Andrei de Araújo Formiga

Algoritmos para Contagem de Referências Cíclicas em Sistemas Multiprocessados

Tese submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Pernambuco como parte dos requisitos para obtenção do grau de **Doutor em Engenharia Elétrica**

Orientador: Prof. Dr. Rafael Dueire Lins

Recife, Maio de 2011.

©Andrei de Araújo Formiga, 2011

Catalogação na fonte
Bibliotecária Margareth Malta, CRB-4 / 1198

F725a Formiga, Andrei de Araújo.
Algoritmos para contagem de referências cíclicas em sistemas multiprocessados / Andrei de Araújo Formiga. - Recife: O Autor, 2011.
170 folhas, il., gráfs., tabs.

Orientador: Prof. Dr. Rafael Dueire Lins.
Tese (Doutorado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Elétrica, 2011.
Inclui Referências Bibliográficas e Apêndices.

1. Engenharia Elétrica. 2. Contagem de referências. 3. Gerenciamento de memória. 4. Dependências cíclicas. 5. Concorrência. 6. Paralelismo. I. Lins, Rafael Dueire (Orientador) II. Título.

UFPE

621.3 CDD (22. ed.) BCTG/2011-201

Universidade Federal de Pernambuco
Pós-Graduação em Engenharia Elétrica

PARECER DA COMISSÃO EXAMINADORA DE DEFESA DE TESE DE DOUTORADO

ANDREI DE ARAÚJO FORMIGA

TÍTULO

"ALGORITMOS PARA CONTAGEM DE REFERÊNCIAS CÍCLICAS EM SISTEMAS MULTIPROCESSADOS"

A comissão examinadora composta pelos professores: RAFAEL DUEIRE LINS, DES/UFPE; VALDEMAR CARDOSO DA ROCHA JÚNIOR, DES/UFPE; JOÃO ALEXANDRE BAPTISTA VIEIRA SARAIVA, DI/Universidade de Minho; FRANCISCO HERON DE CARVALHO JÚNIOR, DC/UFC e RAMIRO BRITO WILLMERSDORF, DEMEC/UFPE, sob a presidência do primeiro, consideram o candidato **ANDREI DE ARAÚJO FORMIGA** **APROVADO**.

Recife, 13 de maio de 2011.

CECÍLIO JOSÉ LINS PIMENTEL
Vice-Coordenador do PPGEE

RAFAEL DUEIRE LINS
Orientador e Membro Titular Interno

JOÃO ALEXANDRE BAPTISTA VIEIRA SARAIVA
Membro Titular Externo

VALDEMAR CARDOSO DA ROCHA JÚNIOR
Membro Titular Interno

FRANCISCO HERON DE CARVALHO JÚNIOR
Membro Titular Externo

RAMIRO BRITO WILLMERSDORF
Membro Titular Externo

Agradecimentos

Esta tese é a culminação não só dos quatro anos de doutorado, mas de todo um projeto, iniciado desde o mestrado, de realizar uma mudança de carreira e receber formação de pesquisador. Agora que o final se aproxima, acredito que posso considerar esse projeto bem sucedido, e por isso devo agradecer às pessoas que me ajudaram e foram importantes ao longo dessa jornada.

Meu orientador, Rafael Dueire Lins, aceitou me orientar e suportou minhas idiossincrasias durante todos esses anos, sem nunca faltar com o apoio, não apenas acadêmico, mas que foi além das obrigações de um orientador.

Os membros da banca no meu exame de qualificação, professores Valdemar Cardoso da Rocha Jr., Francisco Heron de Carvalho Jr. e João Saraiva, dedicaram tempo a este trabalho e trouxeram contribuições, ideias e discussões que ajudaram a melhorar a qualidade do documento final desta tese.

As amizades feitas desde o mestrado continuaram importantes nos anos do doutorado, mesmo que as mudanças naturais na vida de todos (novos empregos, mudanças de cidade, casamentos) tenham nos distanciado do convívio diário. Em especial Márcio Lima, Bruno Ávila e André Ricardson estiveram sempre próximos, mesmo que não fisicamente.

O convívio com os colegas professores do Departamento de Ciências Exatas da UFPB têm se mostrado uma fonte de estímulo e de ideias para novos projetos, além de uma união e companheirismo que tornam todo o trabalho mais agradável.

Meus pais foram fundamentais nesse projeto de mudança de vida, assim como tudo que veio antes. Não só eles sempre me estimularam aos estudos e a aprender sobre tudo que eu me interessei, como me apoiaram no momento em que decidi deixar um emprego promissor para iniciar no mestrado, pois era essa a direção que eu pretendia para a minha vida.

Finalmente, e não menos importante, minha namorada e companheira de vida, Gio, enfrentou comigo as dificuldades e os rigores do trabalho de doutorado, e contribuiu com a manutenção do meu equilíbrio e minha satisfação geral.

A todos, muito obrigado.

Andrei de Araújo Formiga

Universidade Federal de Pernambuco

13 de Maio de 2011

Resumo da Tese apresentada à UFPE como parte dos requisitos necessários para a obtenção do grau de Doutor em Engenharia Elétrica

Algoritmos para Contagem de Referências Cíclicas em Sistemas Multiprocessados

Andrei de Araújo Formiga

Maio/2011

Orientador: Prof. Dr. Rafael Dueire Lins
Área de Concentração: Comunicações
Palavras-chaves: contagem de referências, gerenciamento de memória, dependências cíclicas, concorrência, paralelismo.
Número de páginas: 170

O gerenciamento automático da memória dinâmica, conhecido como *coleta de lixo* (*garbage collection*), se tornou uma necessidade na maioria das linguagens de programação em uso atualmente. Dentre as técnicas para realizar o gerenciamento automático da memória, a contagem de referências se mostra vantajosa por uma série de razões, dentre elas o fato de ser uma técnica naturalmente incremental, o que evita a parada completa do processo do usuário para realizar tarefas de administração da memória. A natureza incremental da contagem de referências indica que o algoritmo pode ser adaptado para uma versão em sistemas multiprocessados, mantendo sua característica não suspensiva no programa do usuário. Entretanto, os problemas causados pela necessidade de sincronização entre *threads* pode anular os ganhos de eficiência obtidos com essa extensão, inviabilizando o uso de um coletor de lixo baseado em contagem de referências em um ambiente multiprocessado. Nesta tese apresenta-se um conjunto de algoritmos eficientes para utilizar a contagem de referências em sistemas com vários processadores, tendo como foco principal o algoritmo para contagem de referências para vários mutadores e um coletor. Este algoritmo foi implementado na máquina virtual

Java Jikes RVM e seu desempenho testado em relação a coletores alternativos; os resultados de tais testes indicam que o algoritmo proposto tem desempenho competitivo com outros coletores similares, mas impõe tempos de pausa muito menores ao programa do usuário. A presente tese também apresenta uma prova informal da corretude do algoritmo proposto.

Abstract of Thesis presented to UFPE as a partial fulfillment of the requirements for the degree of Doctor in Electrical Engineering

Algorithms for Cyclic Reference Counting on Multiprocessor Systems

Andrei de Araújo Formiga

May/2011

Supervisor: Prof. Dr. Rafael Dueire Lins
Area of Concentration: Communications
Keywords: reference counting, memory management, cyclic dependencies, concurrency.
Number of pages: 170

Automatic memory management, or *garbage collection*, has become a necessity for the runtime systems of most current programming languages. One of the main techniques for garbage collection is reference counting, which has been shown to present many advantages to the alternatives. One of the main advantages is that it is a naturally incremental technique, that never stops the processing of user instructions to complete memory management tasks. This incremental nature of reference counting is an indication that it can be adapted to multiprocessor systems and maintain its advantage of avoiding to suspend the process of the user. However, experience has shown that problems caused by the synchronization of user process and collector can severely degrade the performance of the system, making reference counting an undesirable choice for a garbage collector in a multiprocessor system. In this thesis a series of new algorithms for garbage collection on multiprocessor systems is presented, all of them based on the technique of reference counting. The final algorithm in the series supports multiple user processes executing concurrently with the collector. This algorithm was implemented in the Jikes RVM Java Virtual Machine and tested, using a set of benchmarks based on real Java applications, against well established collectors for multiprocessor systems.

The results show that the performance of the algorithm based on reference counting has competitive throughput characteristics in relation to the alternatives, while at the same time presenting much lower latency characteristics; in most cases, the reference counting algorithm has latency about two orders of magnitude lower. The thesis also presents an informal proof of correctness for the new algorithms.

Lista de Tabelas

Sumário

Lista de Figuras

Lista de Algoritmos

capítulo 1

Introdução

Sistemas computacionais com múltiplos processadores, normalmente na forma de múltiplos núcleos de processamento acondicionados em um único *chip* (os chamados *chips multicore*) se tornaram comuns nos últimos anos. Computadores utilizados por usuários domésticos hoje, em sua maioria, possuem de dois a seis núcleos de processamento; em servidores, o número de núcleos tende a ser maior. Mesmo em dispositivos embarcados e dispositivos móveis, a tendência atual é que os processadores utilizados passem a ter múltiplos núcleos.

Essa mudança na arquitetura dos computadores cria a necessidade não só de aproveitar a nova capacidade computacional disponível, como também adaptar os sistemas de *software* utilizados para que funcionem de maneira adequada nesse novo ambiente. Uma parte importante das plataformas de *software* atuais é o *coletor de lixo*, componente responsável pelo gerenciamento automático da memória dinâmica. As características do coletor de lixo utilizado podem afetar o desempenho de programas que executam em sistemas com múltiplos processadores. Por exemplo, uma estratégia ainda comum de coleta de lixo é parar a execução de todos os *threads* do programa do usuário para realizar o processo de coleta (essa estratégia é conhecida como *stop-the-world*). Ao parar todos os *threads* do programa do usuário, todos os processadores disponíveis ficam ociosos, menos um processador que executará o processo de coleta. Esse desperdício da capacidade computacional presente leva a uma degradação no desempenho dos programas.

Por outro lado, algoritmos de coleta de lixo que funcionem correta e eficientemente em ambientes multiprocessados são mais complexos, mais difíceis de implementar e mais difíceis de serem provados corretos. Esta tese tem como objeto de estudo o gerenciamento automático

eficiente da memória em sistemas multiprocessados, usando algoritmos baseados na técnica da contagem de referências cíclicas. Tomando como base algoritmos previamente existentes para sistemas com um ou múltiplos processadores, o trabalho relatado nesta tese expande esse repertório apresentando dois novos algoritmos de contagem de referências cíclicas em sistemas multiprocessados, aplicáveis a plataformas de *software* atuais. Um desses dois algoritmos tem alto desempenho e baixa latência quando executado em sistemas computacionais com múltiplos processadores.

Além da capacidade de multiprocessamento, os computadores continuam vendo aumentada sua capacidade de memória com o passar do tempo. O tamanho do espaço de memória utilizável pelos programas é cada vez maior, mas algumas estratégias de coleta de lixo populares têm seu desempenho proporcional ao tamanho do espaço disponível, enquanto as técnicas baseadas em contagem de referências não têm tal dependência – o desempenho de um coletor baseado em contagem de referências é proporcional ao tamanho da memória ocupada, não da memória total disponível. Os testes realizados com o algoritmo proposto demonstram que seu desempenho sofre menor impacto com o aumento do espaço de memória disponível, em relação às estratégias alternativas de coleta.

Essas duas características – bom desempenho em sistemas multiprocessados e com espaços de memória crescentes – tornam os algoritmos baseados em contagem de referência desejáveis no ambiente de servidores, onde essas características são importantes.

Os novos algoritmos propostos foram implementados e testados em plataformas adequadas; um deles foi implementado em uma máquina virtual Java, e testado nessa plataforma usando uma carga de trabalho realista composta por programas de vários níveis de complexidade escritos em Java. A corretude também é um parâmetro importante para algoritmos de coleta de lixo, e esta tese apresenta argumentos de corretude e uma prova informal de corretude para o algoritmo implementado na máquina virtual Java.

1.1 Gerenciamento automático da memória

A memória é um dos recursos principais que devem ser gerenciados em um sistema computacional. No nível do sistema operacional, a memória é organizada, na maioria dos casos, em sistemas de memória virtual [92]. Nesse caso, cada processo recebe um espaço linear de memória, endereçado de maneira independente do espaço de memória de outros processos. Isso possibilita a cada processo gerenciar os endereços de memória de maneira simples, e

também garante o isolamento entre os processos de programas diferentes, que não podem interferir com a memória usada por outros.

Entretanto, cada processo individual em programas atuais pode precisar gerenciar centenas de milhares ou até mesmo milhões de objetos na sua memória. Alguns desses objetos podem seguir uma disciplina simples de ocupação da memória, como objetos estáticos – que ficam alocados na memória durante toda a execução do processo – ou objetos alocados em pilha. Mas nem todos os objetos manipulados pelos programas seguem essas disciplinas; alguns terão um tempo de vida difícil de determinar estaticamente, e terão de ser gerenciados como memória dinâmica durante a execução.

O gerenciamento da memória dinâmica em algumas linguagens imperativas como C e C++ é realizado pelo próprio programador[1], mas a experiência demonstrou que isso causa uma grande quantidade de erros de programação. Embora objetos possam ser alocados pelo programa a qualquer momento, ter de determinar quando eles se tornam livres e podem ser liberados – tornando a memória que ocupavam disponível para outros objetos – pode ser difícil em programas de grande complexidade ou com certas estruturas de dados dinâmicas. Para determinar quando objetos em memória se tornam inúteis – ou *lixo* – e liberar a memória ocupada por eles para outros usos, surgiram os *coletores de lixo*, componentes do sistema de tempo de execução do programa responsáveis por essa tarefa.

A praticidade dos coletores de lixo e a redução no número de erros de programação que pode ser observada quando eles são usados tornaram a técnica popular. De fato, a grande maioria das linguagens de programação que se tornaram populares nos últimos 10 anos utiliza alguma forma de gerenciamento automático de memória dinâmica; por exemplo: Java [5], Ruby [73], Python [95] e C# [47].

Os algoritmos para coleta de lixo, em sua maioria, podem ser divididos em três classes de técnicas desenvolvidas para esse fim:

- ▷ A contagem de referências [22, 72];
- ▷ A coleta por cópia [19, 33, 82];
- ▷ O algoritmo de marcação e varredura (*mark-sweep* ou *mark-scan*) [78].

Este trabalho tem como objeto de estudo algoritmos baseados em contagem de referências para sistemas multiprocessados.

[1] Existem coletores de lixo para as linguagens C e C++ que funcionam como bibliotecas adicionais da linguagem, como o coletor conservador de Boehm (às vezes chamado de coletor Boehm-Demers-Weiser) [17], mas por padrão essas linguagens contam apenas com gerenciamento manual da memória.

1.2 Coleta de Lixo em Sistemas Multiprocessados

O objetivo do coletor de lixo, como visto, é dividir a memória de um programa em duas classes: as partes que ainda estão sendo utilizadas, e as que podem ser reaproveitadas para outros fins. Em sistemas tradicionais uniprocessados, esta tarefa é normalmente feita intercaladamente com as operações do resto do programa, já que só há um processador para executar as instruções.

Embora a maioria dos computadores em uso sempre tenha sido baseada em um único processador, são realidade há bastante tempo os sistemas multiprocessados. Uma extensão natural da técnica de coleta de lixo tradicional para sistemas multiprocessados é fazer com que o coletor seja executado concorrentemente com o resto do programa, e de fato isso vem sendo estudado há décadas. Um dos primeiros trabalhos sobre gerenciamento automático e concorrente de memória data de 1975 [90]. Em tempos recentes mesmo os computadores de uso doméstico se tornaram multiprocessados (vide a seção 1.4), e a pesquisa em métodos de gerenciamento automático da memória nesse tipo de ambiente computacional se torna cada vez mais importante.

1.3 Objetivos

O estudo relatado nesta tese investiga novos algoritmos baseados em contagem de referências para sistemas multiprocessados de memória compartilhada. Os pontos de partida são os algoritmos sequenciais [69, 72] e os algoritmos já desenvolvidos para sistemas multiprocessados [39, 68]. Um dos novos algoritmos usa uma estrutura similar aos anteriores, permitindo apenas um *thread* mutador (*thread* executando programa do usuário), enquanto que o segundo permite qualquer número de mutadores. Este último é implementado na máquina virtual Java Jikes RVM [1] e seu desempenho é testado em um conjunto de aplicações que representam uma carga de trabalho realista para a plataforma Java. Além disso, considerações sobre a corretude dos novos algoritmos propostos são desenvolvidas.

1.3.1 Questões de Pesquisa

O projeto de pesquisa realizado no trabalho de doutorado aqui relatado teve como guia um conjunto de perguntas de pesquisa que serviram para a definição dos objetivos e elaboração de hipóteses. As perguntas de pesquisa relacionadas são:

1. É possível transportar as características de processamento incremental e baixa latência, presentes nas versões sequenciais do algoritmo de contagem de referências, para uma arquitetura multiprocessada?
2. Um dos problemas relatados em uma das primeiras tentativas de implementação da contagem de referências em sistemas paralelos, feita por DeTreville [27], foi que o excesso de operações de alteração dos contadores de referências, e a sincronização necessária para proteger essas alterações, degradava o desempenho do sistema a níveis inaceitáveis. É possível ter um sistema multiprocessado baseado em contagem de referências com desempenho equiparável a coletores de rastreamento?
3. Como seria o desempenho de uma versão para multiprocessadores do algoritmo de contagem de referências na execução de aplicações reais?
4. Como o desempenho da contagem de referências multiprocessada se altera com o aumento do número de processadores disponíveis?

O presente trabalho descreve as atividades de investigação que buscaram responder essas questões, e seus resultados.

1.4 Motivação

Desde os primeiros tempos da computação, a conexão de vários processadores em paralelo para realizar uma máquina com maior poder de processamento é vista como uma forma de obter mais eficiência do projeto de computadores. Por várias vezes os cientistas da computação previram que os computadores paralelos se tornariam a regra, e não uma exceção relegada a nichos de mercado, e em todas as vezes essas previsões não se cumpriram. Já em 1967 Gene Amdahl escreveu ([2], *apud* [87]):

> Por mais de uma década os analistas anunciam que a organização de um único computador alcançou seus limites e que avanços verdadeiramente significantes só podem ser feitos pela interconexão de uma multiplicidade de computadores de tal modo que permita solução cooperativa... Demonstrou-se a continuada validade do método de processador único...

No momento atual essas previsões se concretizaram com a popularização das arquiteturas baseados em múltiplos núcleos de processamento por *chip*, os chamados processadores *multi-núcleo* ou *multicore* [85]. Com a integração em escala cada vez maior, a estratégia tradicional

de aumentar a freqüência de operação para aproveitar o maior número de transistores em uma única pastilha passou a resultar em menores ganhos de desempenho. Tornou-se atrativo, então, integrar vários processadores em uma única pastilha, provendo maior poder de processamento sem aumentar a freqüência de operação. Esta é a estratégia adotada atualmente pelos maiores fabricantes de unidades centrais de processamento para computadores, como a Intel e a AMD.

Atualmente, mesmo os computadores domésticos são multiprocessados, restando uniprocessadores apenas em sistemas legados ou sistemas embarcados (mesmo nestes últimos, existe uma tendência recente de passar a usar processadores multi-núcleo). A grande questão que se coloca hoje é quanto tempo o *software* demorará para acompanhar o *hardware*, já que a programação de sistemas concorrentes sempre foi menos estudada e menos compreendida que nos sistemas seqüenciais. A dificuldade para que os programadores utilizem adequadamente os recursos de um computador multi-núcleo tem chamado a atenção tanto na indústria de *software* quanto nos meios de pesquisa; já em 2005 se comentava em algumas situações que uma revolução nas técnicas de programação estaria prestes a acontecer [91].

Um candidato óbvio para aproveitar o paralelismo disponível é o sistema de coleta de lixo. Este é um trabalho que é feito automaticamente, sem a intervenção explícita do programador, e que opera em funções que são independentes do resto do programa. Aproveitar o paralelismo no gerenciamento de memória pode tornar qualquer programa automaticamente mais eficiente em arquiteturas paralelas, sem a necessidade de esforço por parte dos programadores. Trata-se então de uma contribuição imediata a esta situação de transição, ajudando os sistemas de software a aproveitar a capacidade de processamento disponível.

