プログラミング 14−19
（1997. 8. 22）

分散メモリ並列計算機における Reference Count GC と Mark and Sweep GC の比較

山本泰宇　田浦健次朗　米澤明憲
{ymmt,tau,yonezawa}@is.s.u-tokyo.ac.jp
東京大学

要旨

本稿では分散メモリ型の大規模並列計算機上におけるガベージコレクションアルゴリズムのうち Mark & Sweep 方式と Reference Count 方式の 2 手法の実装および特徴を述べ、現実的なアプリケーションにおける速度やメモリ使用率の違いを調べる。非同期的な通信を行うアプリケーションでは Mark & Sweep 方式は Reference Count 方式とほとんど変わらない性能を示し、また同期的な通信を行うアプリケーションでは通信を阻害しない分高速な結果を出した。

A Performance Comparison between Reference Count and Distributed Marking for Global Garbage Collection Scheme on Distributed-Memory Machines

Hirotaka Yamamoto, Kenjiro Taura, and Akinori Yonezawa
{ymmt,tau,yonezawa}@is.s.u-tokyo.ac.jp
University of Tokyo

abstract

This paper describes design and implementations of two garbage collection schemes, Reference Count and Distributed Marking, on large scale distributed-memory computers and compares their memory usage and running time. Both Distributed Marking and Reference Count perform well for asynchronously communicating applications, and Distributed Marking performs better than Reference Count for synchronously communicating applications.

1 始めに

分散メモリ型の並列計算機上で動くようにデザインされた言語では他の Processor Element (PE) 上にあるオブジェクトを別の PE から参照出来るリモートリファレンスと呼ばれる機能を持つことがしばしばある。ガベージコレクション (GC) をこのような参照があるもとで行う場合、他の PE から参照されている可能性のあるオブジェクトは各 PE でローカルに行われる GC だけではゴミとなったかどうか判別出来ない。このようなゴミを取り除く手法としては大別するとオブジェクトがいくつの PE から参照されているかを数えておく *Reference Count* に基くものと、リモートリファレンスが存在する場合、元々のオブジェクトを持つ PE にマークメッセージを送って PE 間でオブジェクトをマークしていく *Distributed Marking* に基くものの 2 通りが提案されてきた [7, 4]。

Reference Count　分散メモリ上で Reference Count を行う際には Weighted Reference Count と呼ばれる手法が用いられるのが普通である。これはリモートリファレンスを作る際に重さ $w(>0)$ を渡すようにする。他の PE はそのオブジェクトをもはや参照しなくなるとオブジェクトを元々持っていた PE に現在自分が持っている重さを返却する。外に出した重さと同じ重さが返却されるとそのオブジェクトは他の PE から参照されていないことが分る。この方式の利点はリモートのオブジェクトをさらに別の PE に参照させる時に、元々そのオブジェクトを持っていた PE にそれを通知することなく現在自分が持っている重さを分け与えるだけで済むことである。

Distributed Marking と比較すると必要な同期は少くて済むか、もしくは全くしなくても済む。その代りにリモートリファレンスを送ったり受け取ったりする操作が Distributed Marking よりも多少コストのかかるものとなる。あるリモートのオブジェクトを参照している PE は、ローカル GC によってそのオブジェクトをもはや参照していないことが分かると、元々そのオブジェクトを持っている PE にそれを通知する。そのため他の PE から参照されているオブジェクトはオブジェクト本体を持っている PE のみでは回収出来ず、他の PE で起こる GC を待たねばならない。またゴミの回収操作の対象となるのは生きている

(live) ものではなく、既に参照されていないオブジェクトである。オブジェクトの回収にはそれを参照している PE のみが関係する [2, 14]。

Distributed Marking ローカルメモリ上でポインタを辿ってマークしていく方法を PE 間に拡張したもので、有効なリモートリファレンスが存在する場合、その参照先の PE で当該オブジェクトをマークするようにメッセージを送る。この PE 間にまたがるマーキングは全 PE が協調して行うフェーズとなるため、リモートリファレンスを持たない PE もこのフェーズにまきこまれる。マーキングの対象となるのは生きているオブジェクトであり、一旦マーク付けが終わると生きているオブジェクトのみが残ることになるので、PE 間にまたがる大きなデータ構造を持つオブジェクトのゴミも一気に回収出来る [5, 8]。

