マルチスレッド並列EusLispの分割型メモリ管理手法

松井俊浩，関口智嗣
matsui@etl.go.jp、sekiguchi@etl.go.jp
電子技術総合研究所
〒305 茨城県つくば市梅園1-1-4

EusLispは，幾何モデリング機能を備えたオブジェクト指向プログラミング言語である．EusLispが目的とするロボットのプログラミングに必要な，計算能力の増大と非同期制御を可能とするために，マルチスレッドを用いた並列化機能を付加している．しかし，メモリ要求を頻繁に発するプログラムでは，メモリ管理用のグローバルなデータベースを相互排除する必要上、期待するほどの並列性能が得られなかった．本論文では，このメモリ管理データベースの一部をスレッドにローカルに持たせ、分割型のメモリ管理を行うことで，並列性能の向上が可能となることを示す。ベンチマークを用いた評価では，メモリ管理負荷の大きいプログラムでも２以上の並列利得が得られ，旧版に比べて数倍以上の性能向上が観測された．

Thread Distributed Memory Management Scheme of Multithread Cuncurrent EusLisp

Toshihiro Matsui and Satoshi Sekiguchi
Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, Ibaraki, 305 Japan

EusLisp is an object-oriented programming language with geometric modeling facilities for robot programming. EusLisp has been extended to have the parallel programming capability to provide higher computation power and asynchronous programming facility required for robot applications. Memory hungry programs, however, were unable to expect reasonable performance gain because of the necessity of mutual exclusion when the global database for memory management is manipulated. This paper presents the parallel performance of these programs is improved by the thread distributed memory management, which transfers the portions of free cells to many threads and allows local management of those cells. In this manner, memory hungry programs can obtain higher parallel gain apparently better than the former implementation.

1. はじめに

特にロボット研究への応用を目指してEusLispと呼ぶオブジェクト指向型Lispを開発してきた[1,2,3]．Common Lisp [5]のほとんどの機能をオブジェクト指向の上に実現しており，さらにロボットがセンシングや行動を計画するために必須な3次元幾何モデルを扱う機能を備えている．

ロボットプログラミングには二つの観点から並列処理機能が重要である．一つは，並列処理から得られるスケーラブルな計算パワーである．たとえば3次元環境中でのロボットの行動計画における環境中の物体との干渉検査，画像，音響等の信号処理における大量の情報処理に対応するため，並列処理は本質的なアプローチとなる．二つ目は非同期処理である．ロボットは，さまざまなセンサーからのデータに基づいてを最適な行動を選択するが，センサーから入力されるイベントは基本的に非同期である。一方，数十台規模のマルチプロセッサUNIXマシンが利用できるようになってきている．そこで、これらの並列機の上で並列・非同期プログラミングを達成する機能をEusLispに加え，マルチスレッド並列EusLispとして発表している[4]．

この並列EusLispは，ベンチマークプログラムによる評価によって，メモリ要求をあまり発しないプログラムでは高い並列利得を得られることを実証しているが，メモリ要求頻度の高いプログラムは，逐次実行より性能が劣化する．主たる原因は，共有メモリの割り付けとゴミ集めに要するスレッド間の同期制御にある．本論文では，この同期制御の軽減を目指して実装した分割型の共有メモリ管理手法について論ずる．

2．EusLispの概要

EusLispは，幾何モデラの効率的な実現を目的としている．EusLispのオブジェクト指向機能は、モデルの表現に適しており、Lispのポインタ、リスト操作機能は面や稜線などの要素間のトポロジーの操作に都合がよい。

オブジェクト指向型Lispの効率的実行に重要なのは，メモリ管理と型検査である．EusLispでは，可変長のオブジェクトの割り付けと回収を効率よく実行するために，フィボナッチバディ法によるメモリ管理を行っている[1,3,5]．動的に変化しうる型階層の中で，オブジェクトの属する型継承木を高速で判定する方法として，区分木の考え方に基づき，2回の整数の大小比較でクラスの包含関係が検査できる機構を実装している[1,3]．