Além da capacidade de melhor utilização dos recursos computacionais disponíveis, realizar a coleta de lixo de maneira integrada com um sistema multiprocessado também evita que o desempenho de programas que aproveitem os múltiplos processadores seja degradado. Executar um programa escrito para vários processadores em uma plataforma que usa um coletor de lixo *stop-the-world* pode negar completamente as vantagens de ter tornado o programa concorrente ou paralelo.

Qualquer sistema concorrente de gerenciamento automático de memória pode tirar vantagem dos novos processadores multi-núcleo, mas os métodos baseados na contagem de referências possuem algumas vantagens, como será visto no Capítulo 2.

1.5 Contribuições

As principais contribuições desta tese são as seguintes:

- ▷ Um novo algoritmo para contagem de referências em sistemas multiprocessados usando um coletor e um mutador (Seção 4.3.2), baseado no conceito de fila de atualizações. Este algoritmo foi implementado e mostrou desempenho consistentemente superior aos algoritmos anteriores que usam a técnica da contagem de referências.
- ▷ Um novo algoritmo para contagem de referências adequado para implementação em várias plataformas atuais que seguem o modelo de *threads*, como a máquina virtual Java e similares (Capítulo 5). Esse algoritmo foi implementado em uma máquina virtual Java criada para pesquisa (Jikes RVM) e demonstrou desempenho competitivo em relação às alternativas, ao mesmo tempo que impõe uma latência muito menor do que essas alternativas.

1.6 Organização da tese

Esta proposta de tese está organizada em sete capítulos, incluindo a presente Introdução.

O Capítulo 2 trata do gerenciamento de memória, apresentando um curto histórico e uma análise superficial das três principais técnicas de gerenciamento automático de memória; neste ponto se estabelecem as vantagens e desvantagens da contagem de referências com relação às outras técnicas, sendo a sua principal deficiência a não-recuperação de estruturas cíclicas. A solução para este problema são os métodos para contagem de referências cíclicas, cuja versão mais recente é apresentada no Capítulo 3; o algoritmo neste capítulo está na sua forma seqüencial, para sistemas uniprocessados.

O Capítulo 4 apresenta os algoritmos para contagem de referências cíclicas em sistemas multiprocessados, começando com o algoritmo de Formiga e Lins [39] e propondo dois novos algoritmos para esse tipo de ambiente. Nesse capítulo também são sugeridas algumas estratégias para estender o algoritmo para ambientes com grandes números de processadores.

No Capítulo 5 são descritos a implementação do algoritmo de contagem de referências cíclicas em na máquina virtual Java Jikes RVM, e os testes de desempenho realizados nessa implementação.

O Capítulo 6 considera a corretude dos algoritmos propostos, apresentando uma prova informal do algoritmo implementado na Jikes RVM. O Capítulo 7 apresenta as considerações

finais e sugestões para trabalhos futuros.

capítulo 2

Gerenciamento de Memória e Coleta de Lixo

A memória é um dos principais recursos que devem ser gerenciados em um computador. A lei de Moore [83] prevê que o número de transistores em circuitos integrados (de tamanho e custo fixo) dobra a cada 18 meses, fato que garantiu que a quantidade de memória disponível nos computadores crescesse em progressão geométrica. Juntamente com a quantidade de memória disponível, cresceu também a complexidade das aplicações criadas para esses computadores. Do aumento de complexidade das aplicações seguiram-se maiores exigências para o gerenciamento de memória, principalmente com relação à eficiência temporal e a praticidade para o programador. De início, o programador tinha que administrar toda a memória utilizada por seu programa; entretanto, com o tempo, as tarefas de gerenciamento da memória foram sendo automatizadas e relegadas aos sistemas de tempo de execução das linguagens de programação, livrando os programadores deste trabalho. Isso andou lado a lado com o aumento de capacidade computacional: os computadores tornaram-se multitarefa e multiusuário, com sistemas operacionais cada vez mais complexos. A capacidade computacional adicional garantiu que mais poder de processamento estivesse disponível para a automatização das tarefas de gerenciamento da memória.

Uma das tarefas que pode ser automatizada é detectar quando um item de dados usado pelo programa não é mais necessário, possibilitando que a memória ocupada por este item seja reutilizada pelo programa para guardar outras informações. O componente que realiza esta tarefa é chamado de *coletor de lixo*. Embora os primeiros sistemas de tempo de execução para

a linguagem Lisp já utilizassem coletores de lixo no final da década de 1950 [22], por muito tempo o uso de um coletor de lixo automático foi considerado ineficiente, e muitas das linguagens de programação mais utilizadas na indústria de software em décadas passadas, como C e C++, não incluem suporte a coletores de lixo automáticos[1]. Entretanto, a experiência tem confirmado que os programadores se tornam mais produtivos, e os programas criados mais robustos, quando podem utilizar um sistema de gerenciamento automático de memória. Isso se reflete no fato da maioria das linguagens de programação surgidas nos últimos anos, assim como as plataformas de aplicação dominantes na indústria de software (Java e .NET), incluírem um coletor de lixo que livra o programador do trabalho de gerenciamento manual da memória.

Neste capítulo é apresentado um panorama geral sobre o gerenciamento de memória, enfocando principalmente as técnicas de coleta de lixo sequencial – para computadores uniprocessados. Primeiro, definem-se os conceitos necessários e a notação utilizada. Seguem-se então um breve histórico da evolução do gerenciamento de memória e uma descrição resumida das três principais técnicas para gerenciamento automático, finalizando com a versão clássica do algoritmo de contagem de referências; a técnica de contagem de referências é a base para os algoritmos para sistemas com multiprocessadores, objeto de estudo desta tese. Quando são apresentadas as vantagens e desvantagens do algoritmo de contagem de referências, atenção especial é dada ao problema no tratamento de estruturas cíclicas; a solução é descrita no Capítulo 3.

2.1 Definições e Notação

A memória RAM (*Random Access Memory*) de um computador é um conjunto de *bytes* organizados de tal forma que cada *byte* individual está associado a um endereço único; o valor de cada *byte* pode ser lido ou alterado livremente, sendo necessário apenas conhecer seu endereço para isso[2].

O objetivo do gerenciamento da memória é diferenciar as partes da memória que estão em uso – ou seja, *ativas* – das partes que não estão ativas. Estas últimas podem ser reaproveitadas para outros usos. O gerenciamento da memória em linguagens de programação normalmente

[1] É possível utilizar um coletor de lixo externo, não presente na biblioteca padrão das linguagens C e C++. Um exemplo é o coletor Boehm-Demers-Weiser [17].

[2] Na maioria das arquiteturas de computador atuais, não é possível ler ou escrever apenas um *byte* isolado, sendo a palavra a menor unidade de transferência de ou para a memória. Entretanto, os endereços de memória continuam sendo contados em *bytes*. A capacidade de acessar *bytes* isolados não é importante para o presente trabalho

divide o espaço de memória em três sub-espaços que seguem disciplinas de alocação e liberação bem definidas: a memória *estática* permanece alocada durante toda a execução do programa, a memória local segue uma disciplina de pilha (*Last-In First-Out*), e a memória dinâmica ou *heap* guarda partes da memória que podem ser alocadas em momentos arbitrários. O gerenciamento da memória estática e da memória local são mais simples, mas as partes da memória dinâmica não utilizadas pelo programa mas que ainda estão marcadas como ocupadas são denominadas de *lixo*. O subsistema do programa (ou sistema de tempo de execução) que realiza o gerenciamento automático da memória dinâmica, reaproveitando partes não utilizadas, é comumente conhecido pelo nome de *coletor de lixo*, e o próprio processo de gerenciamento é chamado de *coleta de lixo*. Quando algumas partes da memória dinâmica não estão em uso mas não são identificadas como lixo, e ficam inacessíveis para o programa pelo resto do seu tempo de vida, diz-se que ocorreu um *vazamento de memória* (*space leak*).

Mais interessante para o gerenciamento de memória no contexto das linguagens de programação é interpretar a memória como ocupada por *objetos* de algum programa. Um objeto é uma estrutura de dados que ocupa uma região definida da memória, ou seja, um conjunto definido de *bytes*. Uma *variável* é uma entidade do programa usada para fazer referência a um objeto. Objetos podem ser *atômicos*, caso não possam ser analisados e separados em partes, ou *compostos* (também chamados de *estruturados*) caso sejam formados por composição de outros objetos; estes componentes, chamados de *campos*, podem ser atômicos ou compostos. Um tipo de objeto atômico particular é um *ponteiro* (também chamado de *referência*). No modelo considerado aqui, um ponteiro contém como valor o endereço de algum outro objeto na memória; se o ponteiro P possui como valor o endereço do objeto O, diz-se que P *aponta para* O. Um valor especial para ponteiros, chamado de *nil*, é usado quando nenhum objeto é apontado, ou seja, P tem valor *nil* quando não existe um objeto O tal que P aponta para O. Um ponteiro pode fazer parte de algum objeto composto, e assim diz-se que um objeto O_1 aponta para um outro objeto O_2 quando O_1 tem como componente um ponteiro para O_2. Pode-se dizer também neste caso que O_1 referencia O_2. Se existe uma sequência não vazia de ponteiros que leva do objeto O_i em O_j, diz-se que O_j está no *fecho transitivo* de O_i, ou que O_j está *transitivamente conectado* a O_i.

Essa organização da memória em objetos que podem apontar para outros objetos é representada por um grafo direcionado no qual os objetos são os nós e existe uma aresta de O_1 para O_2 se O_1 aponta para O_2. Este grafo é chamado de *grafo da memória* e captura o estado

da memória em um momento de tempo específico. À medida em que o programa executa, referências podem ser alteradas, criadas ou removidas, alterando assim o grafo da memória.

Uma parte ativa da memória é aquela que está ocupada por algum objeto em uso pelo programa. Um objeto O só pode estar em uso se ele for referenciado por algum outro objeto que esteja em uso, ou seja, se algum objeto ativo O' contém um ponteiro para O. Obviamente, esta cadeia de ponteiros deve começar em algum lugar, e este lugar é o chamado *conjunto raiz*[3] – um conjunto de objetos a partir do qual, seguindo-se as referências em ponteiros, pode-se chegar a qualquer objeto ativo no programa.

Achar os objetos ativos, então, corresponde a determinar que nós no grafo da memória estão transitivamente conectados a algum nó do conjunto raiz. Os nós que não estiverem conectados ao conjunto raiz são lixo e podem ser reaproveitados. Todas as técnicas criadas para o gerenciamento da memória realizam este processo, de alguma forma.

2.2 Histórico

Aqui é apresentado um breve histórico sobre a evolução das técnicas de gerenciamento de memória. Este resumo histórico tem apenas a intenção de explicitar os motivos por trás do desenvolvimento e uso das técnicas de gerenciamento automático. Partes do conteúdo desta seção foram baseadas no livro de Jones e Lins [53], onde o leitor encontrará maiores detalhes sobre o assunto.

Na história do desenvolvimento das linguagens de programação repetem-se casos do seguinte padrão:

1. Uma tarefa de programação é inicialmente feita manualmente;
2. Técnicas para automatizar a tarefa são desenvolvidas, mas julgadas computacionalmente ineficientes para que seu uso se torne comum. Os programadores julgam que o custo computacional de usar as técnicas automáticas não compensam a maior produtividade ganha com elas;
3. Com o avanço no projeto de semicondutores, aumenta-se tanto o poder computacional quanto a memória disponível para os computadores, o que altera a relação de custo e benefício para as técnicas automáticas. Com a evolução dessa tendência, uma tarefa que anteriormente era feita de maneira manual ou explícita passa a ser automatizada

[3] Em linguagens de programação o conjunto raiz inclui os ponteiros presentes na pilha (que contém os registros de ativação) e objetos estáticos.

porque julga-se que os custos se tornaram baixos o suficiente em relação aos requisitos de desempenho das aplicações.

O gerenciamento da memória seguiu esse padrão: inicialmente feito apenas manualmente, técnicas para automatizá-lo logo surgiram, mas permaneceram ignorados por décadas por grande parte dos programadores, que julgavam ineficientes os métodos automáticos. Com o tempo, a relação entre custo e benefício mudou e hoje a maior parte dos programadores utilizam algum sistema que inclui o gerenciamento automático de memória. As etapas mais importantes nesta evolução são resumidamente contadas aqui.

Nos primeiros tempos da computação toda a comunicação entre programador e máquina era bit-a-bit, com chaves simples para entrada e LEDs para saída. Pouco tempo depois, a introdução de dispositivos simples de entrada e saída tornou mais simples a troca de valores hexadecimais entre operador e máquina. O próximo passo foi permitir que os programadores usassem códigos mnemônicos que eram mecanicamente traduzidos para a notação binária esperada pelo computador. Ainda assim, os usuários eram responsáveis por cada detalhe da execução de seus programas. Por exemplo, uma atenção especial era necessária para contar o número de palavras de memória usadas pelo programa e encontrar o endereço absoluto das instruções para determinar se havia espaço disponível para carregar o programa e para especificar o endereço de destino em instruções de desvio.

Pelo final dos anos 40 e início da década de 50, essa tarefa de gerenciamento foi transferida para os códigos macro e as linguagens de montagem (*assembly*). Programas simbólicos são mais fáceis de escrever e entender que programas de linguagem de máquina, primariamente porque códigos numéricos para endereços e operadores são substituídos por códigos simbólicos que carregam mais significado para um usuário. Ainda assim o usuário precisava se preocupar intimamente com a forma que um computador específico operava, e como e onde os dados eram representados dentro da máquina. O grande número de pequenos detalhes específicos da máquina que devem ser gerenciados torna, até hoje, a programação em linguagem de montagem uma tarefa extenuante.

Para vencer estes problemas, ideias para linguagens de programação de alto-nível apareçeram da metade para o fim da década de 40, com o objetivo de tornar a programação mais simples. Em 1952 apareceram os primeiros compiladores experimentais, e o primeiro compilador FORTRAN foi lançado no início de 1957 [8]. Um compilador para uma linguagem de programação de alto nível deve alocar recursos da máquina alvo para representar os objetos de dados

manipulados pelo programa do usuário. Existem três diferentes disciplinas para alocação de memória em computadores:

- ▷ alocação estática;
- ▷ alocação na pilha;
- ▷ alocação dinâmica (ou no *heap*).

Cada uma delas é detalhada nas subseções seguintes.

2.2.1 Alocação estática

A política de alocação mais simples é a de alocação estática. Todos os nomes no programa são associados a localizações na memória em tempo de compilação: estas associações não mudam em tempo de execução. Isto implica que as variáveis locais de um procedimento são associadas às mesmas localizações em cada ativação do procedimento. A disciplina de alocação estática foi usada exclusivamente nas primeiras versões da linguagem FORTRAN, e ainda era a única disciplina utilizada na linguagem Fortran 77[4] e em algumas versões da linguagem de programação paralela Occam [62]. Muitas linguagens de programação em uso atualmente incluem a disciplina de alocação estática para gerenciar alguns tipos de objetos do programa (por exemplo, variáveis globais em C e atributos marcados como *static* em Java). A alocação estática possui três limitações:

- ▷ o tamanho de cada estrutura de dados deve ser conhecida em tempo de compilação;
- ▷ nenhum procedimento pode ser recursivo já que todas as suas ativações compartilham as mesmas localizações para as variáveis locais;
- ▷ estruturas de dados não podem ser criadas dinamicamente.

Apesar destes problemas, a alocação estática tem duas vantagens importantes: eficiência e segurança. Implementações de linguagems que utilizam alocação estática são geralmente eficientes no gerenciamento da memória, pois nenhuma estrutura de dados dinâmica precisa ser criada, mantida ou destruída durante a execução do programa. Como todas as localizações de memória são conhecidas durante a compilação, os dados podem ser acessados diretamente ao

[4] Embora muitos compiladores para a linguagen Fortran 77 incluíssem suporte a chamadas de procedimento recursivas, abandonando assim o uso exclusivo da alocação estática, isto não era previsto no padrão oficial da linguagem. A primeira versão da linguagem cujo padrão determina que seja possível chamar procedimentos recursivos foi a Fortran 90.

invés de indiretamente por ponteiros. Isso pode ser particularmente importante em sistemas embarcados dotados de processadores mais limitados. A outra vantagem está relacionada à segurança do programa: nenhuma falha pode ocorrer por falta de memória durante a execução, já que os requerimentos de memória são conhecidos antecipadamente. Devido a tais características, algumas aplicações de missão crítica podem utilizar a alocação estática como única estratégia de gerenciamento da memória. Entretanto, na maioria das aplicações atuais, outras estratégias são utilizadas em conjunto com a alocação estática.

2.2.2 Alocação na pilha

Duas características das linguagens de programação contribuíram para a proliferação da disciplina de alocação em pilha: as chamadas recursivas de procedimentos, e o escopo léxico em blocos. As primeiras linguagens estruturadas em blocos apareceram em 1958 com Algol-58 e Atlas Autocode. As linguagens estruturadas em blocos eliminam algumas dificuldades da alocação estática reservando espaço em uma pilha. Um *registro de ativação* é empilhado na pilha do sistema cada vez que um procedimento é chamado, e desempilhado quando o procedimento retorna. Como a seqüencia de chamadas de procedimentos em um programa em execução segue uma estrutura similar a um percurso em profundidade em uma árvore de chamadas, a estrutura dos registros de ativação na pilha é ideal para guardar, para cada procedimento, os objetos que são locais a ele no registro de ativação correspondente. A organização em pilhas tem cinco conseqüências:

- ativações diferentes de um procedimento não compartilham as mesmas localizações de memória para as variáveis locais. Chamadas recursivas são possíveis, desta forma aumentando significativamente o poder expressivo da linguagem;
- o tamanho das estruturas de dados locais como vetores podem depender de parâmetros passados para o procedimento;
- os valores de variáveis locais alocados na pilha não podem persistir de uma ativação para outra do mesmo procedimento;
- o registro de ativação de um procedimento chamado não pode estar ativo por mais tempo que o registro do procedimento chamador;
- apenas um objeto cujo tamanho é conhecido em tempo de compilação pode ser retornado como o resultado de um procedimento.

2.2.3 Alocação dinâmica

Embora as linguagens estruturadas em blocos tenham resolvido muitos dos problemas existentes com a alocação estática, ainda assim era impossível criar estruturas de dados com tempo de vida arbitrário e independente de procedimentos específicos do programa. Linguagens cujas estruturas de dados eram eminentemente dinâmicas, como Lisp e suas listas, usavam uma organização de memória chamada de *heap*, que era adequada para alocação de liberação de partes da memória em momentos arbitrários durante sua execução.

Para obter o mesmo tipo de flexibilidade para o programador, algumas linguagens como C e Pascal introduziram a possibilidade de alocar dados dinamicamente em um *heap*. Diferente da disciplina "o último a entrar é o primeiro a sair" de uma pilha, dados presentes em um *heap* podem ser alocados e desalocados em qualquer ordem. A alocação dinâmica possui uma série de vantagens:

- ▷ o projeto de estruturas de dados pode incluir naturalmente estruturas recursivas como listas e árvores, dando-as representações concretas;
- ▷ o tamanho das estruturas de dados não precisa ser fixo, podendo variar dinamicamente. Exceder os limites pré-estabelecidos de estruturas de dados como vetores é uma das causas mais comuns de falhas em programas;
- ▷ objetos de tamanho dimensionado dinamicamente podem ser retornados por procedimentos;
- ▷ muitas linguagens de programação permitem que procedimentos sejam retornados como resultado de outros procedimentos – isso é bastante comum em linguagens de programação funcional, por exemplo. Linguagens que usam alocação na pilha podem fazer isso se proibirem procedimentos aninhados: o endereço estático do procedimento retornado é usado (esta é a abordagem por trás dos ponteiros para função na linguagem C, por exemplo). Linguagens de programação funcional e de alta ordem podem permitir que o resultado de uma função seja uma *suspensão* ou *fechamento*: uma função armazenada juntamente com um *ambiente* que especifica associações de nomes com localizações de memória. Estas associações, portanto, têm um tempo de vida independente da função que as criou.

Atualmente muitas – se não todas – linguagens de alto nível permitem a alocação dinâmica de memória tanto na pilha quanto no *heap*. Muitas linguagens procedurais e imperativas, como

C e Pascal, deixaram a tarefa de gerenciar os dados alocados dinamicamente para o programador; ele deve especificar explicitamente quando uma região de memória alocada no *heap* pode ser liberada. A linguagem C++ seguiu este mesmo caminho, para manter a compatibilidade com a linguagem C. Já as linguagens funcionais, lógicas e a maioria das linguagens orientadas a objetos utilizam o gerenciamento automático de memória. Exemplos de linguagens com gerenciamento automático incluem Scheme [32], Objective Caml [18], Haskell [93], Prolog [21], Java [5], C# [46], Ruby [73], Python [95] e JavaScript [36]. Algumas outras linguagens, como Modula-3 [45] e versões mais recentes de Objective-C [57], oferecem a possibilidade de gerenciamento automático ou manual.