既存の研究では通信や同期のオーバヘッドは大きいと見られてきたため Reference Count の利点が過大に語られてきたが、実装に裏付けされた研究は非常に少かった。すなわち大規模な並列計算機上で Distributed Marking と Reference Count のいずれの方式が本当に良い性能をあげられるかは明かではなかった。

我々はこれまでに同期的な通信を行うようなアプリケーションの性能は、ローカル GC がバラバラに起こるとそれによって通信の待ち時間が増大し結果として計算速度が非常に低下することを示した [13]。すなわち同期を必要としない Reference Count においても、ローカルな GC を全くバラバラに起こしていると同期的な通信を行うアプリケーションにおいては Distributed Marking における同期のオーバヘッドを上回る通信遅延が導入されるであろう。

そこで本研究では並列オブジェクト指向言語 ABCL/f [12] の上に Distributed Marking および Reference Count の 2 種のガベージコレクタを実装し、幾つかのアプリケーションにおけるメモリ使用率や速度に関して調べた。その結果アプリケーションの実行速度全体を考えた場合、Distributed Marking は Reference Count と比較してほとんど変わらない性能であるか、もしくは通信遅延を導入しないで済む分より高速であることが明かになった。

2 関連研究

幾つかの並列言語では Reference Count によって分散メモリマシン上のガベージコレクションを実装している [6]。また六沢らにより Reference Count 方式単体の性能評価も行われている [11] が Distributed Marking との性能比較は行われていない。

Distributed Marking に関する研究の多くは通信が比較的遅いような分散システムや、メッセージの配送が確実に行われないことがあるといった環境を前提としており、実装はほとんどなかった。八杉 [15] や鎌田 [9] は Mark & Sweep 方式に基いた分散ガベージコレクションを実装して評価しているが、一つの実行方式内での評価に留まっている。

田浦 [13] は同期的な通信を行うアプリケーションにおいて、ローカル GC による通信動作の遅れがグローバル GC にかかる時間よりもずっと全体の実行速度に影響を与えることを明かにした。すなわちある PE が別の PE にメッセージを送ってその返事を待つ場合、ローカル GC が各 PE でバラバラに起こると相手が GC 中である確率が高くなり、返事待ちの時間が増す。ローカル GC を全 PE で同時に行うようにすると通信相手が GC 中である確率は低くなり、返事待ちの時間は少くなる。このローカル GC を起こす戦略による返事待ちの時間の違いのほうが、グローバル GC にかかる時間よりもずっと大きいということが明かになった。

3 分散 GC の動作と性質

他の多くの研究と同様我々のガベージコレクタもローカルコレクタの上にグローバルマーカ、もしくはリファレンスカウンタを構成している。ローカルコレクタとしては Boehm による GC ライブラリ [3] を利用している。このライブラリはメモリの割当てに失敗すると、前回の GC から十分な量のメモリ割当てが行われている場合は GC を行い、そうでなければヒープを拡張する。つまり前回の GC でゴミとなったオブジェクトが沢山あればまた GC を行い、まだ生きているオブジェクトばかりであればヒープを拡張する戦略となっている。また ABCL/f ではリモートのオブジェクトへの参照を表すのに *stub* (または *proxy*) と呼ばれる特殊なオブジェクトを用いている。すなわち GC ライブラリが stub オブジェクトをマークするかしないかでリモートリファレンスがまだ有効であるか否かが判定出来る。

