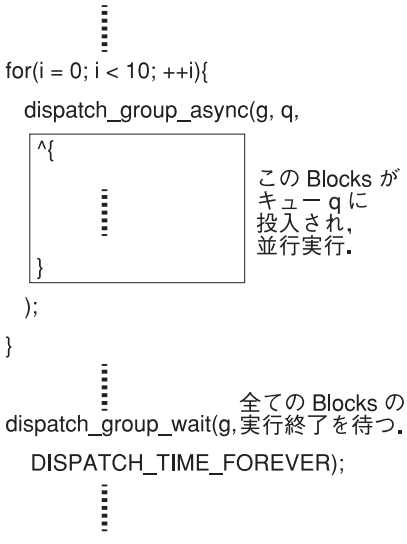

図 1: 反復構造における GCD の例

図 2 に逐次構造の例を示す．前の方の Blocks と後ろの方の Blocks が，主スレッドとは非同期に実行される．これらのスレッドが，実際に並列実行されるか否かは，実行環境のリソースの状況による．

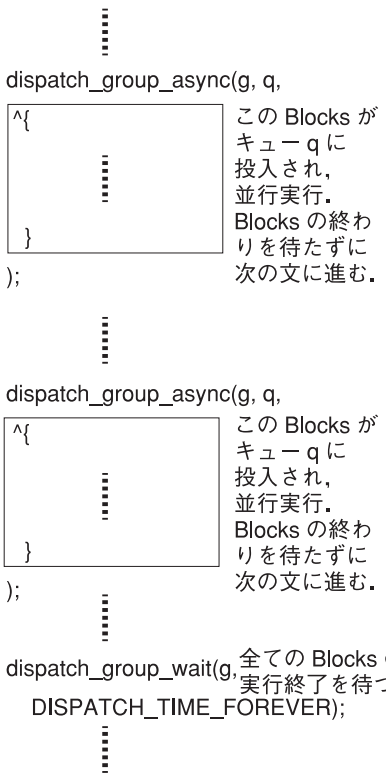

図 2: 逐次構造における GCD の例

図 1 の反復構造，図 2 の逐次構造のどちらの場合も，関数 `dispatch_group_wait` により，並列実行されている Blocks の終了を待ち，主スレッドとの同期が行われる．

これらの例が示すように，GCD は複文の並列実行を基本としている．GCD には，関数を並列実行する Pthreads のように，マルチスレッド化により変数のスコープが途絶える不便さがない．GCD による並列化は，並列実行したいプログラムの部分を `dispatch_group_async` でくくり，主スレッドとの同期場所を示す程度でできるので，GCD は粒度の小さい並列化を容易にする．

3　組み込み関数の並列化

過去の Lisp 言語およびその処理系の研究において，並列処理を行うために，新たな構文 (関数や特殊形式) を導入する事が行われた．しかしながら，プログラマにとって，プログラムを明示的に並列化することは負担であることに変わりはない．並列処理のための構文は標準化されていないので，並列処理のための構文を用いると Lisp プログラムの移植性がなくなってしまう．

プログラマに負担にかけずに並列処理を行い，Lisp プログラムの高速化を行う方法としては，組み込み関数を並列化して実行効率を向上させることが考えられる．組み込み関数自身に，並列化する余地がある場合に限られるが，プログラマが何もしなくても，Lisp プログラムの高速化が行えるのは，大きな利点である．

Lisp には，Lisp 言語が特徴とするリストというデータ構造がある．リストは，データの並びである．このデータの並びは，コンスというデータが連結されて実現されている．コンスは，場所を格納するところが 2 つあるデータである．リストは単純なので簡潔で柔軟性が高いという利点があるが，リストに格納されているデータにアクセスするためには，リストの先頭から順にコンスをたどらなければならず，決して効率が良いとはいえない．配列とは異なり，コンスが連続されたメモリ領域に配置されていないからである．また，組み込み関数 `member` のように，リストを先頭からたどることを前提としている関数もある．`member` は，リストに，探索したいデータが含まれているかどうかをリストの先頭から探索し，探索したいデータがみつかったら，探索したいデータを先頭とするリストを返す関数である．

`member` のように，リストを先頭から順にたどることを前提にしている関数は他にもあるので，図 3 に示すように，2 つのスレッドで手分けをして，それぞれ，偶数番目と奇数番目のデータを調べるという並列化が考えられる．次の次のコンスをたどるための特別なデータ構造を用意していなければ，各スレッドは，`cdr` 部の `cdr` 部をたどって，次のデータにアクセスすることになる．

同様の考え方で，3 スレッド版，4 スレッド版というように，より多くのスレッドを使う実装も考えられるが，データを調べもしないコンスをたどるオーバヘッドが大きくなると推測される．