EusLispは，いくつかの点でCommon Lispと非互換である．EusLispのオブジェクト指向は，単一継承であることとメソッドコンビネーションができない点でCLOSとは異なる．また，関数閉包(closure)は，無限エクステントを持てない．さらに，大数(bignum), 有理数，複素数などのデータ型と多値が実装されていない．

一方，Common Lispに対する拡張機能として，幾何計算機能の組み込み，プロセス間通信機能，他言語プログラムインタフェース，Xwindowインタフェース，非同期入出力などがある[2]．すでに障害物回避軌道の生成，動作シミュレーション，動作拘束の導出，把握動作計画などの研究において顕著な成果を上げて来ている[7]．

EusLispは，Sun/Sparcで開発され，SGI/mips, Windows, Linux/i486などにも移植されているが，これから述べる並列機能は，Solaris 2オペレーティングシステムに基づいている．

3．マルチスレッドEusLisp

3.1 Solarisのマルチスレッド機能

文献8)によると，Solarisの定義するスレッドとは，一つのプロセスの中でメモリを共有しながら異なったコンテキストを持ち，並列にCPUを割り当てられる実行単位である．関数をスレッドに割り当てて実行させることで，物理的なCPUの数に依存しない並列プログラムが書ける．スレッドは，他のスレッドと並行してシステムコールを発行できる．スレッドは優先度に従って，時分割あるいは実時間型のスケジューリングを行う．このスケジューリングは先取りかつ非同期であり，マルチプロセッサ機では複数のスレッドが実際に並行に走るので，スレッドの切り替わりを捉えて変数束縛を入れ替えるような方法をとることができない．

スレッド間の同期機構として，Solarisは，mutex_lock, condition_variable, semaphore, reader/writer_lockを提供している．これらのプリミティブの性能をSS10で測定したところ，mutex_lockとunlockの組み合わせでスレッド切換が入らない場合が約2μ秒，セマフォを使ってタスクを切り替えるのに数十μ秒から数百μ秒を要した．

3.2 Lispコンテキストの分割

EusLispの実行には，スレッドのスタックの他に，変数束縛，スペシャル変数束縛，block，catch，flet/labelsを実装するためのスタックが別途必要である．このスタックとそのフレームポインタは，スレッド毎に独立させ，context構造体に格納した．

スタックの分枝を作ることはない．したがって，let, labmdaなどのローカル変数や，block，catch-throw, flet/labelsなどのコンテキストを共有することはない．スレッドが情報を交換するには，シンボルの値や属性リストなどの大域変数を使う．スペシャル変数束縛には，今回は何等の対策も施していない．

3.3 スレッドの割り当て

スレッドとLispのコンテキストは一対一に対応させ，新しく生成されたスレッドに対しては新しいLispのスタックを割り当てる．スレッドの生成は，プロセスの生成に比べれば軽いと言われるが[8]，実際には，(1) Cの制御スタックとLispのバインドスタックの割り当て，(2) スタックオーバフローを検出するためのページ属性の変更，の２種類のコストが余分にかかる．結局，スレッドの生成に要する時間は2-3ミリ秒程度になる．したがって，動的にスレッドを生成・消滅させるよりは，あらかじめ十分な数のスレッドを生成しておき，共有メモリとセマフォを通じてLisp関数の評価を委託する方が効率がよい．生成しておくスレッドの数は，プログラムの性質に応じてユーザが指定する．

3.4 相互排除

コンテキストにまとめて格納できない大域変数のうち，スレッドの間でアクセスが競合するものはmutex_lockによる相互排除が必要である．そのような変数のうち最も深刻なヒープメモリの管理データベースについてはメモリ管理の節で論ずる．

そのほか，オブジェクトのマークビットも共有資源であり，相互排除が必要である．マークビットは，ゴミ集め以外にもcopy-object関数やprintでの循環参照の検出などに用いる．オブジェクトへのメッセージ送信を高速化するメソッドキャッシュは，たかだか数キロバイトの容量であるから，スレッド毎に持たせることにする．