3.1 Reference Count

我々は Indirect Reference Count という Weighted Reference Count の一種を実装した。このアルゴリズムでは 1) リモートリファレンスを作る際にある重さを渡し、2) ローカル GC で stub オブジェクトへの参照がなくなったことが判ったら元のオブジェクトを持っている PE に *delete* メッセージを送り、3) delete メッセージを受け取った PE はそのオブジェクトの重さを減らす。重さが 0 になったらもはやそのオブジェクトは他の PE から参照されていないということになる。Reference Count の場合ゴミの回収は他の PE でローカル GC が起こるのを待たないとならないので、ローカル GC について以下の 3 通りの起こし方を実装した。

Independent 各々の PE でバラバラに行う。

Guessing 他の PE でローカル GC が起こった際に回収出来るゴミの量はその PE から参照されているオブジェクトの数に関連すると推測出来る。そこで GC を行う際には自分の持つオブジェクトを参照している数の多い他の PE でも同時に GC を行うようにする。同時に行う PE の数が全体の 4 分の 1 を超える場合は全 PE で同時に行うようにしている。

Synchronous ある PE が GC を行う時は全ての PE で同時に GC を行うようにする。

Reference Count の性質としては同期を必要としない分 GC にかかる時間として現れるコストは小さいが、PE 間にまたがる大きなゴミの回収が遅くなるためヒープサイズが増大すること、また Independent と Guessing の方式

ではバラバラに起こるローカルGCとアプリケーションの実行に割込む delete メッセージの受信によって同期的な通信を行うアプリケーションの通信遅延を増大させる可能性等が考えられる。

3.2 Distributed Marking

Distributed Marking はローカルなオブジェクトのマーキングとPE間にまたがるオブジェクトのマーキングの2つを繰り返して行われる。あるPEが Distributed Marking を起こす時はまずマスターPEにその要求を送る。その後 1) 全てのPEを停止させ、2) 各PEでローカルにマーキングを行い、3) マークされた stub オブジェクトを見つけてその参照先のPEにマークメッセージを送る。2がローカルなマーキングであり、3がPE間にまたがるマーキングである。2、3を繰り返して送るメッセージがなくなると終了である。

ローカルなGCを起こす時にはそのPEのみでバラバラにやってしまうのと、全PEで協調してローカルGCをするのと2通りの戦略が考えられるが、バラバラにやっても性能が顕著に改善されることは無いので [13]、以降ではローカルGCは全PEで同時に起こるものとする。

Distributed Marking ではGCそのものにかかる時間は大きくなるが、PE間にまたがる大きなデータ構造のゴミも一気に取れるためヒープサイズの無闇な増大を防ぐであろうこと、またPE間のゴミ取りにかかるオーバヘッドが一時期に集中するため同期的な通信を行うアプリケーションの通信遅延を増大させることは少いであろうことがその性質として考えられる。

4 実験

Distributed Marking と Reference Count の2種のGC手法の性能を比較するためメモリ分散型の計算機であるAP1000+上で並列オブジェクト指向言語 ABCL/f を用いて書かれたアプリケーションについて実験を行った。

4.1 AP1000+

AP1000+は分散メモリ型の並列計算機である。各PEは50MHzのSuperSparcプロセッサであり、16MBの物理的メモリを有している。我々が実験を行ったシステムではPE数は最大256台である。各PEで動作するオペレーティングシステムはシングルタスクであり、並列計算を行うアプリケーションは全てのCPUリソースとメモリを独占出来る。

4.2 ABCL/f

ABCL/f は並行オブジェクトと関数やメソッドの非同期な呼出しをサポートするオブジェクト指向言語である。並行オブジェクトはリモートリファレンスを用いて複数のPE間で共有できる。また他のPEへの関数やメソッドの呼出しにおいては ***reply channel*** と呼ばれるオブジェクトを作り、返り値を得るために用いる。PE間にまたがるゴミはつまり、リモートリファレンスを一度でも作られた並行オブジェクトと reply channel である。