A alocação manual de dados no *heap* pode ocasionar uma série de problemas relacionados ao tempo de vida dos dados. Um dos problemas é deixar de liberar uma região de memória que não é mais utilizada; outro é continuar acessando uma região que já foi liberada e pode estar sendo ocupada por dados diferentes. Além disso, a própria estrutura do programa se torna mais complexa e difícil de manter pela necessidade de sempre controlar o tempo de vida das estruturas de dados. É um fato aceito atualmente na comunidade de desenvolvimento de *software* que o gerenciamento manual da memória deve ser utilizado apenas em um número limitado de casos, como por exemplo quando o controle total sobre a memória é realmente requerido, ou quando é necessário obter um desempenho específico para o programa que não pode ser obtido com gerenciamento automático, e mesmo esses casos são motivo para discussão.

2.3 Técnicas de gerenciamento automático de memória

Como visto na seção anterior, a administração do tempo de vida dos objetos alocados dinamicamente é o maior problema no gerenciamento de memória. Objetos alocados estaticamente ficam ativos durante toda a execução do programa, e não há preocupação em reaproveitar o espaço ocupado por eles; para os objetos alocados na pilha, seu tempo de vida é determinado pelo tempo de execução de algum procedimento, o que permite ao compilador gerar código para liberar o espaço ocupado por tais objetos quando eles não forem mais necessários – pois o procedimento que os contêm terminou sua execução.

O problema, então, é saber quando liberar o espaço dos objetos alocados dinamicamente. Em linguagens com gerenciamento manual da memória, o programador especifica explicitamente quando o espaço de um objeto deve ser liberado. Por exemplo, na linguagem C utiliza-se

a função `free`. Para automatizar essa tarefa, é necessário identificar que objetos ainda estão em uso, e portanto acessíveis através do conjunto raiz, e quais não são mais necessários e podem ter seu espaço liberado. Ao longo da história do desenvolvimento do gerenciamento automático de memória surgiram três técnicas principais, cujas variantes são utilizadas até hoje, seja de forma direta ou misturadas em sistemas híbridos. São elas:

- ▷ marcação e varredura (*mark & sweep*);
- ▷ coleta por cópia;
- ▷ contagem de referências.

As subseções seguintes se ocupam da descrição destas três técnicas, incluindo especificações do seu funcionamento e comparativos de vantagens e desvantagens relativas de cada uma. Para simplificar a descrição, assume-se que os objetos no *heap* são todos de mesmo tamanho; tais objetos uniformes podem ser chamados de *células*. Com esta organização, as células livres do *heap* podem ser convenientemente organizadas em uma lista encadeada, chamada de *lista livre* (*free list*). Admite-se que existem duas subrotinas simples disponíveis para o sistema de gerenciamento de memória: `allocate` retorna uma nova célula retirada da lista livre, e `free` inclui a célula passada como parâmetro para a lista livre, liberando-a para reutilização.

Os algoritmos usados para ilustrar e especificar a implementação de cada uma das técnicas são expressos em uma linguagem algorítmica genérica, com propriedades e semântica similares às de um subconjunto de uma linguagem procedural imperativa como C ou Pascal. Como notação adicional para estes algoritmos, os campos de um objeto composto são referenciados, no texto do algoritmo, na forma *Obj.campo*: o nome da variável que é a referência ao objeto *Obj*, seguido por um ponto e o nome do campo em questão. Por exemplo, se um objeto referenciado pela variável R possui campos chamados *nome* e $telefone$, a notação $R.nome$ indica uma referência para o campo *nome* do objeto referenciado por R. Nos algoritmos descritos, o campo com nome *children* de um objeto O representa o conjunto de objetos apontados por O, e o campo *size* representa o tamanho do objeto, em *bytes*.

Para mais detalhes sobre as três técnicas descritas abaixo e os algoritmos utilizados referencia-se o livro de Jones e Lins [53].

2.3.1 Marcação e Varredura

O primeiro algoritmo para reciclagem automática de memória foi uma técnica de *rastreamento*: o método de Marcação e Varredura (*mark-sweep* ou *mark-scan*), desenvolvido por J.

McCarthy em 1960 para a linguagem Lisp [78]. Esta é uma técnica de rastreamento pois os objetos que são acessíveis a partir do conjunto raiz são explicitamente rastreados, partindo-se dos objetos que são raízes e seguindo-se as referências em ponteiros. Desta forma, o método de marcar-e-varrer é baseado em separar, através de um processo de varredura ou rastreamento, os objetos no *heap* nas duas classes de objetos ativos e inativos, e posteriormente separar os inativos para reaproveitamento. O momento de realizar essa varredura é normalmente quando o programa requer a alocação de um novo objeto e o sistema de gerenciamento de memória determina que não há mais memória disponível, o que dispara um *ciclo de coleta*, durante o qual fica suspenso o processo do usuário. O algoritmo de alocação está no Algoritmo 2.1.

Algoritmo 2.1 Alocação no algoritmo marcar-e-varrer

```
1: procedure NEW
2:     if freeList is empty then
3:         MARK-SWEEP()
4:     end if
5:     newCell := ALLOCATE()
6:     return newCell
7: end procedure
```

O algoritmo Um ciclo de coleta é composto por duas fases: marcação e varredura, como mostrado no algoritmo da Algoritmo 2.2. Na marcação, as células acessíveis a partir da raiz são *marcadas* como em uso; na varredura, as células que não estão marcadas são recolhidas. Para guardar o estado de marcação, cada célula possui um campo – que pode ser apenas um bit – reservado, o *campo de marcação*, que recebe o nome de MARK. O fim da fase da marcação é identificado quando não existem mais células acessíveis que não tenham sido marcadas (veja o Algoritmo 2.3). Depois que a marcação termina, todo o *heap* é varrido linearmente e as células que não estão marcadas são identificadas como livres e podem ser reaproveitadas, ou seja, incluídas na lista livre. O algoritmo está detalhado como Algoritmo 2.4.

2.3.2 Coleta por cópia

O primeiro coletor usando a técnica de cópia foi proposto por Marvin Minsky em 1963 [82]; o algoritmo de cópia de Minsky foi pensado para uso com armazenamento externo. Posteriormente, com o advento de sistemas operacionais com memória virtual, Fenichel e Yochelson [33] propuseram o uso de um algoritmo de cópia, baseada na ideia de Minsky, mas para gerenciar o armazenamento em memória interna. Os algoritmos de Minsky e o de Fenichel e Yochelson

Algoritmo 2.2 Algoritmo principal para coleta marcar-e-varrer

```
1: procedure MARK-SWEEP
2:     for each R in Roots do
3:         MARK(R)
4:     end for
5:     SWEEP()
6:     if freeList is empty then
7:         ABORT("Nao foi possivel alocar memoria: heap esgotado")
8:     end if
9: end procedure
```

Algoritmo 2.3 O algoritmo de marcação

```
1: procedure MARK(N)
2:     if N.mark = unmarked then
3:         N.mark := marked
4:         for each M in N.children do
5:             MARK(M)
6:         end for
7:     end if
8: end procedure
```

Algoritmo 2.4 O algoritmo de varredura

```
1: procedure SWEEP
2:     N := heapBottom
3:     while N < heapTop do
4:         if N.mark = unmarked then
5:             FREE(N)
6:         else
7:             N.mark := marked
8:         end if
9:         Avança o ponteiro N para apontar para próximo objeto no heap
10:    end while
11: end procedure
```

são recursivos, o que implica na necessidade de ter espaço disponível para uma pilha quando o sistema vai realizar a coleta; entretanto, a coleta de lixo é tipicamente ativada quando o sistema não tem mais memória disponível. Cheney [19] propôs uma versão eficiente do algoritmo de cópia que não é recursivo e não sofre dos mesmos problemas das versões anteriores. Os coletores que usam a técnica de cópia quase sempre seguem o modelo do algoritmo de Cheney.

A coleta por cópia é outra técnica de rastreamento, tendo portanto similaridades com a coleta marcar-e-varrer. O funcionamento do algoritmo, entretanto, é diferente: a ideia é que o *heap* é dividido em duas partes, chamadas de *semi-espaços*; em um dado momento, apenas um deles é utilizado. Quando não há mais memória disponível no semi-espaço em uso, o coletor identifica todos os objetos ativos, copiando-os para o outro semi-espaço. Depois disso, o papel dos dois semi-espaços é trocado até o próximo ciclo de coleta. Tradicionalmente os dois semi-espaços recebem os nomes de espaço de origem (*from space*) e espaço de destino (*to space*). O espaço de destino é sempre o semi-espaço que está em uso, enquanto que o espaço de origem não é utilizado e contém sempre os objetos anteriores ao momento da última coleta. O algoritmo de inicialização para o coletor de cópia é mostrado em 2.5; $toSpace$ e $fromSpace$ são os espaços de destino e origem, respectivamente; $free$ é um ponteiro para o primeiro endereço livre para alocação.

O algoritmo Inicialmente, os papéis dos dois espaços são trocados pela rotina FLIP. Cada célula acessível a partir das raízes é copiada do $fromSpace$ para o $toSpace$ pela rotina COPY. É necessário ter um cuidado com as referências que restam durante o processo de cópia para preservar a topologia do grafo da memória, pois caso contrário algumas células poderiam ser copiadas mais de uma vez. A forma usual de evitar esse problema é através de *endereços de encaminhamento* (*forwarding address*). Em cada objeto que é copiado deixa-se um endereço de encaminhamento no $fromSpace$ que aponta para o novo endereço deste no $toSpace$. Sempre que uma célula no $fromSpace$ for visitada durante a cópia, o algoritmo verifica se ela já foi copiada, e neste caso utiliza o endereço de encaminhamento armazenado; caso contrário, a célula é copiada para o $toSpace$ e o endereço de encaminhamento é armazenado. Note-se que o endereço é armazenado *antes* da cópia efetiva – isso assegura a terminação do algoritmo e a preservação da topologia do grafo. O endereço de encaminhamento é guardado na própria célula, sendo desnecessário o uso de espaço adicional na memória para isso; no algoritmo, assume-se que ele é armazenado em um campo chamado $forward$.

Algoritmo 2.5 Inicialização e alocação na coleção por cópia

```
1:  procedure INIT
2:      toSpace := heapBottom
3:      spaceSize := heapSize/2
4:      topOfSpace := toSpace + spaceSize
5:      fromSpace := topOfSpace + 1
6:      free := toSpace
7:  end procedure
8:  procedure NEW(n: Integer)
9:      if free + n > topOfSpace then
10:         FLIP()
11:     end if
12:     if free + n > topOfSpace then
13:         ABORT("Nao foi possivel alocar memoria")
14:     end if
15:     newCell := free
16:     freeattrfree + n
17:     return newCell
18: end procedure
```

Algoritmo 2.6 A rotina de troca de espaços

```
1:  procedure FLIP
2:      fromSpace, toSpace := toSpace, fromSpace        ▷ troca valores dos ponteiros
3:      topOfSpace := toSpace + spaceSize
4:      free := toSpace
5:      for each R in Roots do
6:          R := COPY(R)
7:      end for
8:  end procedure
```

Algoritmo 2.7 Algoritmo de coleta por cópia

```
1:  procedure COPY(P)
2:      if IS-ATOMIC(P) ∨P = nil then          ▷ se P é atomico, ele é uma folha do grafo
3:          return P
4:      end if
5:      if P.forward = nil then
6:          n := P.size
7:          P' := free                          ▷ obtém memória do espaço de destino
8:          free := free + n
9:          P.forward := P'
10:         Copia conteúdo de P para espaço alocado em P'
11:     end if
12:     return P.forward
13: end procedure
```

2.3.3 Contagem de referências

A contagem de referências é um método direto, no qual um objeto é identificado como inativo no momento em que a última referência para ele deixa de existir. A ideia do algoritmo é simples: ter em cada objeto um campo adicional para manter o número de objetos que o referenciam, ou seja, um contador de referências. Quando este contador chega ao valor zero em um objeto O, isso significa que não há mais referências para O, e a memória ocupada por ele pode ser reaproveitada imediatamente. A maior virtude deste algoritmo é a simplicidade para determinar se uma célula está ativa ou não. Também é uma técnica naturalmente incremental, distribuindo a sobrecarga do gerenciamento de memória durante toda a execução do programa.

O invariante deste algoritmo é que o contador de referências de uma célula C tenha valor igual ao número de referências originárias de outras células ativas que apontam para C; o gerenciador de memória deve garantir a manutenção desse invariante. Para isso, é necessário alterar o valor do contador em cada operação com ponteiros. Assim como no algoritmo de marcação e varredura, as células livres que podem ser alocadas são mantidas em uma estrutura de lista encadeada chamada de *lista livre*.

O algoritmo Células livres estão originalmente agrupadas em uma estrutura chamada de lista livre (*free list*). Quando uma nova célula é alocada, ela é retirada da *free list* e ligada transitivamente à raiz; a partir daí, cada vez que uma referência é criada ou alterada para apontar para esta célula, seu contador é incrementado. Da mesma forma, cada referência destruída ou alterada para não mais apontar para a célula causa uma redução de 1 no valor

do contador. Se o contador chegou a zero, a célula se encontra inacessível e pode ser retornada à lista livre.

O Algoritmo 2.8 mostra como é realizada a alocação de células na contagem de referências. A função de baixo nível ALLOCATE obtém uma célula livre da lista, enquanto que NEW usa ALLOCATE e inicializa a célula obtida; o contador de referências é acessado através do campo rc. A rotina CREATE é utilizada quando uma nova referência é criada para o objeto T. Os algoritmos para remoção e alteração de ponteiros são mostrados em 2.9; UPDATE altera o local apontado por R para apontar para S, ajustando as referências de acordo, enquanto DELETE é usada quando uma referência para o objeto T é destruída; a rotina FREE é chamada por DELETE para mover uma célula inativa para a lista livre.

O programa que utiliza o sistema de contagem de referências deve utilizar as rotinas NEW, CREATE, DELETE e UPDATE sempre que for necessário operar com ponteiros, para garantir que o invariante da contagem de referências seja mantido.

Algoritmo 2.8 Alocação de células e criação de ponteiros na contagem de referências

```
1:  function ALLOCATE
2:      newCell := freeList
3:      freeList := freeList.next
4:      return newCell
5:  end function
6:  procedure NEW
7:      if freeList =nil then
8:          ABORT("Nao foi possivel alocar memoria")
9:      end if
10:     newCell := ALLOCATE()
11:     newCell.rc := 1
12:     return newCell
13: end procedure
14: procedure CREATE(T)
15:     T.rc := T.rc + 1
16: end procedure
```

2.3.4 Comparações entre as técnicas

Após apresentar o funcionamento das três técnicas principais para o gerenciamento automático da memória, comparamos agora as suas características, destacando as vantagens e desvantagens relativas de cada uma.

Algoritmo 2.9 Remoção e alteração de ponteiros na contagem de referências

```
1:  procedure FREE(N)
2:      N.next := freeList
3:      freeList := N
4:  end procedure
5:  procedure DELETE(T)
6:      T.rc := T.rc − 1
7:      if T.rc = 0 then
8:          for each U in T.children do
9:              DELETE(U)
10:         end for
11:         FREE(T)
12:     end if
13: end procedure
14: procedure UPDATE(R, S)
15:     DELETE(∗R)
16:     S.rc := S.rc + 1
17:     ∗R = S
18: end procedure
```

2.3.4.1 Marcação e Varredura

As técnicas que utilizam varredura – marcação e varredura e coleta por cópia – têm duas vantagens principais sobre a contagem de referências: a primeira é que ciclos são recuperados naturalmente, sem necessidade de medidas especiais; a segunda é que nenhuma sobrecarga é imposta às operações com ponteiros. Por outro lado, a coleta nestas técnicas impõe uma parada completa do programa principal (do usuário) enquanto o coletor trabalha – na coleta marcar-e-varrer, esse trabalho consiste na marcação das células e na varredura completa do *heap*. Com o aumento do tamanho das memórias, verificou-se que programas que utilizavam um coletor de marcação e varredura gastavam uma porcentagem significativa do tempo de execução no coletor [53]. Além disso, a pausa causada pela varredura pode ser indesejável ou inaceitável em certos sistemas, como os de tempo-real.

Entretanto, se o tempo de resposta não é crucial, esta técnica pode oferecer um desempenho superior ao da contagem de referências. Mesmo assim, o custo da coleta é alto: cada célula ativa é visitada durante a fase de marcação, e todas as células do *heap* são visitadas durante a varredura. Portanto, a complexidade assintótica do algoritmo é proporcional ao tamanho do *heap*. Outros problemas de desempenho são a fragmentação da memória e a

perda da localidade de referências associada a isso; o procedimento de varredura prejudica o desempenho em sistemas com hierarquias de memória, causando *thrashing*. Ainda outro problema é que o desempenho de um coletor de marcação e varredura depende da porcentagem de ocupação do *heap*, pois quanto mais ocupado mais freqüentes serão as coletas; já a contagem de referências não sofre degradação pelo aumento da ocupância.

Em relação à coleta por cópia, a técnica de marcação e varredura tem a vantagem principal de utilizar todo o *heap*, enquanto que a coleta por cópia divide-o em dois semi-espaços, como será visto na próxima seção; além disso, o algoritmo de marcação e varredura não sofre de cinetose (*motion sickness*), uma vez que as células em uso não são deslocadas durante a coleta. Por outro lado, coletores de marcação e varredura causam uma maior fragmentação do *heap* em relação aos coletores por cópia, já que estes últimos compactam a memória. A fragmentação promove uma piora na localidade de referência na memória ocupada, degradando o desempenho. Além disso, a coleta por cópia não precisa varrer a memória toda.

Resumindo, as principais vantagens da técnica de marcação e varredura são:

- ▷ utiliza toda a memória;
- ▷ precisa de apenas um bit por objeto ou célula;
- ▷ não impõe nenhum custo nas operações de ponteiros;
- ▷ não sofre de cinetose (*motion sickness*).

Já suas desvantagens mais acentuadas são:

- ▷ tempo de pausa para varredura e coleta;
- ▷ varre todo o espaço de endereçamento do processo, perdendo a localidade de referências;
- ▷ fragmenta o espaço de memória.

2.3.4.2 Coleta por cópia

As vantagens da coleta por cópia sobre a contagem de referências e a coleta marcação e varredura a tornaram amplamente usada. Custos de alocação são extremamente baixos; a fragmentação é eliminada pela compactação da estrutura de dados ativos. O custo da alocação nas outras técnicas é muito maior, especialmente se objetos de tamanhos variáveis forem utilizados.

O custo mais imediato da coleta por cópia é o uso de dois semi-espaços: o espaço de endereçamento necessário é o dobro em comparação aos coletores que não utilizam cópia. Pode-se argumentar que o uso de memória virtual elimina o problema; mesmo assim, o dobro do espaço de memória deve ser alocado e reservado para o programa. Ao mesmo tempo, esses custos devem ser pesados contra as vantagens da compactação. Em sistemas atuais, onde a diferença de desempenho entre as CPUs e as memórias é crescente, ter uma boa localidade de referência nos dados utilizados pelo programa é cada vez mais importante. Por outro lado, o algoritmo de coleta por cópia sofre de *motion sickness*, uma vez que as células copiadas mudam de endereço; tal característica pode causar dificuldades em alguns tipos de aplicação.

Resumindo, temos como vantagens desta técnica:

- ▷ evita fragmentação por copiar os objetos;
- ▷ tempo de coleta menor que o de marcação e varredura;
- ▷ não necessita de espaço extra para cada célula, pois o endereço de encaminhamento pode ser armazenado em algum outro campo.

Já como desvantagens, podemos citar:

- ▷ perde muito tempo copiando objetos;
- ▷ pode causar muitas falhas no sistema de memória virtual durante o processo de cópia entre semi-espaços;
- ▷ sofre de cinetose (*motion sickness*).