全てのスレッドは，ヒープに割り付けられたオブジェクトを共有する．これらのオブジェクトを同時に複数のスレッドが更新すると，予期しない結果を生じることがある．しかし，全オブジェクトにmutex_lockを付加することは非現実的であるので，オブジェクトへのアクセスの相互排除はプログラマが明示的に行うこととする．

4．メモリ管理

4.1 逐次版EusLispでのメモリ管理

逐次版EusLispは，フィボナッチバディによるメモリ管理を行っている[1,3,6]．ごみ集めはstop & sweep方式である．フィボナッチバディ法は，図１に示すように，各セルの2bitsの管理情報を使ってセルの分割と併合を行い，さまざまな大きさのメモリ要求に答える．バディ法には，ごみ集めによってアドレスが変わらない，メモリ効率が高い，という利点がある．メモリオブジェクトは，種類（クラス）や大きさに無関係に混然となった状態で，論理的に一様なヒープに割り当てられる．

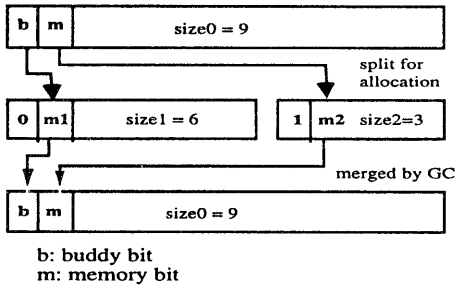

図1　フィボナッチバディメモリ管理

ヒープ中の空きメモリセルは，図2に示すように，buddy_baseと呼ぶグローバルなデータベースのフリーリストにつながれている．メモリの割り付けはalloc関数が処理する．allocは，要求されたセルの大きさを満たす最小の空きメモリを見つけ，必要ならばフィボナッチ数に従った大きさに分割し，buddy_baseを更新する．

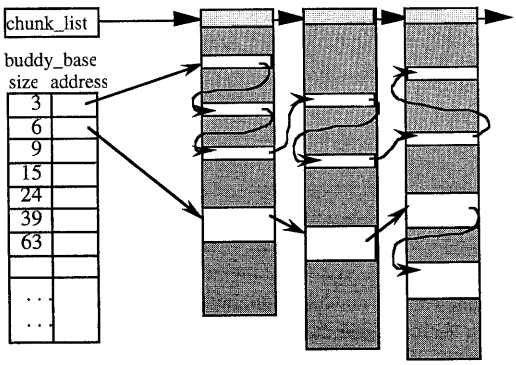

図2　ヒープとフリーリスト管理

4.2 並列版EusLispのメモリ管理

前節のメモリ管理を並列版に適用するには，グローバルデータであるbuddy baseへのアクセスを相互排除しなければならない．具体的には，allocの入り口でmutex-lockをかけ，同時にallocを呼び出せるスレッ

ドをただ一つに制限する．

複数のスレッドがメモリを獲得し，スタックやヒープ中のポインタを書き替えながら走っているとき，あるスレッドのメモリ要求を満たすことができず，突然ごみ集め(GC)が走り出す．他のスレッドは，ゴミ集めの発生に気づかないから，原則として，全ての処理が，いつGCが走っても良い状態を保証しなければならない．文献4)は，alloc直後のセルへのポインタをlast_allocとしてスレッド毎に記録しておき，last_allocをGCのルートに加えることで，安全なGCが実現できると論じている．

5. スレッドの分割メモリ管理

5.1 スレッドへのメモリ管理の委譲

4.2節で述べた方法では，あるスレッドがメモリ要求のためのallocを呼び出すと，他のスレッドのalloc呼び出しは，mutex lockによってブロックされる．mutex-lockは，軽い同期プリミティブではあるが，リストやオブジェクトを大量に生成するプログラムにとっては毎回のmutex lockのコストは無視できない．また，スピンロックの空転によってCPUを浪費する．何よりもmutex-lockによって挿入される待ちは，並列性能を下げる最も大きな要因となる．