4.3 アプリケーション

実験にはフィボナッチ数列の並列計算、N体問題の解法とRNAの2次構造の予測を行うものの3つのアプリケーションを用いた。

FIB FIB はフィボナッチ数列を並列に計算する。非常に多くの reply channel を生成する。

RNA RNA[10] は並列組合せ探索によるRNAの2次構造予測プログラムである。PE間の通信はまれであり、しかも各PE内で十分な並列性を有しているため通信の遅延に非常に強い性質を持つ。

BH BH[1] は Barnes と Hut によって提案されたN体問題の並列解法である。SPMDスタイルのアプリケーションであり、各PEはお互いに同期的な通信を頻繁に行う。各PE内では並列性はなく、逐次的な処理が行われるので通信の遅延に強く影響される。各ターンでPE間にまたがるトリー状のデータ構造を作るが次のターンではそのトリーはもはや参照されない。

4.4 実験項目

以下の3点について調べた。

メモリの使用効率 Distributed Marking と異なり Reference Count は基本的にはゴミの回収は他のPEで起こるローカルGCに依存している。またリモートリファレンス毎に重さの情報等を持たねばならない。それらがどのようにヒープサイズに影響するかを調べる。

アプリケーションの実行速度 現実的なアプリケーションにおいてはGCそのものの実行時間以外にも通信動作がすみやかに行われるかが影響することもある。それら全てを含んだアプリケーション全体での実行時間を調べる。

実行速度の内訳 GCや通信動作、計算等でどの程度の時間がかかっているかを調べる。具体的には、計算中 (*busy*)、待ち時間 (*idle*)、ローカルなGCにかかる時間 (*local gc*)、Distributed Marking もしくは Reference Count にかかる時間 (*global gc*)、そして並列化にかかるオーバヘッド (*overhead*) の5種である。Distributed Marking にかかる実行オーバヘッドは先に述べたようにローカルなマーキングとPE間にまたがるマーキングを繰り返して行うだけなので *global gc* にあたる時間はこのマーキングのフェーズのみである。それに対して Reference Count では以下の4つのオーバヘッドがばらばらに現れる。そこで Reference Count にかかる時間はこれらのオーバヘッドを合計したものとする。

- **Stub オブジェクトのマークチェック** ローカルGCの終了後に stub オブジェクトがマークされているかされていないかを見て delete メッセージを送るが、そのためにどのオブジェクトが他のPEのオブジェクトを参照しているかを管理している表中の要素全てについてマークされているかを調べないとならない。

- **Delete** メッセージの送信　delete メッセージはローカル GC の間バッファリングされており、終了後に小出しに各 PE に送られる。
- **Delete** メッセージの処理　delete メッセージを受け取った PE では重さを減らす等の処理を行わないとならない。
- リモートリファレンスの生成のコスト　リモートリファレンスを生成する際に重さを分けたりするコストが Distributed Marking に比べて余計にかかる。

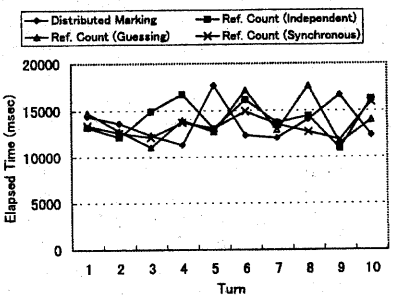
5　実験結果

まず我々は Distributed Marking と Reference Count でリモートリファレンスの生成のコストがどの程度異なるのかを調べた。方法は、ヒープを十分に拡張して GC が起こらないようにした状態でフィボナッチ数列を計算し、かかる時間を比較することで行った。表 1 は 256 台の PE でフィボナッチ数列を 30 まで計算するのにかかった時間である。結果は少くとも我々の実装においてはリモートリファレンスの生成のコストの違いは無視出来ることを意味している。そこで以下の実験結果で Reference Count にかかるコストを計算する際にはこのリモートリファレンスの生成のコストは考えない。