Com relação à complexidade temporal do processo de coleta, ela é proporcional ao tamanho do grafo de objetos na memória, e não ao tamanho do *heap*, como no caso do algoritmo marcação e varredura. Portanto, em situações de baixa ocupância, em que o *heap* está pouco ocupado, o algoritmo de coleta por cópia tende a ser mais eficiente em relação ao algoritmo de marcação e varredura.

2.3.4.3 Contagem de referências

A principal vantagem da contagem de referências é distribuir a carga do gerenciamento de memória ao longo da execução do programa, sem a necessidade de parar toda a computação útil para isso. Em comparação, as outras técnicas indiretas como marcação e varredura suspendem o programa do usuário enquanto buscam células inativas que podem ser coletadas.

Isso pode ser uma vantagem em sistemas que precisem manter um tempo de resposta garantido, como sistemas de tempo-real. Entretanto, a carga de gerenciamento é distribuída de maneira não uniforme – a remoção de uma célula pode ter custo proporcional ao tamanho do sub-grafo sob a célula, por exemplo. Por este motivo, é possível que as pausas em um sistema de contagem de referências podem se tornar comparáveis às das outras técnicas [16], embora isso não seja comumente verificado em situações reais.

Estudos empíricos em várias linguagens apontam que a maioria das células não são compartilhadas e ficam ativas por pouco tempo [53, 61]. O algoritmo de contagem de referências permite o reaproveitamento imediato dessas células, minimizando a ocupação do programa do usuário e ajudando a evitar que a memória seja paginada no disco[5]. Além disso, esse conhecimento de células que podem ser reaproveitadas imediatamente pode ser utilizado em otimizações, quando uma nova célula é similar a uma que está sendo liberada. Por fim, a liberação imediata de células inativas é útil para garantir que ações de *finalização* em objetos sejam determinísticas [15].

Mesmo com essas vantagens, o algoritmo possui uma série de problemas que inibiram seu uso. A desvantagem mais séria é o alto custo sofrido nas atualizações de ponteiros. Nas técnicas indiretas as operações com ponteiros não possuem custos adicionais. Outro impacto sobre o desempenho temporal do programa ocorre por causa da liberação imediata de células inativas e a recursividade na liberação das células – ver a rotina DELETE no Algoritmo 2.9. Suponha que uma célula C seja a raiz de uma estrutura de dados com grande número de células, ou seja, todos os caminhos do conjunto raiz para qualquer célula da estrutura passam por C. No momento que C se torna inativa, toda a estrutura também estará inativa e pode ser liberada; entretanto, isso torna o tempo ocupado com a liberação maior do que seria para liberar apenas uma célula. De fato, como o sub-grafo sob uma célula qualquer é potencialmente ilimitado, não se pode prever qual o tempo gasto durante a liberação recursiva, impedindo o uso desta versão do algoritmo para sistemas de tempo real, por exemplo. Existem técnicas bem conhecidas para contornar tal problema, como a *liberação procrastinada* de Weizenbaum [97].

Outra desvantagem é o alto grau de acoplamento entre o coletor e o resto do programa. Em sistemas de contagem de referências, a existência de uma *barreira de escrita* – código que deve ser executado sempre que uma alteração no valor de ponteiros for realizada – faz

[5] Em geral deve-se utilizar registradores ou alocação em pilha para objetos desta natureza, pois a geração de uma célula que estará ativa durante um curto espaço de tempo tem custo computacional muito maior que a alocação de valores nestes locais. Esta técnica, chamada de "unboxing", é amplamente utilizada nos compiladores atuais, com ganhos significativos de desempenho.

com que o programa do usuário (ou compilador da linguagem) seja escrito de forma a utilizar a barreira de escrita adequada ao invés de operar diretamente com os ponteiros, tornando o uso do coletor de lixo não transparente para o programa do usuário. Essa desvantagem da contagem de referências se torna menos relevante o presente momento, já que todos os coletores estão sendo preparados para executar nos vários processadores disponíveis em um computador típico, e isso quase sempre impõe o uso de barreiras de escrita, igualando a situação da contagem de referências com as outras técnicas.

Há também um custo espacial: cada célula precisa guardar um contador além dos dados do programa. Dependendo do tamanho das células, essa sobrecarga pode ser mais ou menos significativa. Entretanto, existem técnicas para minimizar o tamanho do contador, até mesmo usando apenas um bit.

Estruturas de dados cíclicas O maior problema da contagem de referências é a detecção de células inativas em grafos cíclicos. Ciclos de células que se referenciam umas às outras mas que não possuem nenhuma referência que parta das raízes são inativas, e portanto poderiam ser reaproveitada. Mas no algoritmo clássico de contagem de referências, estas células sempre terão registrado um número de referências pelo menos igual a 1. Um exemplo é mostrado na Figura 2.1.

Existem algumas possibilidades para resolver este problema, como será visto no próximo capítulo. Em geral, combina-se a contagem de referências com alguma técnica de rastreamento, como um coletor local de marcação e varredura, para detectar os ciclos.

Um algoritmo de contagem de referências que pode ser utilizado para resolver o problema das referências cíclicas é apresentado no capítulo a seguir.

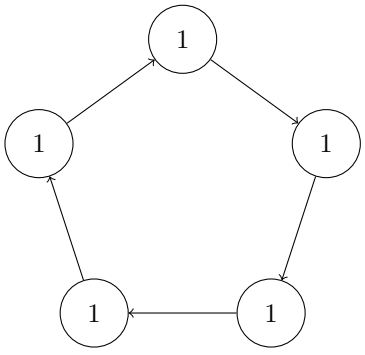

Figura 2.1: *Exemplo de ciclo de lixo. Nenhuma célula está conectada à raiz, mas o contador permanece em 1 devido às referências internas do ciclo.*

capítulo 3

Contagem de Referências Cíclicas

Como visto no capítulo anterior, a técnica original de contagem de referências não trata corretamente as estruturas cíclicas, pois qualquer conjunto de objetos que se referencie em um ciclo de ponteiros não é recuperado, mesmo que seja lixo. Este problema foi percebido por J. Harold McBeth em 1963 [77]. A solução mais comum, durante os anos seguinte, foi utilizar uma técnica híbrida, combinando a contagem de referências com alguma técnica baseada em rastreamento – normalmente marcação-e-varredura – para detectar e recuperar ciclos. Mas isso requer combinar dois sistemas globais para gerenciamento de memória, o que aumenta a complexidade geral e introduz redundância, já que os dois sistemas são relativamente independentes.

Neste capítulo é apresentada uma solução eficiente para o problema [64], baseada em um algoritmo que foi sucessivamente melhorado ([67, 69, 71, 72]) a partir da idéia inicial [72], tratada na seção 3.1. Em seguida, será tratada a versão mais recente e mais eficiente do algoritmo até agora, na seção 3.2. Por fim, o capítulo conclui com um exemplo para ilustrar melhor o funcionamento do algoritmo.

3.1 Detecção de ciclos usando uma varredura local

A primeira solução geral para o problema dos ciclos no algoritmo da contagem de referências foi publicada em 1990 no artigo de Martinez, Wachenchauzer e Lins [72]. A idéia do algoritmo neste artigo é realizar uma varredura local nos nós do grafo da memória que po-

dem ser ciclos, detectando corretamente os ciclos que não estão transitivamente conectados ao conjunto raiz. Um objeto O é considerado candidato para a varredura local quando seu contador de referências é decrementado para um valor maior que 0. Neste caso, o nó era compartilhado e existem duas possibilidades para a referência remanescente: ou ela vem de algum objeto ligado transitivamente às raízes, e O ainda está ativo; ou vem de um objeto ligado ciclicamente a O, e neste caso todos os objetos ligados ao ciclo estão inativos se nenhum deles estiver conectado a alguma raiz. O algoritmo então procede fazendo uma varredura no sub-grafo iniciado em O, removendo temporariamente as referências que podem ter origem em um ciclo; ao fim deste processo, se não restar nenhuma referência nos nós desse sub-grafo, pode-se concluir que os nós fazem parte de um ciclo, e podem ser recuperados.

Posteriormente, Lins publicou uma série de artigos [64, 67, 69] detalhando melhorias no algoritmo original. A principal melhoria é adiar a varredura local realizada em nós compartilhados, pois verifica-se empiricamente que a maioria dos nós compartilhados não faz parte de um ciclo (o artigo de Levanoni e Petrank [61] contém evidências recentes). Neste caso, o algoritmo original estaria desperdiçando o trabalho de varredura local, na maior parte do tempo. Já quando a varredura é adiada, muitas vezes é possível determinar sem esforço adicional se o nó não faz parte de um ciclo.

3.2 Contagem eficiente de referências cíclicas

Segue a descrição da versão mais recente do algoritmo para contagem de referências cíclicas [71]. O leitor interessado na evolução do algoritmo pode consultar os artigos [64, 67, 69] ou a dissertação de mestrado de Salzano Filho [35].

Além do contador de referências, é preciso armazenar em cada objeto um campo indicando sua cor. Três cores são possíveis: verde indica que a célula está em uso; vermelho é usado para células que estão sendo varridas localmente pelos algoritmos de verificação de ciclos; e células com a cor preta estão marcadas para uma varredura local posterior. Uma estrutura de dados, chamada de *status analyser*, é utilizada para guardar ponteiros para todas as células que poderão ser varridas.

Novas células são obtidas através da função NEW, mostrada no Algoritmo 3.1. A nova célula é obtida da lista livre, se esta não estiver vazia; caso contrário, o algoritmo dispara uma varredura do *status analyser* para tentar recuperar células fora de uso. Em todo caso, se uma célula é obtida, seu contador de referências inicia com o valor 1 e sua cor é verde. A função

ALLOCATE, que retira uma célula da lista livre, é idêntica à mostrada no Algoritmo 2.8. Quando é necessário criar uma nova referência para a célula T, chama-se a rotina CREATE, que apenas incrementa o contador de referências de T e muda sua cor para verde; isso é feito pois se T só pode receber uma nova referência se estiver ativa. Note que isto pode ocorrer enquanto T está no *status analyser*, o que significa que quando esta estrutura for varrida, a condição de T já será conhecida e não será necessário fazer uma varredura local no seu sub-grafo.

Algoritmo 3.1 Operação para obter uma nova célula na contagem de referências cíclicas

```
1:  procedure NEW
2:      if freeList is not empty then
3:          newCell := ALLOCATE()
4:      else
5:          if statusAnalyser is not empty then
6:              SCAN-STATUS-ANALYSER()
7:              newCell := NEW()
8:          else
9:              ABORT("Não há células disponíveis")
10:         end if
11:     end if
12:     newCell.rc := 1
13:     newCell.color := green
14:     return newCell
15: end procedure
```

Algoritmo 3.2 Criação de uma referência na contagem de referências cíclicas

```
1: procedure CREATE(T)
2:     T.rc := T.rc + 1
3:     T.color := green
4: end procedure
```

Algoritmo 3.3 Alteração de ponteiros na contagem de referências cíclicas

```
1: procedure UPDATE(R, S)
2:     DELETE(*R)
3:     S.rc := S.rc + 1
4:     *R = S
5: end procedure
```

A outra parte da interface do sistema de contagem de referências com o programa são as rotinas para alteração e remoção de referências. A rotina UPDATE para a contagem de

referências cíclicas é idêntica à versão para a contagem de referências tradicional, mostrada no Algoritmo 2.9, e repetida aqui no Algoritmo 3.3 por conveniência. Entretanto, DELETE é diferente: caso o contador da célula tenha valor 1 na chamada a DELETE, significa que ela será liberada, e seu sub-grafo precisa ser ajustado, da mesma forma que no algoritmo tradicional; entretanto, se o contador tinha valor maior que 1, a célula era compartilhada e deve ser examinada posteriormente para verificar a existência de ciclos. Por isso, a célula neste caso é pintada com a cor preta e adicionada ao *status analyser*.

Algoritmo 3.4 Remoção de um ponteiro na contagem de referências cíclicas

```
1:  procedure DELETE(S)
2:      if S.rc = 1 then
3:          for each T in S.children do
4:              DELETE(T)
5:          end for
6:          S.color := green
7:          LINK-TO-FREE-LIST(S)
8:      else
9:          S.rc := S.rc − 1
10:         if S.color ≠ black then
11:             S.color := black
12:             ADD-TO-STATUS-ANALYSER(S)
13:         end if
14:     end if
15: end procedure
```

A parte da técnica que trata da detecção de ciclos está expressa em um conjunto de rotinas inter-relacionadas. Como visto no Algoritmo 3.1, a rotina NEW pode disparar uma varredura do *status analyser* caso não haja mais células disponíveis na lista livre. Esta varredura é feita pela rotina SCAN-STATUS-ANALYSER, mostrada como Algoritmo 3.5. Esta rotina examina todas as células no *status analyser* em alguma ordem[1], disparando uma varredura local em todas as que ainda tiverem a cor preta. Esta varredura é feita pela rotina MARK-RED; ao fim da varredura, se a célula tiver cor vermelha mas contador de referências maior que zero, isso significa que há alguma referência externa (transitivamente ligada às raízes) para ela, e ela está ativa. Neste caso, é chamada a rotina SCAN-GREEN, para reestabelecer a cor verde na célula e em seu sub-grafo. A chamada recursiva em SCAN-STATUS-ANALYSER indica que

[1] Em geral, as células do *status analyser* são processadas em uma ordem FIFO, como em uma fila. As células podem ser processadas em qualquer ordem, sem que isso afete a corretude do algoritmo, mas a eficiência do processo de verificação do *status analyser* pode depender da ordem escolhida.

o processo continua com outras células; ao final, as células que ainda estiverem com a cor vermelha podem ser consideradas inativas e devolvidas à lista livre, o que é feito pela rotina COLLECT.

Algoritmo 3.5 Rotina para varrer o *status analyser*

```
1:  procedure SCAN-STATUS-ANALYSER
2:      S := SELECT-FROM(statusAnalyser)        ▷ Seleciona e retira um item do status analyser
3:      if S = nil then
4:          return
5:      end if
6:      if S.color = black then
7:          MARK-RED(S)
8:      end if
9:      if S.color = red ∧ S.rc > 0 then
10:         SCAN-GREEN(S)
11:     end if
12:     SCAN-STATUS-ANALYSER()
13:     if S.color = red then
14:         COLLECT(S)
15:     end if
16: end procedure
```

MARK-RED é a rotina responsável pela varredura local de células compartilhadas, a fim de detectar ciclos. Quando chamada para examinar uma célula S, primeiro a cor de S é alterada para vermelho. Em seguida, MARK-RED reduz o contador de referências de todas as células apontadas por S; isso é feito para remover referências internas a um ciclo ou sub-grafo local. Em seguida, as células apontadas por S são varridas através da chamada recursiva a MARK-RED, caso elas já não tenham cor vermelha, o que significa que já foram examinadas. Após a chamada recursiva, toda célula apontada por S que continua com um contador de referências maior que zero é adicionada ao *status analyser*, para que seja examinada posteriormente. Isto é feito para identificar outros pontos críticos no grafo da memória que possam indicar a presença ou ausência de ciclos, possivelmente acelerando o processo de varredura.

Na varredura local com MARK-RED as células de um sub-grafo em análise têm seus contadores reduzidos para eliminar o papel das referências internas ao sub-grafo. Depois disso, se nenhuma célula no sub-grafo possuir um contador com valor maior que zero, pode-se concluir que o mesmo não está transitivamente conectado a alguma raiz e pode ser coletado; isso acontece na chamada a COLLECT dentro de SCAN-STATUS-ANALYSER (ver Algoritmo 3.5).

Algoritmo 3.6 Rotina para marcar células de vermelho

```
1:  procedure MARK-RED(S)
2:      if S.color ≠ red then
3:          S.color := red
4:          for each T in S.children do
5:              T.rc := T.rc − 1
6:          end for
7:          for each T in S.children do
8:              if T.color ≠ red then
9:                  MARK-RED(T)
10:             end if
11:             if T.rc > 0 ∧ T is not in statusAnalyser then
12:                 ADD-TO-STATUS-ANALYSER(T)
13:             end if
14:         end for
15:     end if
16: end procedure
```

Entretanto, se alguma célula C possui contador maior que zero, isso indica que há pelo menos uma referência externa ao sub-grafo da célula C, e todas as células no fechamento transitivo da relação de referência sobre C que tenham contador maior que 1 podem ser consideradas ativas. Para reestabelecer a cor verde nessas células, e reajustar a validade dos contadores a partir da célula C, é chamada a rotina SCAN-GREEN, detalhada no Algoritmo 3.7. Quando chamada em uma célula S, SCAN-GREEN simplesmente muda a cor de S para verde e aumenta a contagem de referências de todas as células apontadas por S, chamando a si mesma recursivamente nessas. Isso é suficiente para desfazer o trabalho feito por MARK-RED caso as células sejam ativas.

Algoritmo 3.7 Rotina para varrer células pintadas de verde

```
1:  procedure SCAN-GREEN(S)
2:      S.color := green
3:      for each T in S.children do
4:          T.rc := T.rc + 1
5:          if T.color ≠ green then
6:              SCAN-GREEN(T)
7:          end if
8:      end for
9:  end procedure
```

Finalmente, COLLECT é responsável pela coleta das células que foram determinadas como

inativas após a varredura com MARK-RED. Neste caso, o processo de coleta para uma célula S consiste em remover a referência de S para todas as células apontadas por ela, e devolver S à lista livre. Se alguma célula apontada por S também tiver cor vermelha, ela pode ser coletada similarmente; isso é indicado pela chamada recursiva a COLLECT. Desta forma, um sub-grafo considerado inativo, ou seja, um ciclo sem referências externas, pode ser recuperado.

Algoritmo 3.8 Rotina para coleta das células inativas

```
1:  procedure COLLECT(S)
2:      for each T in S.children do
3:          DELETE(T)
4:          if T.color = red then
5:              COLLECT(T)
6:          end if
7:      end for
8:      S.color = green
9:      ADD-TO-FREE-LIST(S)
10: end procedure
```

3.3 Exemplos

Nesta seção são apresentados dois exemplos simples do algoritmo de contagem de referências cíclicas em funcionamento, para ilustrar a técnica. O primeiro mostra um ciclo que contém uma referência externa, e portanto ainda está ativo, enquanto o segundo mostra uma situação de um ciclo que é lixo e pode ser recuperado.

3.3.1 Exemplo de um ciclo ativo

A Figura 3.1 mostra a situação inicial para o primeiro exemplo. Dois objetos, X e Y, apontam para um ciclo de três outros objetos. Considera-se que X e Y estão transitivamente conectados ao conjunto raiz, embora isso não seja mostrado na figura. Os objetos do ciclo são mostrados com o valor de seus contadores de referência; o valor dos contadores de X e Y não é importante para o exemplo. Todos os objetos estão com cor verde, pois todo conjunto está ativo.

Em seguida, a referência de X para o ciclo é retirada, o que muda o contador do objeto mais à esquerda no ciclo de 2 para 1; neste momento o objeto é alterado para a cor preta e adicionado ao *status analyser*, para análise posterior. Esta situação é mostrada na Figura 3.2.

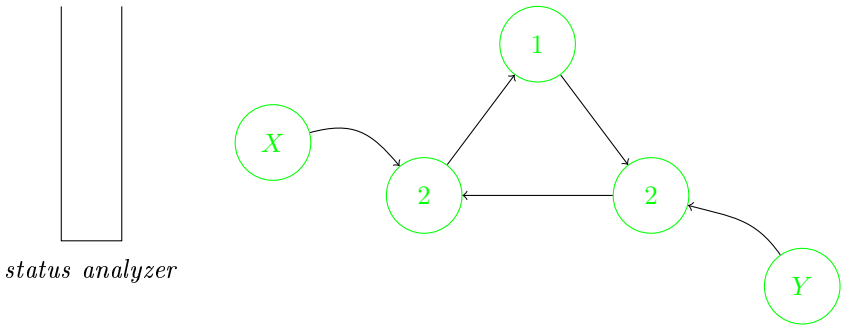

Figura 3.1: *Exemplo 1: situação inicial*

Supondo que a situação local deste sub-grafo da memória não mude até uma varredura do *status analyser*, o nó de cor preta na figura será eventualmente escolhido em SCAN-STATUS-ANALYSER, que verificará sua cor como preta e chamará MARK-RED. Esta rotina irá examinar o sub-grafo começando no nó à esquerda, pintando-o de vermelho, decrementando o contador de referências nos nós descendentes, e chamando MARK-RED recursivamente. A Figura 3.3 mostra a situação após a rotina MARK-RED ter concluído nesta parte do subgrafo. É importante notar também que, nessa figura, o nó do ciclo que fica com o contador de referências igual a 1 é adicionado ao *status analyser* por MARK-RED.