この，相互排除を除くことができれば，並列性能を向上させられる．ヒープはスレッドで共有するメモリであり，これをスレッドの数に応じて分割することは意味がない．ヒープの中に格納されるポインタは，他のスレッドが管理するヒープの中を指し示し，そのポインタがさらに他のスレッドに渡されることがあるからである．

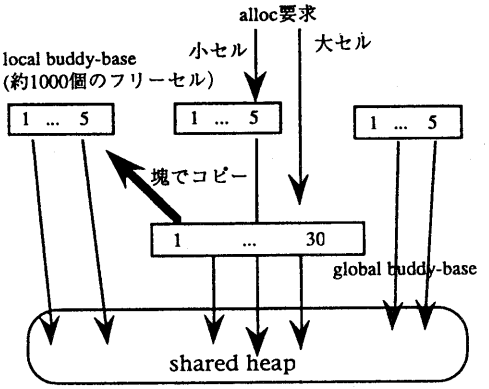

図3: スレッド分割型ヒープ管理

しかし，「フリーリスト」は，スレッドで独立に管理することが可能である．alloc関数が実行する処理は，ほとんどの場合，要求された大きさを満たすセルをフリーリストに見つけて，それをフリーリストから取り除き，返すことである．フリーリストをスレッド毎に用意すれば，この処理は，他のスレッドと独立に実行できる．したがって，図3に示すように，グローバルなフリーリスト(buddy base)を元帳とし，スレッドに一定量のフリーリストの管理を委譲すればよい．

5.2 ローカルフリーリストの大きさ

フリーセルではあるが、一度グローバルなbuddy baseからスレッドのlocal buddy baseに移したセルは、GCにとっては「予約中」としてマークされる。したがって、全フリーセルをスレッドの数で等分したのでは、もしmemory hungryなスレッドがごく少数であった場合に、多くのメモリが利用されないで残る。スレッド間のメモリ使用の不均衡によって大きくメモリ効率が落ちないようにするには、local buddy_baseにつながれるフリーセルの総量は少ないほどよい。一方、local buddy_baseのフリーセルがたとえば100個であれば、100回のallocはローカルに処理され、101回目にグローバルなallocが呼ばれるから、相互排除の頻度を1/100に減ずることができ、これは十分な性能向上になる。

また、グローバルbuddy baseには、最小の大きさ3のフリーセルから、最大約4MWのセルまでの30種類のエントリが用意されているが、16KW以上のフリーセルはほとんど常時空である。さらに、要求されるメモリのサイズは偏っており、幾何モデリングのアプリケーションを走らせた結果では、表1のようなalloの頻度分布が得られている。99%以上の要求が小さい方から5種類のセルサイズに集中しており、local buddy baseには、これら5種類のエントリを持たせるだけで十分な効果が期待できる。

セルサイズ (語)	要求回数	要求メモリ量
3 (cons)	48.4%	30.6%
6 (3D vector)	47.8%	60.5%
9-24	3.7%	7.2%
39-4M	0.01%	1.7%

表1: セルサイズ別メモリ要求の頻度分布

そこで、セルの要求頻度に応じて、表2のような数のフリーセルを各スレッドに渡すこととした。このメモリの総量は、約16KBであり、ヒープの総量5MB、スレッドが32本ある状態で、約1割のメモリがスレッドにローカルに管理されることになる。

インデクス	サイズ	セル数	メモリ量（語）
1	3	500	1500
2	6	300	1800
3	9	20	180
4	15	15	225
5	24	10	240

表2 スレッドに割り当てるフリーメモリ量

スレッドローカルなallocにバディ法をそのまま適用すると、小さなセルの要求にぴったりのサイズのフリーセルがない場合、大きなセルを分割して小さなセルを得ることになる。実験の結果、この方法はかなりの性能劣化をもたらすことが分かった。たとえば、大きさ3のメモリがlocal buddy_baseになくなったとして、サイズ6のセルを分割するよりも、グローバルヒープからサイズ3のフリーセルを大量に譲り受ける方がずっと効率がよいことが観測された。