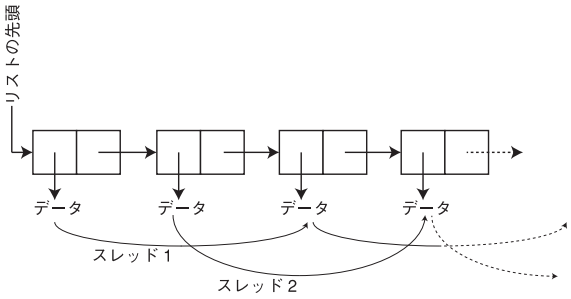

図 3: リストの走査の並列化

4 並列化した member のアルゴリズム

今回並列化を試みた組み込み関数 member のアルゴリズムを，図 4 の疑似コードで示す．list, item は，(member item list) というように member を呼び出したときの引数に相当する．endp は，リストの終わりに達しているかどうかを調べるマクロである．->car や ->cdr は，それぞれ car 部や cdr 部をさす．

最初に非同期に実行されるスレッドは，リストの先頭から 0 番目，2 番目，4 番目，…の要素をあたっていく．一方，次に非同期に実行されるスレッドは，先頭から 1 番目，3 番目，5 番目，…の要素をあたっていく．等しいかどうかは，eql で判定する．

非同期で実行される 2 つのメッソドは，最終的に主メソッドと同期され，最初に非同期に実行されたメソッドの結果が優先される．つまり，member の定義にあうように，次のようなリダクション (reduction) が行われる．最初に実行されたメソッドの結果が nil でない場合は，最初に実行されたメッソドの結果が採用される．さもなければ，次に非同期に実行されたメソッドの結果が採用される．

5 並列化した member の性能評価

並列化を提案した member のアルゴリズムを，Common Lisp 言語処理系である Kyoto Common Lisp (KCL) [7, 8, 9] の member の実装に適用した．その実行時間を計測することにより，評価を行った．次の評価用プログラムを用い，オリジナル (逐次版) の member と並列版の実行時間を比較した．

```
(member
  無限長整数 無限長整数を要素とするリスト)
```

検索対象の無限長整数 (bignum) は約 2000 桁，リストは 6 万個の約 2000 桁の無限長整数を要素とするリストである．リストの最後にある無限長整数と，探索したい無限長整数が一致するようなデータを引数として与えた．この評価用プログラムは，無限長整数を用いていて一般的でないと思われるかもしれないが，その理由については後で説明する．

実行時間を計測するのに用いた環境は，Apple 社製の Mac Pro (Intel Dual Core Xeon 2.66GHz × 2, Mac OS X 10.6.4) と Mac Book (Intel Core2 Duo 2.0GHz × 1, Mac OS X 10.6.4) である．member を並列化した KCL は，64 ビット版である [10]．GCD をサポートしているのは，64 ビット環境が標準である MacOSX10.6 以降だからである．

```
object list, item, reuslt;
object list1, list2;

list1 = list2 = list;
以下を，主スレッドとは，非同期実行
  while(!endp(list1)){
    if(eql(list1, item))
      break;
    list1 = list1->cdr;
    if(endp(list1))
      break;
    list1 = list1->cdr;
  }

以下を，主スレッドとは，非同期実行
  if(!endp(list2)){
    list2 = list2->cdr;
    while(!endp(list2)){
      if(eql(list2, item))
        break;
      list2 = list2->cdr;
      if(endp(list2))
        break;
      list2 = list2->cdr;
    }
  }

非同期実行したスレッドが全て終了するのを待つ

if(list1 != nil)
  result = list1;
else
  result = list2;
```

図 4: 並列版 member のアルゴリズム

評価用プログラムを実行するのに要した時間を，表 1 に示す．並列版の実行時間は，逐次版の約 1/2 から 2/3 程度に短縮された．マルチスレッドによる並列化の効果が現れている．

表 1: member の逐次版と並列版の実行時間の比較

実行環境	逐次版	並列版	比
MacPro MacOSX10.6	0.033	0.017	0.52
MacBook MacOSX10.6	0.050	0.033	0.66

(単位は秒)

member は，eql で比較を行うので，無限長整数の比較は，(アドレスの比較ではなく) 全桁の比較により行われるので，コストが高い．KCL では，無限長整数を，基本データの整数をポインタで連結して，実現している．64 ビット版の KCL では，この基本データの整数は 64 ビットの整数である．約 2000 桁の無限長整数どうしの比較は，KCL では約 105 回の (基本データの) 整数の比較によって実現される．同値か否かの判定であれば，必ずしも全桁比較する必要はないが，同値か否かの判定は，大小比較もあわせて行う関数によって内部的に実装されているので，全桁比較することになる．