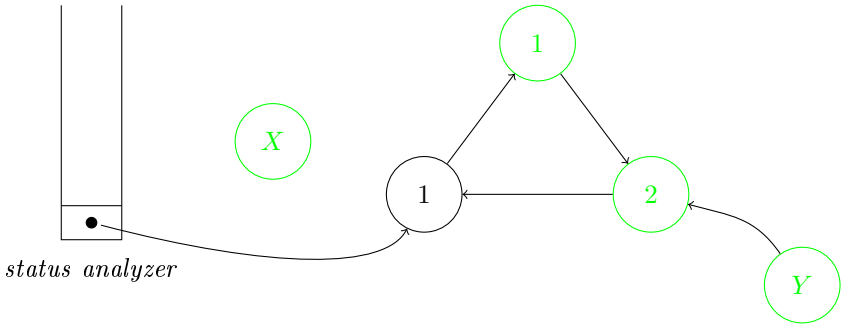

Figura 3.2: *Exemplo 1 após remoção da referência*

Tendo MARK-RED terminado o trabalho de análise local no sub-grafo, o controle retorna para SCAN-STATUS-ANALYSER, que continua a analisar células até eventualmente chegar na célula do ciclo que tem o contador de referências igual a 1. Neste momento, como a cor da célula é vermelha e seu contador é maior que zero, é chamada a rotina SCAN-GREEN para desfazer o que foi feito durante a varredura em MARK-RED, começando pelo nó com contador igual a 1. Então, SCAN-GREEN muda novamente a cor das células vermelhas para verde e reestabelece a contagem correta de referências. A situação final, após o fim de SCAN-GREEN,

é mostrada na Figura 3.4

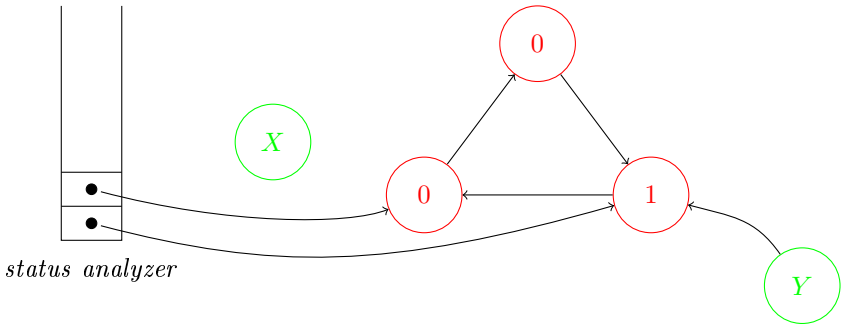

Figura 3.3: *Exemplo 1 após a varredura local e marcação em vermelho*

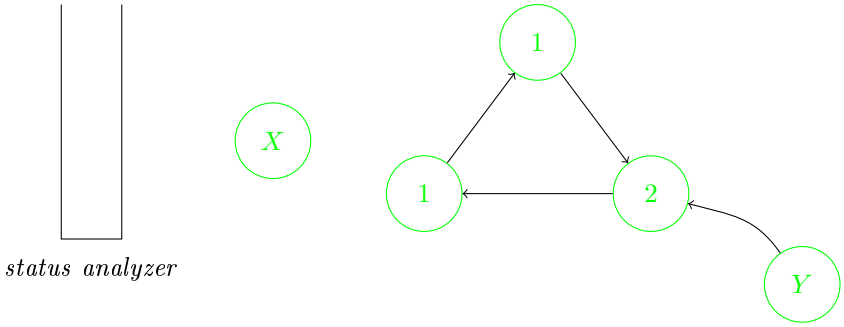

Figura 3.4: *Exemplo 1 após a recuperação das células para verde*

3.3.2 Exemplo de um ciclo inativo

O próximo exemplo mostra uma situação similar à anterior, mas desta vez o ciclo não possui referências externas, e portanto pode ser liberado. A Figura 3.5 mostra a situação inicial.

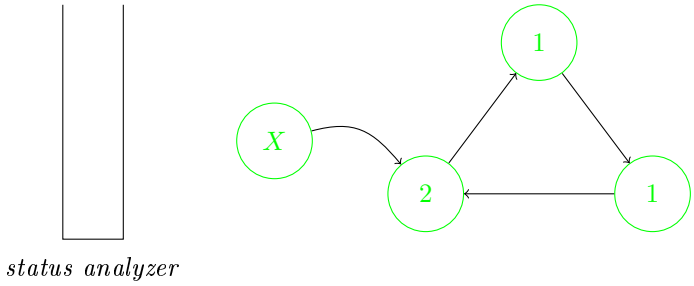

Figura 3.5: *Exemplo 2: situação inicial*

A referência de X para o ciclo é retirada, o que causa mudança no contador de referências

na célula afetada, do valor 2 para 1. A mesma célula é adicionada ao *status analyser*. Esta é a situação mostrada na Figura 3.6.

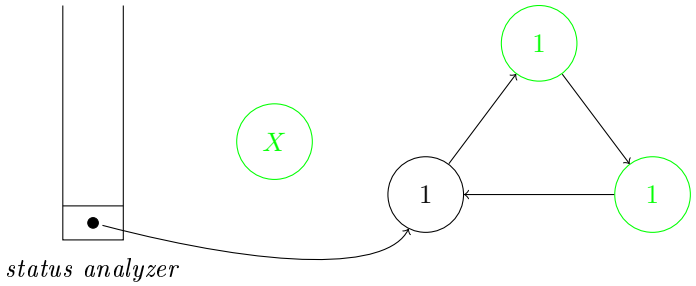

Figura 3.6: *Exemplo 2 após retirada da referência*

Novamente, supondo que a situação nesta região do grafo da memória não se altere até uma chamada a SCAN-STATUS-ANALYSER, esta rotina eventualmente tomará a célula de cor preta na Figura 3.6 para análise, chamando MARK-RED para varrer seu sub-grafo. As células conectadas ao ciclo são então pintadas de vermelho e seus contadores de referência decrementados em uma unidade, para desconsiderar as referências internas. A situação após o final da varredura local é mostrada na Figura 3.7. Note-se que nesse caso não resta nenhuma célula no ciclo com contador de referências maior que zero, o que indica, corretamente, que o ciclo está inacessível a partir das raízes, e portanto é lixo.

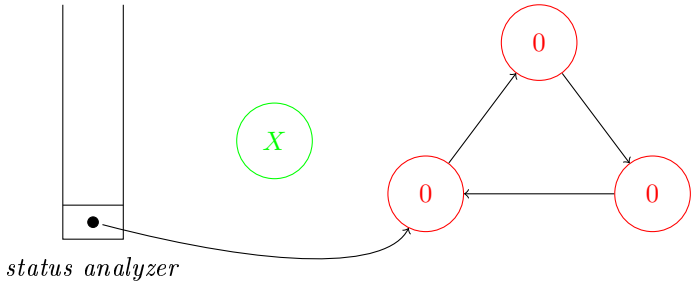

Figura 3.7: *Exemplo 2 após varredura local*

Após MARK-RED, o controle volta para a rotina SCAN-STATUS-ANALYSER, que continuará analisando células. Como o ciclo foi desconectado do conjunto raiz e nenhuma de suas células está no *status analyser*, sua situação não muda com relação à Figura 3.7. Depois de terminar com todas as células que precisavam de análise, a rotina SCAN-STATUS-ANALYSER volta a considerar as células já analisadas para coletar as que permanecerem com a cor vermelha. Neste caso, a célula mais à esquerda no ciclo das figuras estava no *status analyser* e, como

continua vermelha, será coletada pela chamada a COLLECT. Esta última rotina se encarregará de coletar todas as células no sub-grafo representado pelo ciclo, devolvendo-as à lista livre.

Neste capítulo foi considerado o algoritmo para gerenciamento de memória baseado na contagem de referências cíclicas, mas apenas para o caso seqüencial, para computadores uniprocessados. Os Capítulos 4 e 5 descrevem técnicas para realizar o gerenciamento de memória em um ambiente multiprocessado. O algoritmo seqüencial visto neste capítulo serve como base para as versões da contagem de referências cíclicas em ambientes multiprocessados.

capítulo 4

Sistemas com Um Mutador e Um Coletor

Como foi visto no Capítulo 2, a contagem de referências é um método para gerenciamento automático da memória que tem algumas vantagens em relação às outras técnicas tradicionais, como coleta por marcação e varredura e coleta por cópia. Entre estas vantagens estão o fato de ser um algoritmo naturalmente incremental, que não possui a necessidade de parar a computação realizada pelos processos do usuário, e ter uma complexidade temporal que não depende do tamanho total do *heap*.

Entretanto, uma das desvantagens do algoritmo básico de contagem de referências é a sua incapacidade em recuperar ciclos que se tornam inacessíveis no grafo da memória. No Capítulo 3 foi visto como solucionar este problema sem utilizar um segundo coletor de lixo para recuperar os ciclos. O algoritmo completo apresentado no Capítulo 3 mantém as características citadas acima: é incremental e sua complexidade temporal não é afetada pelo tamanho do heap. Entretanto, este algoritmo é adequado apenas para ambientes de execução sequencial, nos quais o programa é composto de apenas um *thread*.

Neste capítulo são apresentados algoritmos e técnicas que podem ser usados para estender a coleta de lixo por contagem de referências para ambientes multiprocessados, com execução concorrente e simultânea de vários processos ou *threads*[1]. Inicialmente, são descritos os problemas da versão sequencial do algoritmo de contagem de referências cíclicas (visto no Capítulo 3)

[1] Neste trabalho, usa-se geralmente o termo *thread* para referenciar uma unidade de execução que pode ser escalonada pelo sistema operacional para executar em um dos processadores disponíveis. Esse mesmo conceito é chamado de *processo* em alguns trabalhos mais antigos na literatura relevante, mas em geral os processos em questão compartilham um espaço de memória, o que os torna mais parecidos com os *threads* dos sistemas atuais.

quando usado em ambientes multiprocessados com execução concorrente real. Em seguida, é apresentada a nomenclatura normalmente usada na literatura sobre coleta de lixo em sistemas multiprocessados. Os algoritmos de coleta de lixo por contagem de referências em ambientes multiprocessados são então apresentados, para o caso mais simples no qual é designado apenas um *thread* para realizar a coleta, enquanto um *thread* executa o programa do usuário.

Os algoritmos apresentados neste capítulo são baseados em algoritmos descritos em trabalhos anteriores [68, 70], mas esses algoritmos anteriores nunca tinham sido implementados. Uma implementação dos algoritmos descritos é apresentada na Seção 4.4, juntamente com os resultados de testes que foram realizados para analisar seu desempenho.

No Capítulo 5 são apresentados algoritmos para um caso mais geral em que mais de um *thread* pode ser usada para executar o programa do usuário.

4.1 Coleta de Lixo em Sistemas Multiprocessados

É possível usar o algoritmo sequencial visto no Capítulo 3 na execução de programas compostos por mais de um *thread*, se duas condições forem satisfeitas: primeiro, os *threads* não podem executar paralelamente, apenas terem sua execução intercalada; e segundo, o sistema de tempo de execução deve fazer o escalonamento desses *threads* e coordenar tal escalonamento com o coletor de lixo. Ou seja, deve-se assumir que o ambiente computacional onde o programa executa possui apenas um processador, e executar os *threads* intercalando-os em uma única unidade de execução. Um sistema de execução desta forma é dito *single-threaded*, pois apenas um *thread* pode executar por vez. Mesmo com essa limitação, o sistema de tempo de execução deve coordenar que operações podem ser intercaladas entre os *threads*, caso contrário as estruturas de controle do coletor de lixo ou mesmo objetos no grafo da memória podem ser corrompidos.

Entretanto, esse tipo de organização é pouco desejável para sistemas de tempo de execução em computadores atuais, já que a maioria destes computadores possui processadores compostos por vários núcleos de processamento independentes, funcionando efetivamente como sistemas multiprocessados. Neste caso, usar um sistema de tempo de execução *single-threaded* é subutilizar a capacidade computacional disponível, e desperdiçar a possibilidade de aumentar a eficiência do programa executado.

Para tornar a execução de programas mais eficiente em sistemas multiprocessados, é preciso permitir a concorrência real entre os *threads* que compõem os programas, fazendo com

que esses *threads* possam ser executadas em diferentes processadores (ou núcleos de processamento). Neste caso, o sistema de tempo de execução deve incluir um coletor de lixo que possa funcionar no ambiente multiprocessado.

Os problemas que se apresentam para um coletor que deve operar em ambiente multiprocessado, com concorrência real entre *threads*, se devem às possibilidades de *interferência* entre o coletor e o programa do usuário. Em programas com múltiplas *threads*, uma decomposição comum é separar o coletor de lixo em um ou mais *threads*, enquanto o programa é executado em outro(s) *thread*(s). A interferência entre esses *threads* de natureza diferente – ambos os tipos precisam de acesso a informações nos objetos em memória – é que causa as dificuldades inerentes aos coletores de lixo em sistemas multiprocessados.

Do ponto de vista do coletor de lixo, tudo que os *threads* do programa do usuário fazem é alterar o grafo de objetos na memória. Por esse motivo, é tradicional chamar esses *threads* de *mutadores* ou *threads mutadores*, enquanto que os *threads* do coletor de lixo são chamadas de *coletores* ou *threads coletores*[2]. Os problemas causados pela interferência entre os dois tipos de *threads* são detalhados na subseção seguinte.

4.1.1 Interferências entre o Coletor e Mutador

Tanto os coletores quanto os mutadores acessam o grafo de objetos na memória e, em alguns casos, os dois tipos de *threads* precisam alterar o grafo. Uma solução simples para evitar que o coletor de lixo interfira com os mutadores é parar a execução de todos os *threads* mutadores no momento que o coletor de lixo precisar iniciar sua operação. Essa solução é natural para coletores baseados em rastreamento do *heap*, como coletores de cópia, pois esse tipo de algoritmo, mesmo em suas versões sequenciais, para a computação do usuário quando precisa realizar a coleta. Embora simples e correta, essa solução é claramente indesejável, pois enquanto ocorre a coleta apenas um processador estaria sendo utilizado, problema que se torna mais sério à medida que o número de núcleos de processamento disponíveis aumenta.

Para coletores baseados em contagem de referências, parar todos os mutadores ao realizar a coleta é menos natural, já que, na versão original do algoritmo, as operações de gerenciamento da memória são intercaladas com as operações dos mutadores. Entretanto, é possível usar

[2]Neste capítulo e nos seguintes, a palavra *coletor* é usada em dois sentidos diferentes: primeiro, como a parte do sistema de tempo de execução responsável pelo gerenciamento da memória dinâmica e pela recuperação da memória utilizada por objetos não mais acessíveis; e segundo, como um *thread* que participa do processo de coleta de lixo, sendo possivelmente uma entre muitos *threads* empregados pelo coletor. O sentido utilizado em cada ocorrência específica da palavra deve ficar claro pelo contexto.

a estratégia de parar todos os mutadores – conhecida como "parar o mundo" ou *stop-the-world* – em coletores por contagem de referências, se os invariantes do algoritmo original forem modificados; de fato, essas modificações geralmente devem ser feitas em ambientes multiprocessados para evitar problemas de interferência entre coletores e mutadores.

Do ponto de vista do coletor por contagem de referências, as informações importantes em cada objeto são o contador de referências e os ponteiros ou referências que o objeto possui apontando para outros objetos. No algoritmo do Capítulo 3, qualquer alteração de uma referência que apontava para um objeto O_1 e passa a apontar para um objeto O_2 causa uma redução no contador de referências de O_1 e um incremento no contador de O_2. Se vários mutadores puderem alterar referências para objetos em comum ao mesmo tempo, condições de corrida no acesso concorrente podem causar inconsistência no grafo de objetos e desrespeitar as invariantes do algoritmo de contagem de referências.

Um coletor por contagem de referências trabalha fortemente acoplado com o mutador (ou mutadores), o que acarreta que a ocorrência de interferências entre coletor e mutador se torna freqüente quando executados concorrentemente. De maneira informal e para os propósitos deste trabalho, um *thread interfere* com outro quando executa alguma instrução que invalida uma asserção de alguma instrução no outro *thread*. Uma definição mais formal pode ser encontrada nos livros de Andrews [3, 4].

Especificamente, existem duas dificuldades principais ou intrínsecas da contagem de referências em sistemas multiprocessados, e mais uma dificuldade que decorre de uma característica comum da sua implementação:

1. a manutenção dos contadores de referências;
2. as alterações nos ponteiros;
3. a manipulação da lista livre.

A primeira dificuldade ocorre porque toda operação com ponteiros deve alterar o contador de referências de uma ou duas células, e portanto podem acontecer condições de corrida quando duas ou mais *threads* tentam alterar o contador da mesma célula. Por exemplo, seja uma célula C com contador de referências igual a um, e duas *threads* que executam concorrentemente as seguintes operações:

- ▷ *thread* 1 adiciona uma referência a C
- ▷ *thread* 2 remove uma referência a C

Claramente, o contador de referências de C deve ter, ao final das duas operações, o valor 1. Mas pode ocorrer que, no escalonamento do sistema, o *thread* 2 execute suas operações antes que o *thread* 1; neste caso, o contador de C chegará ao valor zero, e a célula será liberada imediatamente. Quando o *thread* 1 tentar adicionar uma referência a C, esta estará na lista livre, e estará sujeita a ser alocada para outro fim, o que é uma situação indesejável e pode tornar o sistema instável. Esse é um exemplo de interferência entre as *threads*, pois uma delas invalida uma asserção da outra. Muitas outras condições de corrida podem ocorrer, inclusive que duas *threads* se intercalem durante a alteração do valor de um contador, deixando-o com um valor diferente do que ambas esperam.

A dificuldade com relação à alteração dos ponteiros ocorre pois os mutadores não realizam diretamente essas alterações; é preciso utilizar as operações da interface do coletor para isso, já que toda alteração deve contabilizar as mudanças nos contadores de referências das células envolvidas. Entretanto, enquanto o programa dos mutadores pode conter o conhecimento de que células são compartilhadas por quais *threads*, o programa do coletor é genérico e não deve ter conhecimento dos arranjos de compartilhamento entre os mutadores. Isso se traduz em um problema de interferência entre mutadores no qual o coletor não tem o conhecimento de quais podem interferir com quais outros, então deve assumir o pior: que qualquer *thread* mutador pode interferir com qualquer outra. Assim, para garantir que os ponteiros serão alterados corretamente, uma solução direta para garantir a não-interferência é usar um *mutex* (semáforo para exclusão mútua) compartilhado entre o coletor e todos os mutadores, efetivamente serializando todas as alterações em ponteiros. Esta é a solução usada, por exemplo, no coletor por contagem de referências da linguagem Modula-2+, segundo relatado por DeTreville [27]. Infelizmente é uma solução que dificulta o escalonamento para um maior número de processadores: intuitivamente, quanto maior o número de processadores executando concorrentemente, maior será a contenção para obter o *mutex* para alterações de ponteiros, e portanto mais tempo será gasto com sincronização; com um número suficiente de processadores, pode-se imaginar que a sincronização chega a tomar um tempo comparável ou mesmo maior que o tempo gasto em computação útil. O relatório técnico de DeTreville [27] cita uma experiência deste tipo, na qual um coletor que utilizava somente contagem de referências foi descartado por ter desempenho insatisfatório.

Por último, a manutenção da lista livre inclui mais duas condições de corrida possíveis: quando é preciso alocar uma nova célula, e portanto retirá-la da lista livre; e quando é preciso

devolver uma célula para a lista, no momento da liberação. Alterações concorrentes sem exclusão mútua podem corromper a lista livre, inviabilizando o uso continuado do sistema.

Nas seções que seguem serão apresentadas e analisadas as arquiteturas propostas para implementar a contagem de referências em sistemas multiprocessados, e para cada uma delas será observado como esses problemas são resolvidos, e quão satisfatórias são as soluções empregadas.

4.2 Nomenclatura e Classificação

Existem várias possibilidades para aproveitar as capacidades de um sistema multiprocessado para realizar a coleta de lixo, e isso dá origem a uma nomenclatura que pode parecer confusa para quem não está habituado a ela. O uso dos termos constantes nessa nomenclatura também variou com o tempo. Nos últimos anos, entretanto, pelo aumento do interesse e da pesquisa na área, os termos se tornaram mais padronizados, como pode ser visto no glossário presente no artigo de Levanoni e Petrank [61].