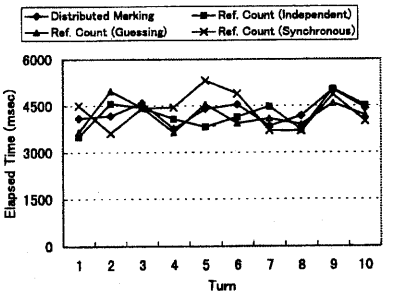
GC Scheme	Time (msec)
Distributed Marking	2356
Reference Count	2284

表 1: GC なしフィボナッチ数列の計算時間

図 1 および図 2 は RNA の計算を 10 回繰り返した時の各回の実行時間とヒープサイズの変化を示している。RNA は実行に不規則な性質を持っているため各回の実行時間はバラつきが多少あるが Distributed Marking および Reference Count のどの方式でも定性的な違いというほどの差は認められない。ヒープサイズに関しては 64 台での実行の際に Distributed Marking と Reference Count の各方式では初期ヒープサイズに隔たりがあるものの Reference Count では繰り返しによってヒープサイズが増大する傾向は無い。また 256 台での実行ではヒープの使用量にほとんど差位は無い。Synchronous な Reference Count が一番小さく収まっているのは頻繁に起こる GC によってゴミが一番良く取れるからであろう。

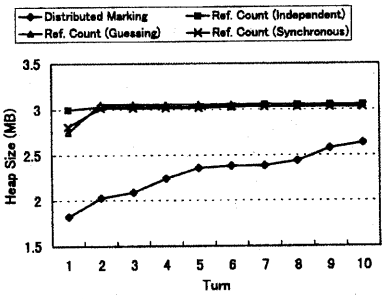


64 台

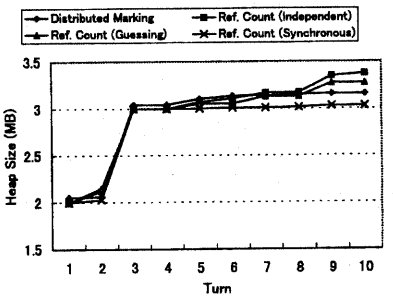


256 台

図 1: RNA の実行時間



64 台



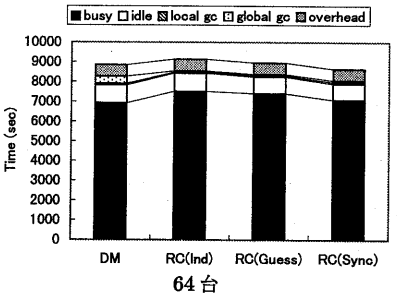
256 台

図 2: RNA のヒープサイズ

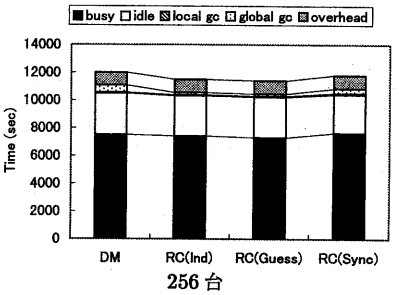
図 3 は RNA の実行時間の内訳を全 PE について集計したものである。これは実行時間の差はほぼ busy である時間の差であることを示している。64 台、256 台いずれでの実行においても Distributed Marking での global gc にかかる時間が多少影響してはいるが全体のパフォーマンスを大きく低下させるものではない。

図 4 および図 5 は N 体問題を 10 ターン実行した時の各ターンの実行時間とヒープサイズの変化を示している。実行時間では Distributed Marking に比べて Reference Count でローカル GC をバラバラ (Independent) に行う方式ではターンが進むにつれて大きく性能が劣るようになる。適当に他のいくつかの PE と同期的にローカル GC を行う (Guessing) 方式でもやや実行時間が遅くなっていく傾向があるが、Independent ほどではない。対して常に全ての PE で同期的にローカル GC を行う (Synchronous)