5.3 GCとの競合

以上の方法で、alloc時にmutex lockによる相互排除を回避され、メモリ管理性能の向上が見られた。小さなメモリの要求のほとんどはスレッドローカルに処理され、使い尽くされるとグローバルヒープからまとまった数のフリーセルを譲り受ける。また、24語以上の大きなセルの要求は、直接グローバルヒープから割り当てられる。これらのグローバルヒープへのアクセスは、以前と同様に相互排除が必要であるが、その頻度は、旧版の1%程度に減っている。

しかし、スレッドローカルにフリーセルを管理するとしても、別のスレッドがGCを実行している間は、local buddy_baseにアクセスするのは危険である。local buddy_baseが、GCのルートになるからである。GCが走っていないことを確認してallocを処理するために、reader/writer lockを使用している。reader/writer lockは、多数のreaderに対してwriterが少数の場合に、readの性能を下げることの少ない相互排除機構である。この場合は、GCをwriterに、localにallocを行う多数のスレッドをreaderに見立てる。

6．性能評価

6.1 ベンチマークプログラム

マルチスレッドEusLispを，32 CPUのCRS社製Cray Superserver 6400 (CS6400) で走らせることで，性能を評価した．並列性能評価には，表3に示す性質を持つ6種類のベンチマークプログラムを用いた．

(a) compfib: コンパイルされたフィボナッチ関数
(b) intfib: インタプリットされるフィボナッチ関数
(c) inner-prod: ベクタの内積
(c) list-replace: リストの転写
(e) copy-seq: リストの複製
(f) body-intcheck: 幾何モデルの干渉検査

	code in cache	local data in cache	sequential access	non-shared data	no memory allocation
compfit	○	○	N/A	○	○
intfib	×	×	N/A	○	○
inner-prod	○	○	○	×	○
list replace	○	○	×	×	○
copy-seq	○	○	×	×	××
body-intcheck	×	×	×	×	×

Table 3: ベンチマークプログラムの性質.

6.4 実行性能の計測

CS6400上で上記のベンチマークプログラムを実行し，1から16の並列度における，経過時間，実行時間の合計，ごみ集め時間を計測した．これらの実測値から、並列度に応じて得られる速度の向上率を並列化利得としてプロットした（図4）。

図4(a)は、allocを相互排除した場合で、メモリ要求の発生しないベンチマークでは、良い並列性能が得られているが、allocを頻繁に発するcopy-seqおよびbody-intcheckでは、並列度を上げるにしたがって性能が劣化している。図4(b)は、この二つのベンチマークについて、前節で述べた方法でスレッドローカルなメモリ管理を取り入れ、mutex-lockをreader/writer lockで置き換えた場合である。他のベンチマークに比べると性能向上の度合いはずっと低いが、図4(a)に比べて顕著な性能の向上が見られる。

しかし、メモリ要求が頻繁に発生するプログラムは、実は、大きな並列利得を上げることが難しい。その理由は、図4(c)に示すように、全計算量に占めるGCの時間が30%から60%にも達し、GCは、逐次的に実行されるからである。GC以外の部分が無限に高速化できたとしても、逐次部分が1/2あれば、全体として2倍以上の高速化はありえない。したがって、これ以上の高速化には、GCをも並列に実行することで、逐次部分を減らす必要がある。