Os quatro termos principais para coletores em sistemas multiprocessados são:

- ▷ *stop-the-world*
- ▷ paralelo
- ▷ concorrente
- ▷ *on-the-fly*

Essa classificação e os tipos de coletores especificados são detalhados abaixo e ilustrados na Figura 4.1. Os termos usados possuem definições específicas na literatura de coletores de lixo; tais definições são apresentadas a seguir.

Um coletor *stop-the-world* (ou *coletor de parada*) emprega a estratégia mais simples: parar a execução de todos os mutadores enquanto o coletor realiza sua tarefa. Tal simplicidade tem a vantagem de evitar completamente os problemas de interferência, fazendo com que um coletor sequencial correto possa ser empregado. Entretanto, a maioria dos processadores disponíveis estará sem utilização durante a coleta; apenas um processador será aproveitado nesses momentos, o que significaria uma eficiência muito baixa em um sistema com dezenas de processadores disponíveis, por exemplo. Note-se que em um sistema com apenas um processador disponível, o modelo *stop-the-world* coincide com o funcionamento normal dos

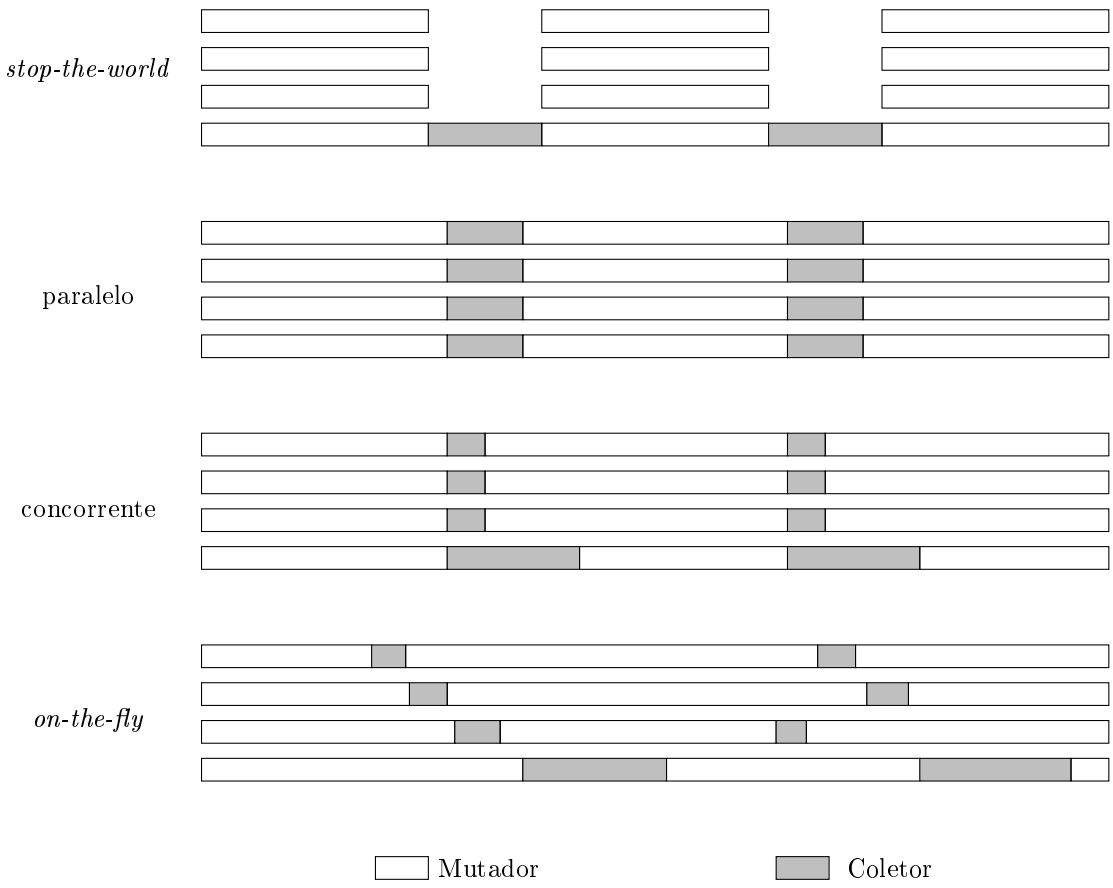

Figura 4.1: *Ilustração da diferença entre os tipos de coletores adequados para emprego em sistemas multiprocessados. Em cada grupo, as barras horizontais representam* threads *executando paralelamente em processadores diferentes; a dimensão horizontal representa a passagem do tempo, enquanto a dimensão vertical representa a situação em diferentes processadores.*

coletores de rastreamento (por cópia ou marcação e varredura). Coletores *stop-the-world* foram usados exclusivamente pelas máquinas virtuais Java até a versão 1.4 [43].

Em um coletor de parada, além da ineficiência no uso das capacidades de um sistema multiprocessado, todos os *threads* do usuário são paradas enquanto a coleta é realizada, o que pode causar pausas e baixa responsividade na interface com o usuário, e aumentar a variabilidade no tempo de resposta de servidores. Longas pausas também impossibilitam o uso do sistema em situações de tempo-real, onde o tempo de resposta deve ser controlado. Por esse motivo, é desejável reduzir o tempo de pausa imposto pelo coletor. A redução das pausas é um dos motivos que levaram a Sun (antes da aquisição pela Oracle) a integrar um novo coletor experimental (chamado de *garbage-first* ou G1) nas versões futuras da sua máquina virtual Java [23, 25].

O coletor paralelo também para todos os *threads* mutadoras para realizar a coleta, mas tem o processo de coleta em si paralelizado para utilizar os processadores que estiverem disponíveis. Nesse caso, o coletor utiliza toda a capacidade computacional disponível no sistema, normalmente obtendo um desempenho superior aos coletores de parada. Todas os *threads* de usuário ainda são paradas ao mesmo tempo, o que pode comprometer a responsividade do programa para o usuário. Espera-se que a duração das pausas deve ser menor, com relação aos coletores de parada, já que todos os processadores podem ser usados para acelerar o processo de coleta. A máquina virtual *HotSpot* da Sun Microsystems incluiu coletores paralelos a partir da versão 1.4.1 [43].

Um coletor concorrente executa simultaneamente com *threads* mutadores na maior parte do tempo, mas ainda necessita parar todos os mutadores ao mesmo tempo por algum período, geralmente curto. Na Figura 4.1, o coletor concorrente para todas menos um dos *threads* mutadores por um curto espaço de tempo, e depois continua com a maior parte do processo de coleta em um *thread* específico. Um coletor concorrente tende a apresentar tempos menores de pausa do programa do usuário, em relação aos coletores paralelos e de parada. O limitante para a redução dessa pausa é a necessidade de parar todos os mutadores ao mesmo tempo.

Quando um coletor precisa parar um mutador para coordenar a atividade de coleta, diz-se que ocorreu um ***handshake*** entre mutador e coletor [30, 31]. Um *handshake* no qual todos os mutadores devem ser parados simultaneamente é classificado como um *hard handshake*. Um coletor concorrente, portanto, é um coletor que necessita de um *hard handshake* no seu

processo de coleta.

Por fim, um coletor ***on-the-fly*** não precisa parar todos os mutadores ao mesmo tempo, embora ainda possa precisar parar os *threads* mutadores individualmente. Esse tipo de parada em que apenas um mutador precisa ser suspendido por vez é chamado de *soft handshake* [30, 31]. Coletores *on-the-fly* são desejáveis por poderem reduzir os tempos de pausa ainda mais com relação aos coletores concorrentes. Várias técnicas de coleta de lixo pesquisadas recentemente são *on-the-fly* [7, 9, 61].

Os algoritmos de coleta de lixo baseados em contagem de referências apresentados neste capítulo são todos *on-the-fly*, como pode ser visto nas seções seguintes. A característica naturalmente incremental da técnica de contagem de referências é um bom ponto de partida para coletores *on-the-fly*, que nunca param todos os mutadores simultaneamente.

4.3 Contagem de Referências Cíclicas com um Coletor e um Mutador

Inicialmente, as primeiras tentativas de criar algoritmos para coleta de lixo em múltiplos processadores trabalharam apenas com um mutador e um coletor. Um exemplo bem conhecido é o artigo de Steele [90]. As primeiras tentativas de adaptar o algoritmo de contagem de referências cíclicas para o ambiente multiprocessado também usam uma arquitetura baseada em dois processadores: um mutador e um coletor; vide os artigos de Lins [66, 68].

Assumir a existência de apenas um coletor e um mutador é um primeiro passo na adaptação dos algoritmos de coleta de lixo, e simplifica o algoritmo resultante, facilitando sua compreensão. Coincidentemente, nem o algoritmo de Steele [90], nem os algoritmos de Lins [66, 68] foram implementados. Na época em que ambos foram desenvolvidos, não existia consenso sobre que tipo de arquitetura de *hardware* e *software* se tornaria dominante em sistemas multiprocessados, e era mais difícil realizar uma implementação. O modelo de programação baseado em *threads* e memória compartilhada, presente na maioria das plataformas atuais, ou não existia ou não era muito difundido[3].

Desta forma, a primeira implementação de uma versão do algoritmo de contagem de

[3] Por exemplo, o padrão POSIX para *threads*, conhecido como *pthreads*, foi publicado apenas em 1995 (Padrão IEEE 1003.1c, 1995; incorporado no Padrão IEEE 1003.1, 1996, disponível em `http://standards.ieee.org/catalog/olis/arch_posix.html`). A plataforma Java, cujo sucesso na indústria de *software* ajudou a popularizar o modelo de *threads*, só teve sua primeira versão divulgada em 1996, como pode ser visto em `http://www.java.com/en/javahistory/timeline.jsp`

referências cíclicas de Lins adequado para arquiteturas multiprocessadas só apareceu mais tarde, no trabalho de Formiga e Lins [39], que propunha um algoritmo modificado com o objetivo de reduzir a quantidade de operações de sincronização realizadas; o algoritmo original de Lins [66, 68] se baseava no uso de semáforos e na priorização de operações de acesso à memória (priorizando o processo mutador) como mecanismos de sincronização. Entretanto, o uso de semáforos ou outros tipos similares de *locks* (*mutex*, monitores) tende a ser ineficiente quando há contenção, o que faz que a maioria dos algoritmos recentes de coleta de lixo evitem esse tipo de operação de sincronização. Esse algoritmo é descrito a seguir.

4.3.1 O Algoritmo de Formiga e Lins para um Coletor e um Mutador

O algoritmo de Formiga e Lins [39] foi implementado em uma plataforma de testes sintética que consistia em um compilador para uma linguagem funcional *lazy* simples, usando técnicas baseadas na G-Machine [34, 89] e usando o algoritmo de coleta no seu sistema de tempo de execução.

O algoritmo é, assim como os seus antecessores, planejado para trabalhar com um mutador e um coletor; essa suposição sobre o ambiente computacional não é adequada para a maioria das linguagens atuais com suporte a *threads*, já que em geral o número de *threads* que pode ser criadas não é limitado a apenas uma, mas o algoritmo serve como um ponto de partida para a versão para n mutadores e um coletor, como será visto na Seção 5.3.

A Figura 4.2 mostra uma visão esquemática do algoritmo para um mutador (representado como P_1) e um coletor (representado como P_2 na figura). Ambos compartilham o acesso a três estruturas de dados: a lista livre – tratada como uma fila – e duas filas que registram alterações nas referências, chamadas de *fila de incrementos* e *fila de decrementos*; essas estruturas mediam a coordenação entre coletor e mutador. Como tais estruturas são filas, mutador e coletor têm acesso a extremidades diferentes de cada fila, dependendo do papel como produtor ou consumidor em relação à fila. Assim, o mutador consume células livres, e portanto tem acesso ao início da lista de células livres (*topFreeList*), para retirar da lista as células livres que precisa. De forma complementar, o coletor produz células livres, que são reaproveitadas no processo de coleta, e as insere na lista de células livres; portanto, o coletor tem acesso ao final da lista (*botFreeList*). A situação se inverte para as filas de incrementos e decrementos: o mutador produz pedidos de incrementos e decrementos – como será visto adiante – e portanto tem acesso ao final das filas correspondentes (*botIncQueue* e *botDecQueue*). Já o

coletor consome os pedidos de incrementos e decrementos, e portanto tem acesso ao início das filas correspondentes (*topIncQueue* e *topDecQueue*).

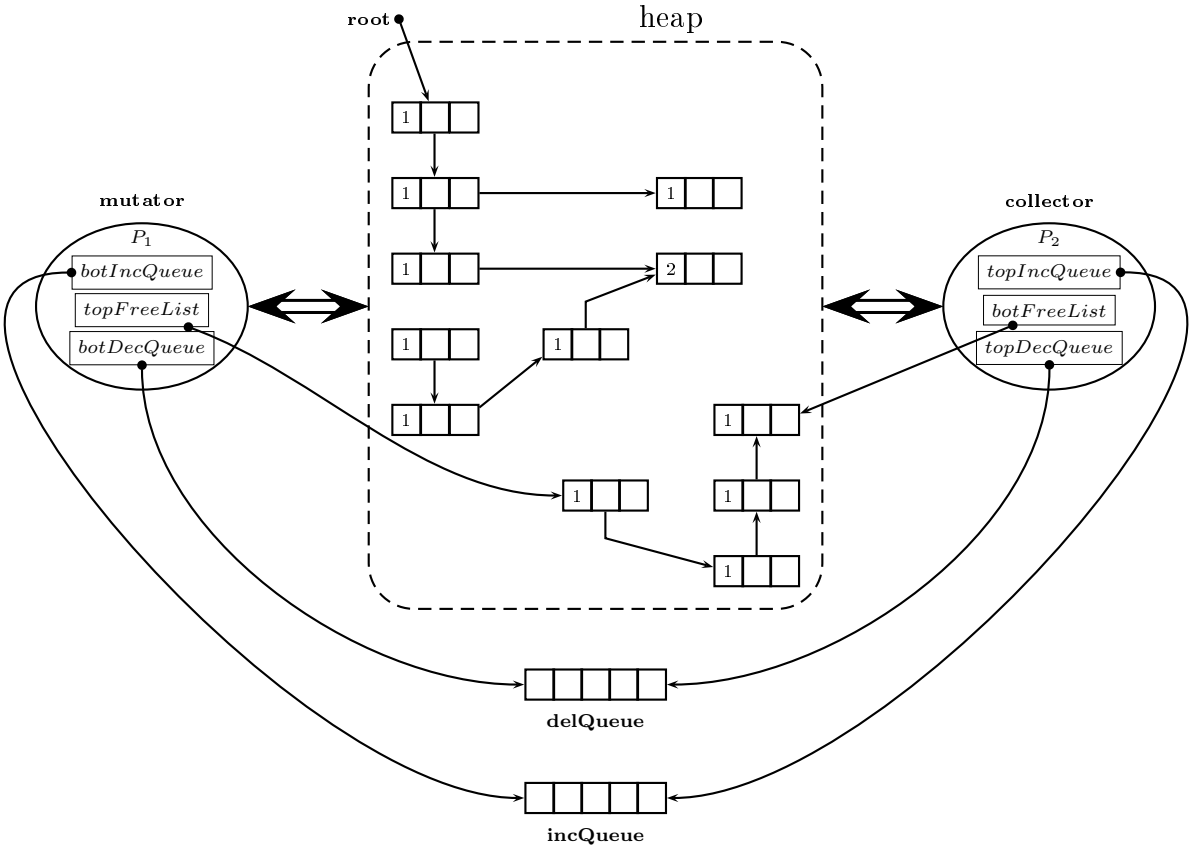

Figura 4.2: *Representação esquemática do algoritmo Formiga e Lins para um coletor e um mutador*

O mutador não altera diretamente os contadores de referência, mas inclui nestas duas filas pedidos de incremento e decremento do contador, conforme uma referência tenha sido criada ou destruída. Desta forma, só o coletor altera os contadores de referências, evitando quaisquer condições de corrida ou problemas de sincronização, e eliminando a necessidade de usar semáforos. Esta organização é similar ao *arquivo de transações* do coletor de Deutsch e Bobrow [28], mas com uma representação bastante eficiente e na memória principal. Com estas três estruturas em forma de fila, o mutador tem acesso apenas a uma das extremidades das filas, e o coletor acessa a outra extremidade; assim, para uma dada fila, apenas um dos dois processadores inclui itens, enquanto o outro apenas exclui itens da mesma. O mutador retira células da lista livre para utiliza-las, e inclui pedidos de incremento e decremento de contadores

de referências; o coletor retira e processa pedidos de incremento e decremento, e adiciona células liberadas à lista livre. A idéia geral desta arquitetura está expressa na Figura 4.2. Como é usual inserir itens no final de uma fila, e retirá-los no início, o mutador tem acesso ao final das filas de incremento e decremento (*botIncQueue* e *botDecQueue*, respectivamente), e ao topo da lista livre (*topFreeList*). Já o coletor tem acesso ao topo das filas de incremento e decremento (*topIncQueue* e *topDecQueue*), e ao final da lista livre (*botFreeList*). Para implementar estas filas de maneira concorrente, pode-se usar algum dos algoritmos conhecidos para filas *lock-free*, que são especialmente simples no caso de apenas um produtor e um consumidor [49, 94].

A decisão de usar duas filas para os pedidos de alteração nos contadores de referências foi motivada por questões de eficiência na representação: desta forma, um item em uma das filas é apenas o endereço da célula, já que a operação requerida no contador – incrementar ou decrementar – fica ímplicita na própria fila onde o item está localizado. Em geral, o tamanho de um endereço é igual ao tamanho da palavra, então cada item nessas filas ocupa exatamente uma palavra na memória. Caso fosse desejado usar uma fila apenas para todos os pedidos, seria necessário usar pelo menos um bit para especificar a operação de ajuste necessária, o que impediria ou, no mínimo, dificultaria a representação de cada item individual em uma palavra. Além disso, essa organização permite processar os incrementos antes dos decrementos, o que é importante para garantir a corretude do programa, como será visto adiante.

O processador P_1 é o mutador, e P_2 o coletor. O mutador está encarregado das alterações no grafo da memória – que decorrem das operações de computação útil – juntamente com a alocação de novas células da lista livre e a inclusão de pedidos para incremento e decremento de referências. O coletor, por sua vez, tem como tarefa o processamento dos pedidos de incremento e decremento de referências, alterando o valor dos contadores envolvidos, e a liberação efetiva das células, devolvendo-as para a lista livre. Estas operações são descritas mais formalmente abaixo.

Operações do mutador As operações do mutador, com relação ao gerenciamento de memória, são:

- ▷ New – aloca novas células
- ▷ Delete – gera um pedido para remoção de uma referência
- ▷ Update – altera uma referência

Em alguns algoritmos de contagem de referências se inclui também uma operação COPY que copia ou duplica uma referência, mas esta não é necessária aqui, sendo um caso particular da rotina UPDATE.

Como o algoritmo utilizado como base é o de contagem de referências cíclicas, as células incluem não só um contador de referências mas também uma cor, para ser usada pelo coletor; isso traz conseqüências para a sincronização que são similares às do contador de referências, como será explicado adiante. A coordenação da lista livre e das filas de incremento e decremento é feita guardando o endereço para uma das extremidades de cada uma delas no mutador, enquanto a outra é guardada no coletor, através dos pares de registradores como *topIncQueue* e *botIncQueue*.

A rotina para NEW aparece no Algoritmo 4.1. Primeiro verifica-se se a lista livre tem células disponíveis; se for o caso, a célula no topo da lista livre é obtida e retirada da lista livre. Aqui assume-se que a lista livre está organizada como uma lista duplamente encadeada e que o campo *prev* de uma célula na fila aponta para a célula anterior na fila; esta é uma representação comum para filas, e é aqui utilizada nas três filas do sistema. Caso não existam células na lista livre, o mutador entra em espera ocupada (*busy waiting*) até que o coletor devolva ao menos uma célula para a lista livre.

Algoritmo 4.1 Rotina para alocação de novas células

```
1: procedure NEW
2:     if topFreeList ≠ nil then
3:         newCell := topFreeList
4:         topFreeList := topFreeList.prev
5:     else
6:         newCell := NEW()
7:     end if
8:     returnnewCell
9: end procedure
```

Quando uma referência deve ser removida, é chamada a rotina DEL, mostrada no Algoritmo 4.2. A rotina simplesmente cria um pedido para incluir a célula S – para a qual uma referência está sendo removida – na fila de decrementos, chamando a rotina ADDDECREMENT; esta é encarregada de adicionar um item na fila de decrementos, usando o campo *next* para encadeamento.