方式ではほぼ Distributed Marking と同程度の速度で計算している。これは同時期に GC を行うと delete メッセージの送受信も連続して同時期に行われるので通信遅延の増大が少いからである。ヒープサイズの変化を見ると Independent な Reference Count ではターンが進むにつれて大きく増大していく。これは N 体問題では PE 間にまたがる木構造が毎ターン作られるのでゴミとなったデータの回収が遅れるためである。256 台での Synchronous な Reference Count でも同様にヒープサイズが膨らんでいくが原因は木構造をなすゴミが一斉に行う GC では取りきれず、GC ライブラリがガベージコレクションを行うよりもヒープの拡張をするような判断をするようになるからだと思われる。それらに対して Guessing を行う場合はヒープサイズは Distributed Marking と同じ程度に抑え込まれている。Synchronous な Reference Count よりも効果的にゴミを回収出来ているからだと考えられる。

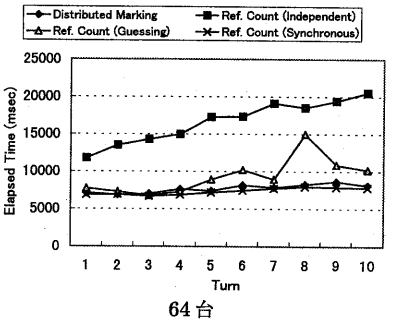


64 台

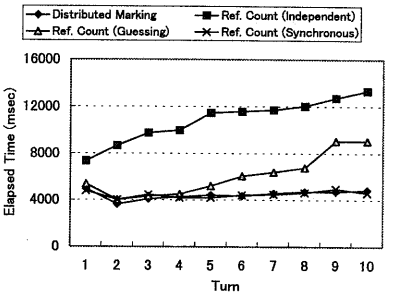


256 台

図 3: RNA の計算の内訳

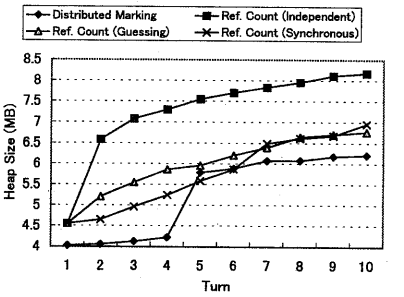


64 台

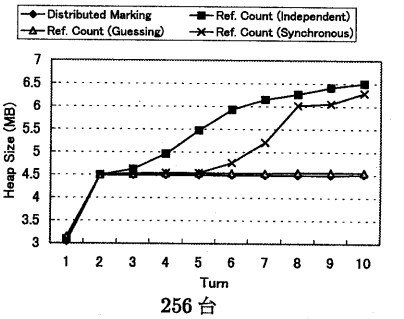


256 台

図 4: N 体問題の実行時間



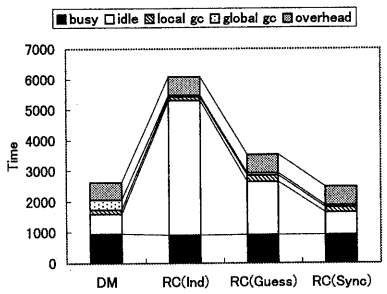
64 台



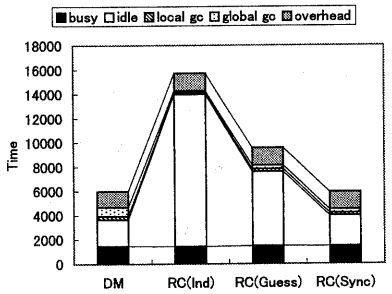
256 台

図 5: N 体問題のヒープサイズ

図 6 は N 体問題の実行時間の内訳を全 PE について集計したものである。これを見ると遅延の原因の大部分は idle となっている時間、つまり通信遅延によるものであることがわかる。Independent な Reference Count の場合 PE 間にまたがる木構造がゴミとしてたまっていくにつれて各 PE でバラバラな GC が細かく起こるようになり、通信遅延が増大する。Guessing でも同様であるが、こちらはある程度まとまった単位でローカル GC が起こることで idle となる時間が少くて済んでいる。グローバルな GC にかかる時間そのものは確かに Distributed Marking が一番多いが、通信遅延の増大と比してアプリケーション全体の性能に与える影響はごく小さいといえる。