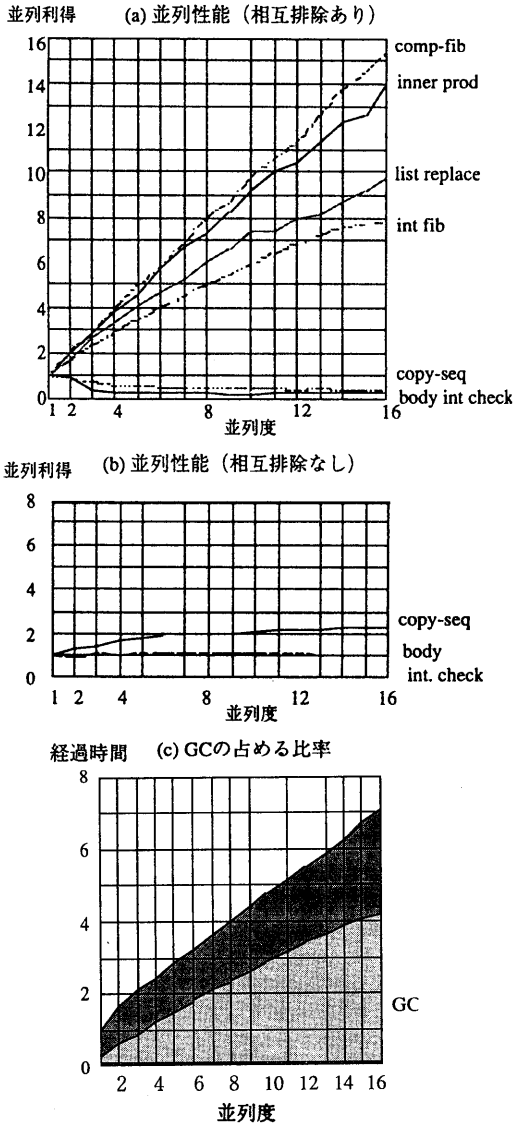

図4 各種ベンチマークの並列化利得

6．結論

メモリの管理をスレッドに分割することで，従来の方法に比べて性能向上が図れることを示したが，GCが逐次的に走る以上，メモリ管理への負担が大きいプログラムでは，CPU数に比例した利得を得ることは難しい．今後，GCの並列化，ゴミを出さないプログラミングの支援，メモリ管理への負担が大きいプログラムとそうでないプログラムの分離，などによる改善の必要がある．

また，並列プログラムのデバッグは，逐次版の何倍も難しい．並列プログラム中で発生するエラーは，同期構造を破壊し，再立ち上げが必要になってせっかくのLispの対話性が失われることもある．対話的な並列プログラム開発環境の充実も急務である．

EusLispのマルチスレッド化は，並列計算と同時に非同期プログラミングも可能にする．自律移動ロボットのbehavior記述を非同期のイベント駆動でガイドするような方法を検討中である．EusLispは，公開ソフトウェアであり，etlport.etl.go.jp (192.31.197.99)からanonymous ftpで入手可能である．

謝辞

並列EusLispを研究する機会を与えてくださった知能システム部築根部長、情報アーキテクチャ部太田部長に感謝いたします．日頃から議論頂くEusLispユーザー，ならびに電総研HPC研究グループの方々に感謝いたします．

参考文献

1) 松井俊浩, 稲葉雅幸: EusLisp: オブジェクト指向に基づくLispの実現と幾何モデラへの応用, 情報処理学会記号処理研究会報告, SIGSYM, Vol. 89-SYM, No. 50-2, (1989).
2) 松井俊浩，原功：EusLisp version 8.0 Reference Manual, 電子技術総合研究所研究速報，ETL-TR-95-19, (1995).
3) Matsui, T. and Inaba, M : EusLisp: an Object-Based Implementation of Lisp", Journal of Information Processing, Vol. 13, No. 3, pp.327-338, (1990).
4) 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, (1995).
5) Steele, G. L. Jr.: Common Lisp the Language, Digital Press, (1984).
6) Peterson, J. L. and Norman, T. A.: Buddy systems, Communication of the ACM, Vol. 20, No. 6, (1977).
7) 松井俊浩: オブジェクト指向型モデルに基づくロボットプログラミングシステムの研究, 電子技術総合研究所研究報告, 第926号, (1991).
8) Multithreading, SunOS 5.3 System Services, pp. 105-134, Sun Soft, (1994).