Por fim, a rotina UPDATE é utilizada quando é necessário alterar um ponteiro R para apontar para uma célula S. O algoritmo para UPDATE está mostrado em 4.3. Primeiro,

Algoritmo 4.2 Algoritmo para remoção de referências

```
1: procedure DELETE(S)
2:     ADDDECREMENT(S)
3: end procedure
4: procedure ADDDECREMENT(S)
5:     S.next := topDelQueue
6:     topDelQueue := S
7: end procedure
```

remove-se uma referência para a célula apontada por R (representada aqui por $*R$); em seguida, insere-se um pedido para incrementar o contador de referência de S – através da chamada a ADDINCREMENT – e, por fim, atribui-se o endereço de S para o ponteiro R. Aqui é importante notar que um ponteiro é alterado, uma operação passível de sincronização, como visto na seção anterior. Entretanto, como só há um mutador, não é necessário sincronizar as alterações de ponteiros, pois o coletor nunca muda o valor de um ponteiro em uma célula ativa. Por fim, ADDINCREMENT apenas adiciona o endereço da célula em questão na fila de incrementos, de forma similar ao processo de inclusão de itens em todas as filas.

Algoritmo 4.3 Alteração de referências no algoritmo para um coletor e um mutador

```
1: procedure UPDATE(R, S)
2:     DELETE(*R)
3:     ADDINCREMENT(S)
4:     *R := S
5: end procedure
6: procedure ADDINCREMENT(S)
7:     S.next := topIncQueue
8:     topIncQueue := S
9: end procedure
```

Operações do coletor Como dito anteriormente, a função do coletor é processar os pedidos de alteração no contador de referências, e liberar as células que se tornem inativas – ou seja, cujo contador chega ao valor zero. Como o algoritmo para varredura local em busca de ciclos opera durante a remoção de referências, isso conseqüentemente é uma função do coletor também.

A operação principal do coletor é chamada de PROCESSQUEUES, que executa continuamente no processador P_2 como mostrado no Algoritmo 4.4; sua função principal é processar os pedidos de incremento e decremento nas filas. O primeiro passo da rotina é examinar a

fila de incremento: cada incremento encontrado na fila é retirado desta e causa o incremento do contador da célula apontada, e a alteração de sua cor para verde; isso é feito até que a fila esteja vazia. Só então ProcessQueues passa a tratar dos decrementos. Isso é feito para garantir que o sistema estará sempre à frente no processamento dos incrementos em relação aos decrementos, o que evita que uma célula com apenas uma referência, cujo contador vai ser incrementado e decrementado logo em seguida, seja liberada antes que o incremento seja visto, por causa de diferenças no escalonamento dos processadores. É importante notar que processar os incrementos à frente dos decrementos nunca deixará o sistema em algum estado inconsistente: uma célula que se torna inatingível nunca será incrementada além de zero, e será eventualmente liberada; e como os contadores são sempre incrementados primeiro, é obviamente impossível que uma célula ativa seja encarada como lixo. Supondo um programa que gere incrementos e decrementos aproximadamente na mesma taxa, o que é uma suposição bastante razoável, é de se esperar que o coletor processe primeiro todos os incrementos de um grupo, e depois os decrementos, mantendo sempre a corretude do sistema sem deixar de processar os decrementos por muito tempo. Entretanto, nesse algoritmo não há garantia contra inanição (*starvation*); isso é resolvido no algoritmo que utiliza fila de atualizações, mostrado na Seção 4.3.2. O processamento da fila de decrementos é mais simples: a rotina apenas verifica se a fila está vazia e, caso contrário, retira uma célula do fim da fila e chama a rotina RecDel para efetivamente remover a referência. Para tirar um item das filas de incremento e decremento, utiliza-se as rotinas auxiliares GetIncrement e GetDecrement, respectivamente; estas apenas retornam a célula no topo da fila correspondente, e alteram o topo para apontar para a célula anterior.

Se não houver pedidos de incremento nem de decremento, o coletor ficaria ocioso, e para aproveitar esse tempo ocioso a rotina ProcessQueues chama ScanStatusAnalyser para realizar varreduras locais em busca de ciclos de células inativas. Caso a varredura do *status analyser* ocorra com mais freqüencia do que desejado, pode ser adicionado um contador para só chamar ScanStatusAnalyser depois que as filas de incremento e decremento forem encontradas vazias um certo número de vezes. Esse tipo de ajuste de parâmetros pode ser feito pelo programador do sistema, que terá mais conhecimento para achar o valor mais adequado.

RecDel é a rotina para liberação recursiva de células inativas, mostrada na Figura 4.5. Ela funciona de uma forma similar à rotina Delete do algoritmo seqüencial para contagem

Algoritmo 4.4 Processamento das filas de incremento e decremento

```
1:  procedure ProcessQueues
2:      while botIncQueue ≠ nil do
3:          S := GetIncrement()
4:          S.rc := S.rc + 1
5:          S.color = green
6:      end while
7:      if botDecQueue ≠ nil then
8:          S := GetDecrement()
9:          RecDel(S)
10:     else
11:         ScanStatusAnalyser()
12:     end if
13:     ProcessQueues()
14: end procedure
15: procedure GetIncrement
16:     C := topIncQueue
17:     topIncQueue := topIncQueue.prev
18: end procedure
19: procedure GetDecrement
20:     C := topDecQueue
21:     topDecQueue := topDecQueue.prev
22: end procedure
```

de referências cíclicas (ver Algoritmo 3.4): se a célula tiver contador de referências com valor menor ou igual a 1[4], ela será liberada, mudando sua cor para verde, e ligando-a à lista livre, antes disso eliminando recursivamente as suas referências. Caso o contador tenha valor maior que 1, isso pode indicar que um ciclo de lixo foi gerado, e então a célula é adicionada ao *status analyser*, se já não estiver nele.

Algoritmo 4.5 Liberação recursiva no algoritmo para um mutador e um coletor

```
1:  procedure RecDel(S)
2:      if S.rc ≤ 1 then
3:          S.color := green
4:          for each T in S.children do
5:              RecDel(T)
6:          end for
7:          botFreeList.next := S
8:          botFreeList := S
9:      else
10:         if S.color ≠ black then
11:             S.color := black
12:             AddToStatusAnalyser(S)
13:         end if
14:     end if
15: end procedure
```

A varredura local em busca de ciclos é feita pela rotina ScanStatusAnalyser, que é igual à do caso seqüencial (Algoritmo 3.5). MarkRed, que é chamada por ScanStatusAnalyser, também é quase idêntica, mas por uma questão de sincronização há uma mudança no decremento do contador de referências. A rotina alterada está no Algoritmo 4.6, e muda apenas a linha onde ocorre o decremento dos contadores de referências. O mesmo acontece com ScanGreen, que só é alterada na linha onde o contador de referências é incrementado, como visto no Algoritmo 4.7. Por último, Collect permanece igual ao caso seqüencial (Algoritmo 3.8), embora altere também a contagem de referência e cor das células. O motivo é visto adiante, na análise da sincronização desta arquitetura.

Análise A arquitetura proposta difere da de Lins [68] principalmente na forma de tratar a interferência entre coletor e mutador: Lins recomenda tratar o problema através do uso de

[4] O teste do contador ser menor ou igual a 1 aqui faz sentido pois acessos concorrentes a referências entre o mutador (que pode alterá-las) e o coletor (que segue referências para procurar ciclos de lixo) pode causar com que MarkRed seja chamada em objetos com contador com valor 1, baixando-o para 0.

Algoritmo 4.6 Rotina para marcar células de vermelho no algoritmo para um coletor e um mutador

```
1:  procedure MarkRed(S)
2:      if S.color ≠ red then
3:          S.color := red
4:          for each T in S.children do
5:              T.rc := T.rc − 1
6:          end for
7:          for each T in S.children do
8:              if T.color ≠ red then
9:                  MarkRed(T)
10:             end if
11:             if T.rc > 0 ∧ T ∉ statusAnalyser then
12:                 AddToStatusAnalyser(T)
13:             end if
14:         end for
15:     end if
16: end procedure
```

Algoritmo 4.7 Rotina para varrer células pintadas de verde no algoritmo para um coletor e um mutador

```
1:  procedure ScanGreen(S)
2:      S.color := green
3:      for each T in S.children do
4:          T.rc := T.rc + 1
5:          if T.color ≠ green then
6:              ScanGreen(T)
7:          end if
8:      end for
9:  end procedure
```

sincronização explícita, utilizando semáforos, embora isso não seja detalhado na sua proposta; já a arquitetura apresentada acima utiliza as técnicas de *variáveis disjuntas* – pois coletor e mutador não concorrem no acesso de variáveis na memória – e *enfraquecimento de asserções* – pois o invariante da contagem de referências é enfraquecido – para lidar com a interferência [3], o que evita o uso de semáforos e melhora a eficiência do algoritmo concorrente resultante. De fato, o sistema de gerenciamento de memória não utiliza nenhuma operação de sincronização, em comparação com a versão clássica do algoritmo concorrente de contagem de referências [27], que precisa obter um semáforo global associado a todas as *threads*. Considera-se atualmente que o uso de semáforos afeta negativamente o desempenho do sistema a ponto de tornar-se proibitivo [26, 27, 61]; há também o problema de usar apenas um semáforo para todas as *threads*, mas isto não afeta a arquitetura em questão pois está previsto apenas um mutador. Entretanto, este algoritmo é suscetível à inanição do mutador se o coletor estiver sempre processando pedidos de incrementos e ficar impedido de processar os pedidos de decrementos. Apesar de ser uma preocupação quanto à corretude do algoritmo, esse tipo de situação não foi observada nos testes experimentais com a sua implementação. O algoritmo baseado em fila de referências, apresentado na Seção 4.3.2, a seguir, não apresenta esse problema.

Com relação ao aproveitamento dos recursos computacionais, é plausível que o processador do coletor fique ocioso se as filas de incremento e decremento estiverem vazias e não houver células no *status analyser*. Entretanto, esse problema concerne mais ao escalonador do sistema operacional, que pode utilizar o processador alocado para o coletor em outras tarefas.

Uma otimização interessante que é possibilitada por essa arquitetura é identificar alterações redundantes nos contadores de referências. Por exemplo, se houver um pedido de incremento para uma célula C na fila de incremento, e um pedido de decremento para a mesma célula na fila de decremento, o valor do contador de C não deve ser alterado. Se ao invés de processar os pedidos um de cada vez a rotina ProcessQueue analisar um grupo de pedidos nas duas filas, é possível cancelar pedidos e mesmo alterar os contadores de forma mais eficiente. Uma sugestão para implementar isso é obter um lote de itens nas duas filas e processá-los conjuntamente, usando uma tabela *hash* cuja chave é o endereço da célula e o valor associado é o valor final do contador de referência; cada pedido processado altera o valor associado na tabela hash, sem precisar alterar a célula em si. Ao final de um lote, pode-se enumerar os itens na tabela e mudar de uma vez os contadores de referência para seus valores finais após o lote. Isso evitará alterações redundantes e pode ser mais eficiente mesmo sem a

presença de redundância, pois a tabela *hash* deverá apresentar melhor localidade de referência no coletor.

Outra possibilidade para reduzir a quantidade de atualizações nos contadores de referência é apresentada na seção a seguir.

4.3.2 Uma Variante Baseada em Fila de Atualizações

Ao invés de registrar diretamente incrementos e decrementos, como no algoritmo apresentado anteriormente, uma alternativa é usar uma *fila de atualizações*. Nessa variante, há apenas uma fila entre o mutador e o coletor, a fila de atualizações, como mostrado na Figura 4.3. Essa variante do algoritmo que usa a fila de atualizações tem desempenho melhor que o algoritmo de Formiga e Lins [39], apresentado na Seção 4.3.1. Além disso, os conceitos e consequências relacionadas ao uso da fila de atualizações são utilizados no algoritmo para vários mutadores e um coletor, detalhado na Seção 5.3.

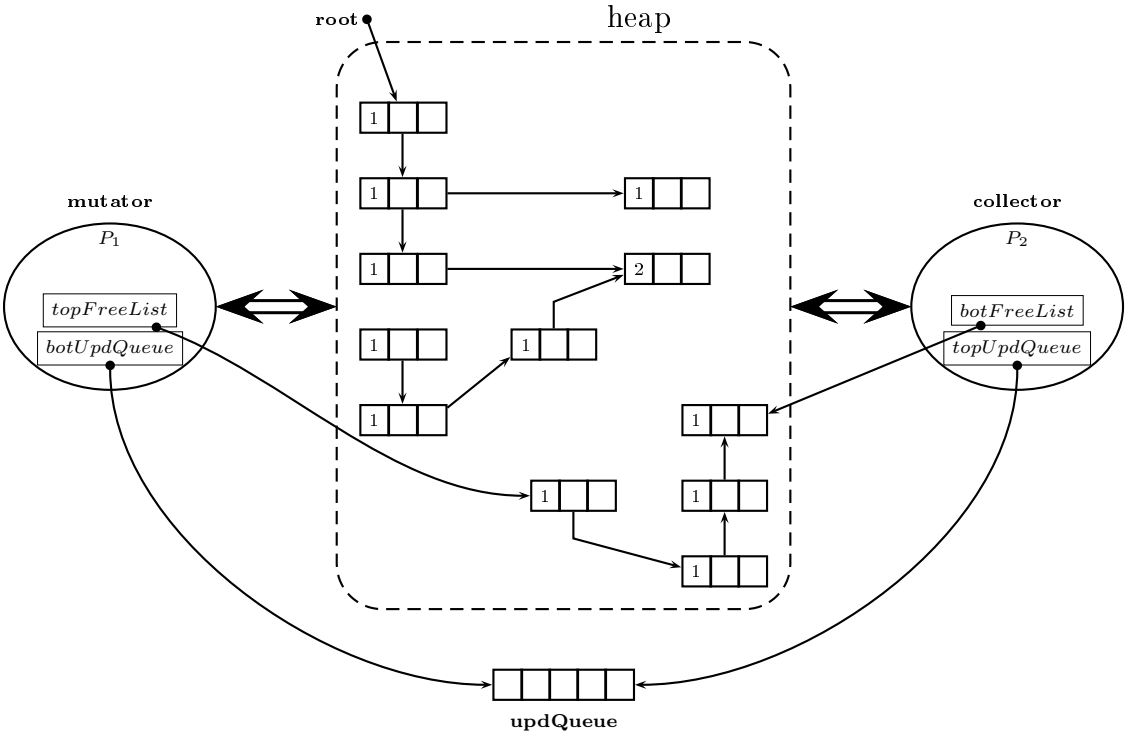

Figura 4.3: *Algoritmo de contagem de referências cíclicas para um coletor e um mutador, usando fila de atualizações*

A fila de atualizações contém registros da operação UPDATE no grafo de objetos na memória. Uma operação UPDATE altera o valor de uma referência r[5], que contém um valor o_1, para um novo valor o_2. Os valores o_1 e o_2 referenciam objetos O_1 e O_2 no *heap*; a alteração significa que o objeto O_1 terá seu contador de referências decrementado, enquanto O_2 terá seu contador de referências incrementado (ver Algoritmo 4.3). A vantagem de processar registros de atualização, em comparação a pedidos de incremento e decremento, é que os primeiros contêm estritamente mais informação que os últimos: dos registros de atualização é possível reconstruir a sequência de incrementos e decrementos, mas o contrário não acontece. Além disso, usando o registro de atualizações é mais fácil simplificar uma sequência de incrementos e decrementos, como será visto adiante.

Podemos representar um registro de atualização como uma tripla $\langle r, o_1, o_2 \rangle$, onde a referência r é alterada de o_1 para o_2. A remoção completa de uma referência r pode ser representada por uma tripla $\langle r, o_1, nil \rangle$, e a criação de uma nova referência (alocando um novo objeto ou apontando para um objeto já existente) por uma tripla $\langle r, nil, o_1 \rangle$. Sequências de registros de atualização podem ser simplificados facilmente se observarmos algumas características da operação UPDATE no contexto do algoritmo para um coletor e um mutador. Se fixarmos uma referência r, a relação de alterar r de um valor o_1 para um valor o_2 é transitiva, ou seja:

$$\langle r, o_1, o_2 \rangle \wedge \langle r, o_2, o_3 \rangle \Rightarrow \langle r, o_1, o_3 \rangle,$$

e, neste caso, apenas os contadores de referência de o_1 e o_3 precisam ser alterados. Se acontecer um ciclo de atualizações em uma mesma referência, nenhum contador precisa ser alterado. Por exemplo, $\langle r, o_1, o_2 \rangle$, $\langle r, o_2, o_3 \rangle$ e $\langle r, o_3, o_1 \rangle$; neste caso, os três registros podem ser ignorados.

Como só há um mutador e uma fila, a sequência de registros de atualização na fila deve seguir a ordem em que as operações de alteração das referências foi realizada pelo mutador, ou seja, não há a chance de um decremento ser processado antes de um incremento correspondente. Também não é possível que ocorram sequências inconsistentes de registros de atualização, como $\langle r, o_1, o_2 \rangle$ e $\langle r, o_1, o_3 \rangle$.

Se essas características dos registros de atualizações forem utilizadas diretamente, é possível construir um algoritmo de coleta que simplifica os registros e reduz o número de alterações que precisam ser realizadas nos contadores de referências. O problema é que isso envolve buscas nos registros recebidos pela fila de atualização, para identificar as simplificações possíveis.

[5] Neste caso, a referência r representa um ponteiro armazenado em um objeto na memória, por exemplo um objeto Java ou uma célula de uma máquina de combinadores; r não representa simplesmente um endereço, mas um espaço em memória onde um endereço pode ser armazenado.

Uma observação importante aqui é que para o propósito de atualizar os contadores de referências de objetos apontados por uma referência r em um intervalo de tempo entre os instantes t_1 e t_2, apenas é necessário saber qual o valor de r em t_1 e qual o valor de r em t_2. Todas as atualizações que ocorrem em r entre t_1 e t_2 podem ser ignoradas: se r tem valor o_1 em t_1 e tem valor o_2 em t_2, o_1 aponta para O_1 e o_2 aponta para o objeto O_2, então o contador de O_1 precisa ser decrementado, e o contador de O_2 precisa ser incrementado. Se todas as referências existentes entre t_1 e t_2 forem rastreadas dessa forma, não há prejuízo para a recuperação de objetos que se tornam inacessíveis, e o número de operações de incremento e decremento pode ser bem menor. Se não houver nenhum registro de atualização para r entre os tempos t_1 e t_2, nada precisa ser alterado.

Para aproveitar essa observação, o algoritmo pode tomar *amostras* da situação das referências em momentos específicos. Como no algoritmo de contagem de referências não há, normalmente, um momento específico onde a coleta ocorre, o algoritmo adota o conceito de *ciclo de coleta* encontrado em outros coletores. Como a operação do coletor é repetir os mesmos passos em *loop* (processar filas, rastrear o *status analyser*, ver Algoritmo 4.4), fica definido um ciclo de coleta como o espaço de tempo entre o início do processamento da fila de atualizações em uma iteração e o início do processamento da fila na iteração seguinte. Neste caso, seguindo as observações feitas, é preciso que o algoritmo tenha acesso à situação das referências nos dois momentos: o tempo t_1 sendo o início do i-ésimo ciclo de coleta, e o tempo t_2 sendo o início do ciclo $i+1$. Com essas informações, é possível determinar quais devem ser os valores dos contadores de referências de todos os objetos no início de um ciclo.

A fila de atualização é organizada de forma que, em cada ciclo de coleta, só exista no máximo um registro de atualização para cada referência. Assim, se em um ciclo ocorrerem as atualizações $\langle r, o_1, o_2\rangle$ e $\langle r, o_2, o_3\rangle$ em r, deverá existir na fila de atualizações deste ciclo apenas o registro da primeira atualização. Para evitar inserir registros de atualização para uma referência que já foi atualizada no ciclo atual, pode-se usar um *flag* binário, *updated*, para isso. $updated(r)$[6] terá valor verdadeiro quando r já tiver sido atualizada no ciclo atual. Com um único registro na fila de atualizações para cada referência atualizada e o uso do *flag*, é possível reconstruir o valor de todas as referências entre um ciclo de coleta e o próximo,

[6] Pela notação empregada, $updated(r)$ é indicada como uma função cujo domínio é o conjunto de todas as referências possíveis, e não um campo da referência r; isso ocorre porque r é normalmente implementada como um ponteiro armazenado em uma palavra na memória, e muitas vezes o único valor que essa referência tem é o endereço do objeto referenciado. Pode-se usar um bit deste ponteiro para indicar o *flag*, o que seria similar a ter um campo da referência, mas outras implementações são possíveis, por exemplo uma estrutura de *mapa de bits*.

como será visto a seguir.