6　考察

性能評価のプログラムにおいて，無限長整数を用いたのは，記号を要素としたリストに対する記号の探索では並列化の効果が十分確認できなかったからである．

記号の比較は，記号の唯一性により，アドレスの比較だけでできてしまうので，ごく短時間で終わってしまう．そのうえ，今回提案した member の並列化では，要素の比較は一つおきだが (各スレッドは比較は半分ずつ分担して担当するが)，双方のスレッドは (リストの終わりに達していないかどうかををチェックしながら)，リストを先頭から終わりまで 順に全部たどるというオーバヘッドがある．したがって，同値か否かの比較に十分なコストを要しないと，並列化の効果がでないのであろう．

リストを半分にわけて，前半分，後半分を別々のスレッドに探索させるという手法も考えられるが，リストの破壊的な操作を考えると，リストの中間点を決めるのは困難である．また，前半分で探索が終わってしまうケースでは，並列化の恩恵が受けられない．

比較のコストが高いケースの方が並列化の効果が現れるということであれば，member において，キーワード :test あるいは :test-not を使って，eql でない検査のための関数を指定することが考えられる．しかしながら，eql ではない関数を指定する場合は，並列化のコストが高くなってしまう．:test や :test-not を指定しない場合は，eql であるが，KCL では内部的な C 言語関数 eql が用いられので，Lisp 関数のフレームは必要ない．しかし，:test や :test-not を指定した場合は，任意の Lisp 関数が指定できる．任意の Lisp 関数の場合は，スレッドで並列実行される Lisp 関数のフレームを関数ごとに用意しなければならない．スタックの分岐を作るなど，メモリ管理が複雑となることが容易に予想される [2]．これが大きなオーバーヘッドとなり，並列化による実行性能の向上が得られない可能性がある．また，スレッドで並列実行されている Lisp 関数にエラーが生じた時，どのように解釈して対処するかといった意味論の問題もある．現時点では，:test や :test-not による任意の Lisp 関数の指定は想定していない．

7　まとめ

並列処理のための特別な構文を用意するのではなく，組み込み関数を並列化することによって，Lisp プログラムの速度向上を試みた．GCD というマルチスレッドのための技術を用いて，組み込み関数 member の並列化を行った．非常に長い桁数の無限長整数を要素とする長いリストに対して，非常に長い桁数の無限長整数を探すというケースにおいて，逐次版 (オリジナル) に対して，並列版 (2 スレッド版) では実行時間が約 1/2 から 2/3 程度に短縮された．リストをたどるコストより，要素比較のコストの割合が高くなると，マルチスレッドによる，速度向上が顕著に現れるようである．

今回用いた GCD は，Pthereds のように並列実行したい部分を関数化する必要がなく，並列化したい複文を指定するだけでよいので，member のマルチスレッド化は容易に行う事ができた．GCD では，Pthread より粒度の小さい並列化が可能であることを経験した．

今日では，一昔前では考えられなかった，著しく大容量のデータをコンピュータで扱う事が多くなった．小容量のデータを扱う場合においては，実行効率の向上が明確にみられなかった手法でも， 大容量では有用となる事例もでてくるであろう．Lisp 言語処理系において潜在的に並列化可能な部分をみつけ，GCD などを用いてマルチスレッド化するチューニングにより， 明示的な並列構文を導入しなくても，Lisp プログラムの速度向上をはかる必要性があると思われる．

参考文献

[1] 川本真一, 伊藤貴康 : マルチスレッドを用いた PaiLisp インタプリタの実現と評価, 情報処理学会プログ

ラミング研究会研究報告, Vol.1994, No.65, pp.97–104 (1994).

[2] 松井俊浩, 関口智嗣 : マルチスレッドを用いた並列 EusLisp の設計と実現, 情報処理学会論文誌, Vol.36, No.8, pp.1885–1896 (1995).

[3] Concurrency Programing Guide, Apple Computer (2009).

[4] Grand Central Dispatach (GCD) Reference, Apple Computer (2009).

[5] Grast, B. : *APPLE'S EXTENSIONS TO C*, http://open-std.org/jtc1/sc22/wg14/www/docs/n1370.pdf, March 10,2009.

[6] 千種菊里 : Mac 解体新書 第 2 回 Blocks—GCD の根幹を担う文法拡張, 月刊アスキードットテクノロジース, Vol.14, No.9, pp.156–164(2009).

[7] Steele, G. L. Jr. : *Common Lisp the language*, Digital Press (1984).

[8] Yuasa, T. and Hagiya, M. : *Kyoto Common Lisp Report*, Teikoku Insatsu Publishing (1985).

[9] Yuasa, T. : Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol.13, No.3, pp.284–295 (1990).

[10] 安本太一 : Common Lisp 言語処理系の 64 ビット化, 愛知教育大学研究報告 自然科学編, 五十三輯, pp.27-32 (2004).

（2010年 9 月17日受理）