64台



256台

図 6: N 体問題の計算の内訳

6 まとめと今後の課題

分散メモリ並列計算機上で PE 間の GC を行う方式として Distributed Marking と Reference Count を実装し、アプリケーションの実行時間およびメモリの使用効率について調べた。Distributed Marking は Reference Count に比較して PE 間で行う GC の時間そのものは大きいものの、アプリケーションの実行速度全体と比較すると無視できる程度であるか、もしくは同期的な通信を行うものの場合はナイーブな Reference Count よりもずっと通信遅延が減らせるため結果的にはむしろ速いということが判った。またメモリの使用効率に対してもゴミの回収が速やかに行えるため Reference Count より悪くなることはなかった。Reference Count はナイーブに実装するとヒープサイズが膨れ上がるものの、適当に他の PE と同期的にローカル GC を行いゴミの回収を積極的に行うようにすればメモリの使用効率がそれほど悪くはならないということも判った。

Reference Count で用いたローカル GC の起こし方のうち、Guessing はアプリケーションに対応してよりアダプティヴな戦略を取るようにすることで N 体問題のように同期的な通信を頻繁に行う場合でも速度の向上が望めるかもしれない。

参考文献

[1] Josh Barnes annd Piet Hut. A hierarchical $o(n \log n)$ force-calculation algorithm. *Nature*, (324):446–449, 1986.

[2] David I. Bevan. Distributed garbage collection using reference counting. In *Parallel Architectures and Languages Europe*, number 258 in Lecture Notes in Computer Science, pages 176–187. Springer-Verlag, 1987.

[3] Hans-Juergen Boehm. Space efficient convervative garbage collection. In *Conference on Programming Language Design and Implementation*, SIGPLAN NOTICES, pages 197–206. ACM, June 1993.

[4] Plainfossé and Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, number 986 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[5] John Hughes. A distributed garbage collection algorithm. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 256–272. Springer-Verlag, 1985.

[6] Nobuyuki Ichiyoshi, Kazuaki Rokusawa, Katsuto Nakajima, and Yu Inamura. A new external reference management and distributed unificatin for KL1. In *New Generation Computing*, pages 159–177, 1990.

[7] Richard Jones and Rafael Lins. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Joen Wiley & Sons, 1996.

[8] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *Proceedings of International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 103–115. Springer-Verlag, 1992.

[9] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In *Proceedings of SuperComputing*, pages 79–88, 1994.

[10] Akihiro Nakaya, Kenji Yamamoto, and Akinori Yonezawa. RNA secondary structure prediction using highly parallel computers. *Computer Applications of Bioscience (CABIOS)*, 1995.

[11] Kazuaki Rokusawa and Nobuyuki Ichiyoshi. Evaluation of remote reference management in a distributed KL1 implementation. In *IPSJ SIG Notes 96-PRO-8 (Proceedings of Summer Workshop on Parallel Processing)*, volume 96, pages 13–18, 1996. (in Japanese).

[12] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/*f* :a future-based polymorphic typed concurrent object-oriented language — its design and implementation. In *Specification of Parallel Algorithms*, DIMACS, pages 275–291, 1994.

[13] Kenjiro Taura and Akinori Yonezawa. An effective garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In *Proceedings of Principles and Practice of Parallel Programming*, SIGPLAN, pages 264–275. ACM, June 1997.

[14] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *Parallel Architectures and Languages Europe*, number 258 in Lecture Notes in Computer Science, pages 432–443. Springer-Verlag, 1987.

[15] Masahiro Yasugi. Evaluation of distributed concurrent garbage collection on a data-driven parallel computer. In *Proceedings of Joint Symposium on Parallel Processing*, volume 97, pages 345–352, May 1997.