Uma invariante deste algoritmo é que *updated*(r) é verdade se, e somente se, há um registro na fila de atualizações que contém a referência r. Para garantir que o conteúdo da fila e os os *flags* sejam observados e atualizados sem interferência, o coletor deverá suspender o *thread* mutador enquanto opera nessas estruturas. A pausa imposta, entretanto, será curta.

Apenas duas operações precisam ser alteradas nessa variante, em relação ao algoritmo apresentado na Seção 4.3.1: a operação UPDATE e a operação PROCESSQUEUES, que é o programa principal do coletor. No mutador, a operação NEW continua a mesma (ver Algoritmo 4.1), mas DELETE é desnecessária; quando é necessário remover uma referência r sem alterá-la para outro valor, deve-se chamar UPDATE(r, **nil**). No coletor, as demais operações além de PROCESSQUEUES (que muda de nome para COLLECTORMAIN) continuam as mesmas vistas na Seção 4.3.1.

UPDATE precisa verificar se a referência já foi alterada (pelo *flag updated*) e, em caso negativo, precisa incluir um registro de atualização na fila e marcá-la como atualizada. O Algoritmo 4.8 mostra a operação UPDATE; a operação ADDUPDATE(R, S) adiciona um registro de atualização na fila que contém a referência R, o valor atual de R, e o novo valor S. Como só existe um mutador, não há nenhuma possibilidade de interferência nessas operações, e portanto nenhuma delas precisa ser sincronizada (nem mesmo o acesso à fila, já que o coletor suspende o mutador para o conteúdo da fila). Isso significa que a atualização de referências – uma operação que ocorre com frequência na maioria dos sistemas – não precisa de nenhuma sincronização para executar. Essa é uma das vantagens de usar uma fila de atualizações.

Algoritmo 4.8 Alteração de referências no algoritmo para um coletor e um mutador, usando uma fila de atualizações

```
1: procedure UPDATE(R, S)
2:     if ¬updated(R) then
3:         ADDUPDATE(R, S)
4:         updated(R) := true
5:     end if
6:     R := S
7: end procedure
```

O programa principal do coletor, antes chamado de PROCESSQUEUES (Algoritmo 4.4), aqui é chamado de COLLECTORMAIN, mostrado no Algoritmo 4.9. Inicialmente, o coletor suspende o mutador, copia o conteúdo da fila de atualizações para um *buffer* local, remove

todos os itens da fila (ClearQueue) e zera todos os *flags* de atualização. Isso marca o início de um novo ciclo de coleta. A seguir, o mutador é sinalizado para continuar sua execução, e o coletor atualiza os contadores de referência dos objetos afetados pelas atualizações no *buffer*. Por fim, o coletor faz o rastreamento dos possíveis ciclos isolados chamando ScanStatusAnalyser, e reinicia sua operação com a chamada recursiva.

Algoritmo 4.9 Programa principal do coletor para algoritmo com fila de atualizações

```
1:  procedure CollectorMain
2:      Suspend the mutator
3:      CopyQueueToBuffer(buffer)
4:      ClearQueue()
5:      ClearUpdatedFlags(buffer)
6:      Resume the mutator
7:      UpdateReferenceCounters(buffer)
8:      ScanStatusAnalyser()
9:      CollectorMain()
10: end procedure
```

Da operação do coletor em CollectorMain, a atualização dos contadores de referências precisa ser detalhada. Como discutido antes, se for conhecido o valor da referência r em t_1 (início de um ciclo de coleta) e em t_2 (início do próximo ciclo de coleta), basta decrementar o contador do objeto apontado por r em t_1 e incrementar o contador do objeto apontado por r em t_2. A operação UpdateReferenceCounters trabalha para estabelecer essas informações relativas ao ciclo que terminou. Se r não foi alterada no ciclo anterior, não há nenhum registro de atualização contendo r no *buffer*, e nada precisa ser feito; caso contrário, se r foi alterada, há exatamente um registro $\langle r, o_1, o_2 \rangle$ no *buffer*. O valor de r em t_1, início do ciclo anterior, é o_1 (já que o registro grava a primeira alteração em r). Entretanto, o valor de r em t_2, fim do ciclo anterior e início do atual, não é necessariamente o_2, pois outras alterações podem ter sido feitas em r sem gerar um registro de atualização (pois o *flag* $updated(r)$ já estava marcado). Neste caso, a solução é olhar para o valor atual de r e do *flag* $updated(r)$. UpdateReferenceCounters é chamada depois que o mutador é suspenso, então o valor atual de r pode não ser o mesmo do início do ciclo atual (quando o mutador estava suspenso). Para isso é preciso verificar $updated(r)$: se o *flag* for falso, o valor de r não mudou desde o início do ciclo, e portanto o valor atual é o valor de r em t_2; caso contrário, o valor de r mudou, mas o valor no início do ciclo está gravado na fila de atualizações como o valor antigo de r. A operação UpdateReferenceCounters é mostrada no Algoritmo 4.10. A

ordem das operações é primeiro ler o valor atual de r e depois verificar o *flag* $updated(r)$ pois isso garante que o valor correto será lido sem necessidade de sincronização, observando-se a ordem das operações que podem ocorrer no mutador, em UPDATE (ver Algoritmo 4.8). Em UPDATEREFERENCECOUNTERS, a rotina GETOLDVALUEFROMQUEUEr varre a fila de atualizações procurando por um registro $\langle r, o_i, o_j \rangle$ e retorna o_i, o valor anterior de r gravado no registro; neste caso, esse é o valor de r ao final do ciclo anterior. O acesso à fila pode ser feito sem sincronização, já que o registro buscado está na fila e isso não muda até o próximo ciclo de coleta.

Algoritmo 4.10 Atualização dos contadores de referência no algoritmo com fila de atualizações

```
1:  procedure UPDATEREFERENCECOUNTERS(buffer)
2:      for each ⟨r, o1, o2⟩ in buffer do
3:          curObj := r
4:          if updated(r) then                    ▷ curObj não é o valor atual de r
5:              curObj := GETOLDVALUEFROMQUEUE(r)
6:          end if
7:          curObj.rc := curObj.rc + 1
8:          RECDEL(o1)                            ▷ Remove referência a o1
9:      end for
10: end procedure
```

O algoritmo usando fila de atualizações, implementado na mesma plataforma de testes descrita no artigo de Formiga e Lins [39], mostrou um desempenho superior à versão original do artigo, impondo pausas muito pequenas ao tempo de resposta do mutador. A implementação dos algoritmos descritos e os testes de desempenho, juntamente com seus resultados, são descritos nas seções a seguir.

4.4 Implementação e Plataforma de testes

Os algoritmos de contagem de referências com um coletor e um mutador anteriores ao algoritmo de Formiga e Lins ([66, 68]) nunca foram implementados. A fim de analisar o desempenho dos novos algoritmos, foi criada uma plataforma de testes que permitisse a implementação e a comparação direta de diferentes algoritmos de coleta de lixo, incluindo algoritmos de contagem de referência, sequenciais ou multiprocessados.

A plataforma de testes implementada consiste em um compilador para uma linguagem funcional simples. Na época em que os testes foram realizados, julgou-se necessário imple-

mentar todo um compilador para garantir a execução concorrente em multiprocessadores, pois vários compiladores então não garantiam que os *threads* do programa do usuário seriam executadas pelos diferentes processadores em um computador multiprocessado. Os resultados dos testes que comparam o desempenho do algoritmo de Formiga e Lins em relação às alternativas existentes na época podem ser encontrados no artigo que descreve tal algoritmo [39]. Esta seção mostra os resultados de testes similares realizados para determinar o desempenho do algoritmo usando fila de atualizações apresentado na Seção 4.3.2.

O compilador implementado gera código para a máquina-G [89] e então traduz este código para código nativo para os processadores Intel IA-32. O Apêndice A descreve a linguagem utilizada e o compilador em maiores detalhes. Para os propósitos deste capítulo, os fatos importantes são que o compilador gera código nativo e executa as operações de uma máquina de redução de grafos; as atividades de redução no grafo são responsáveis por gerar células inativas, criando a necessidade de um coletor de lixo. Além disso, a implementação de funções recursivas na redução de grafos cria ciclos, que não podem ser recuperados com o algoritmo tradicional de contagem de referências.

Os programas de teste (ver Apêndice B) foram executados em um computador com um processador Intel Core 2 Duo de $2.1GHz$, com dois núcleos de processamento (*cores*) e 2Gb de memória RAM. O sistema operacional utilizado foi o Ubuntu Linux versão 10.04. Três versões principais foram comparadas nesta plataforma: o algoritmo de Lins [68], o algoritmo de Formiga e Lins [39], e o algoritmo com a fila de atualizações da Seção 4.3.2. Para obter uma base de comparação, foi também testado o algoritmo sequencial de contagem de referências cíclicas.

Os programas de teste foram selecionados para realizar medições de desempenho por causarem a geração de muitos ciclos durante a execução; como a máquina G implementa funções recursivas através do combinador Y, que forma um ciclo no grafo da expressão a ser reduzida, a maioria dos programas utilizados geram muitas chamadas recursivas às funções principais. Também foram testados programas que geram grandes estruturas de listas que não são recursivas nem cíclicas. As cargas de trabalho utilizadas para cada teste podem ser vistas no texto completo dos programas, incluído no Apêndice B.

4.5 Resultados e análises

Cada um dos seis programas foi executado 15 vezes por versão, medindo-se o tempo de cada execução, e calculou-se a média dos 15 tempos obtidos. Os resultados são mostrados na Tabela 4.1, na qual as colunas recebem nomes de acordo com o algoritmo testado:

- ▷ **rc-seq**: Algoritmo sequencial de contagem de referências cíclicas, descrito no Capítulo 3
- ▷ **rc-lins**: Algoritmo de Lins [68]
- ▷ **rc-fl**: Algoritmo de Formiga e Lins [39]
- ▷ **rc-upd** Algoritmo da Fila de Atualizações, descrito na Seção 4.3.2.

Todos os tempos indicados nas tabelas a seguir estão em segundos.

Para analisar em maiores detalhes o desempenho das diferentes algoritmos, foram coletados dados do perfil de execução de cada programa de teste, incluindo o número de chamadas aos procedimentos do algoritmo de contagem de referências. Um sumário dos dados de perfil de execução coletados está apresentado na Tabela 4.2, na qual cada linha, exceto a última, indica a média do número de chamadas de algum procedimento do algoritmo de contagem de referências cíclicas:

- ▷ Alloc: o número total de células alocadas durante o programa;
- ▷ ScanSA: o número de chamadas à operação ScanStatusAnalyzer;
- ▷ MarkRed: o número de chamadas à operação MarkRed;
- ▷ ScanGreen: o número de chamadas à operação ScanGreen;
- ▷ Collect: o número de chamadas à operação Collect;
- ▷ Update: o número de chamadas à operação Update, a barreira de escrita.

A média mostrada na Tabela 4.2 foi tomada em relação ao número de chamadas realizadas em cada teste para o mesmo coletor; por exemplo, o número de chamadas a Alloc para o algoritmo *rc-seq* na Tabela 4.2 representa a média dos números de chamadas a Alloc para os oito programas de teste executados. A última linha da Tabela 4.2 mostra os tempos médios de execução para os oito programas de teste em cada algoritmo, sendo uma média dos tempos mostrados na Tabela 4.1.

Tabela 4.1: *Tempos de execução para os programas de teste, em segundos*

Teste	*rc-seq*	*rc-lins*	*rc-fl*	*rc-upd*
acker	0.6820	0.4120	0.3850	0.2010
conctwice	0.3530	0.1990	0.1769	0.1156
fiblista	3.8922	3.0921	2.7230	2.0150
recfat	0.7610	0.4888	0.4612	0.4070
somamap	0.4712	0.2373	0.2198	0.2040
somatorio	0.4173	0.1474	0.1102	0.0983
tak	9.9020	8.4613	8.3440	7.3990
queens	5.9323	5.6049	4.8091	4.0319

Tabela 4.2: *Perfil de execução dos testes para os algoritmos de contagem de referências. Os valores são médias obtidas da execução dos programas de teste da Tabela 4.1.*

Chamada	*rc-seq*	*rc-lins*	*rc-fl*	*rc-upd*
ALLOC	3199869	3199869	3199869	3199869
SCANSA	17754	17881	17913	17806
MARKRED	161473	162112	161772	162833
SCANGREEN	157755	157318	156992	157122
COLLECT	3718	3699	3721	3732
UPDATE	4877902	4867711	4879665	3792718
Tempo (s)	2.8014	2.3303	2.1536	1.809

Pode-se perceber na Tabela 4.2 que o número de chamadas à maioria das operações foi bastante similar em todos os algoritmos testados. A exceção é a operação UPDATE, que foi chamada muito menos no algoritmo da fila de atualizações. Isto ocorre pois o processamento da fila reduz o número de operações individuais de atualização de ponteiros, assim como o número de operações de incremento e decremento dos contadores de referência que precisam ser feitas. Os dados indicam que isso se traduz em um melhor desempenho geral do algoritmo, como pode ser visto na Tabela 4.2. Análises comparativas baseadas nos dados obtidos podem ser realizadas para aumentar a confiança na relação entre os desempenhos dos algoritmos estudados, como feito a seguir.

4.5.1 Análise Comparativa

Para analisar melhor as diferenças nos tempos, a Tabela 4.3 mostra as diferenças relativas entre os três algoritmos de contagem de referências para sistemas multiprocessados, juntamente com a versão sequencial, para identificar o impacto das alterações no gerenciamento de

Tabela 4.3: *Diferença relativa entre os tempos de execução das versões diferentes*

Teste	*rc-lins/rc-seq*	*rc-fl/rc-lins*	*rc-upd/rc-fl*
acker	0.3959	0.0655	0.4779
conctwice	0.4362	0.1110	0.3465
fiblista	0.2056	0.1194	0.2600
recfat	0.3577	0.0565	0.1176
somamap	0.4964	0.0737	0.0719
somatorio	0.6468	0.2524	0.1079
tak	0.1455	0.0139	0.1136
queens	0.0552	0.1419	0.1616

memória.

Os números na coluna *rc-lins/rc-seq* da Tabela 4.3 foram calculados segundo a fórmula

$$\frac{t_{rc-lins} - t_{rc-seq}}{t_{rc-seq}}$$

onde $t_{rc-lins}$ é o tempo de execução do programa correspondente usando o algoritmo de Lins [68], e t_{rc-seq} é o tempo para a versão do programa usando o algoritmo sequencial apresentado no Capítulo 3; os tempos são obtidos para o programa em questão a partir da Tabela 4.1. Ou seja, esta coluna mostra o quanto o algoritmo sequencial é mais lento em relação à versão para sistemas multiprocessados do algoritmo de Lins [68].

De forma similar, os números na coluna *rc-fl/rc-lins* foram calculados pela fórmula

$$\frac{t_{rc-fl} - t_{rc-lins}}{t_{rc-lins}}$$

e portanto indicam o quanto o algoritmo de Formiga e Lins é mais rápido, para os programas de teste, que o algoritmo de Lins. Por fim, a última coluna foi calculada com o uso da fórmula

$$\frac{t_{rc-upd} - t_{rc-fl}}{t_{rc-fl}}$$

e indicam o quanto o algoritmo da fila de atualizações da Seção 4.3.2 é mais rápido que o algoritmo de Formiga e Lins, para os testes em questão. A Figura 4.4 mostra a média dessas relações da Tabela 4.3 em um gráfico de barras. Pelo gráfico, é possível ver que o algoritmo de Lins é, em média, 30% mais rápido que o algoritmo sequencial, que o algoritmo de Formiga e Lins é 10% mais rápido que o de Lins, na média, e que o novo algoritmo baseado em filas de atualização é mais que 20% mais rápido que o algoritmo de Formiga e Lins, na média. A última coluna inclui uma comparação entre o algoritmo usando filas de atualização e o algoritmo sequencial (não mostrado na Tabela 4.3).

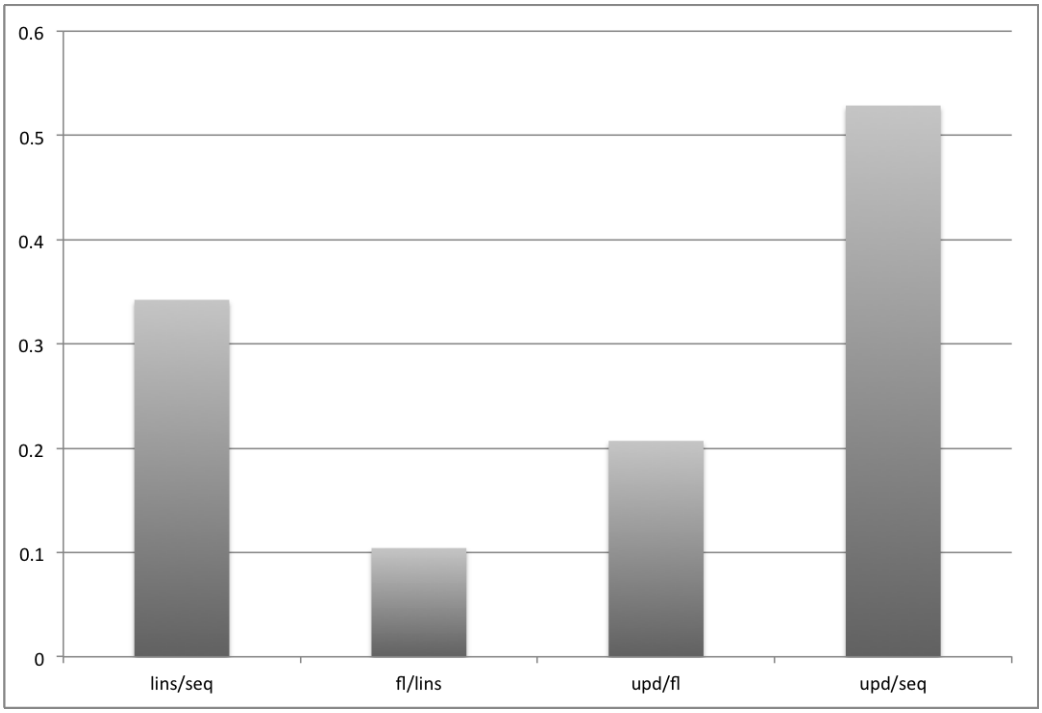

Figura 4.4: *Média das relações de desempenho entre as diferentes versões do algoritmo de contagem de referências cíclicas testadas.*

Com base na Tabela 4.2 e Figura 4.4, uma primeira observação que pode ser feita é que os algoritmos adequados para sistemas multiprocessados são consistentemente mais eficientes do que o algoritmo sequencial, nos testes – chegando a diferenças de mais de 50%. Em um computador com mais de um processador ou núcleo de processamento disponível, isso é o que se espera que aconteça, já que o algoritmo sequencial só usa um processador. O algoritmo de Formiga e Lins se mostra consistentemente superior ao algoritmo anterior de Lins, adicionando evidências à hipótese que a menor necessidade de operações de sincronização no algoritmo de Formiga e Lins tem um impacto positivo no seu desempenho. Por fim, o algoritmo com a fila de atualizações tem o melhor desempenho dos algoritmos testados, chegando a mostrar melhorias de mais de 40% em relação ao desempenho do algoritmo de Formiga e Lins. O uso da fila de atualizações reduz significantemente o número de chamadas à barreira de escrita para alterações com ponteiros (o procedimento UPDATE), como pode ser visto na Tabela 4.2, e isso tem um impacto positivo bastante significativo no seu desempenho. Todos os programas alocam uma grande quantidade de células e geram vários ciclos, por meio de chamadas recursivas. Isso indica que o sistema de gerenciamento de memória é exigido praticamente durante toda a execução do programa, afetando em muito os números

de desempenho. Em programas que utilizam mais a entrada e saída, a diferença deve se mostrar bem menor.

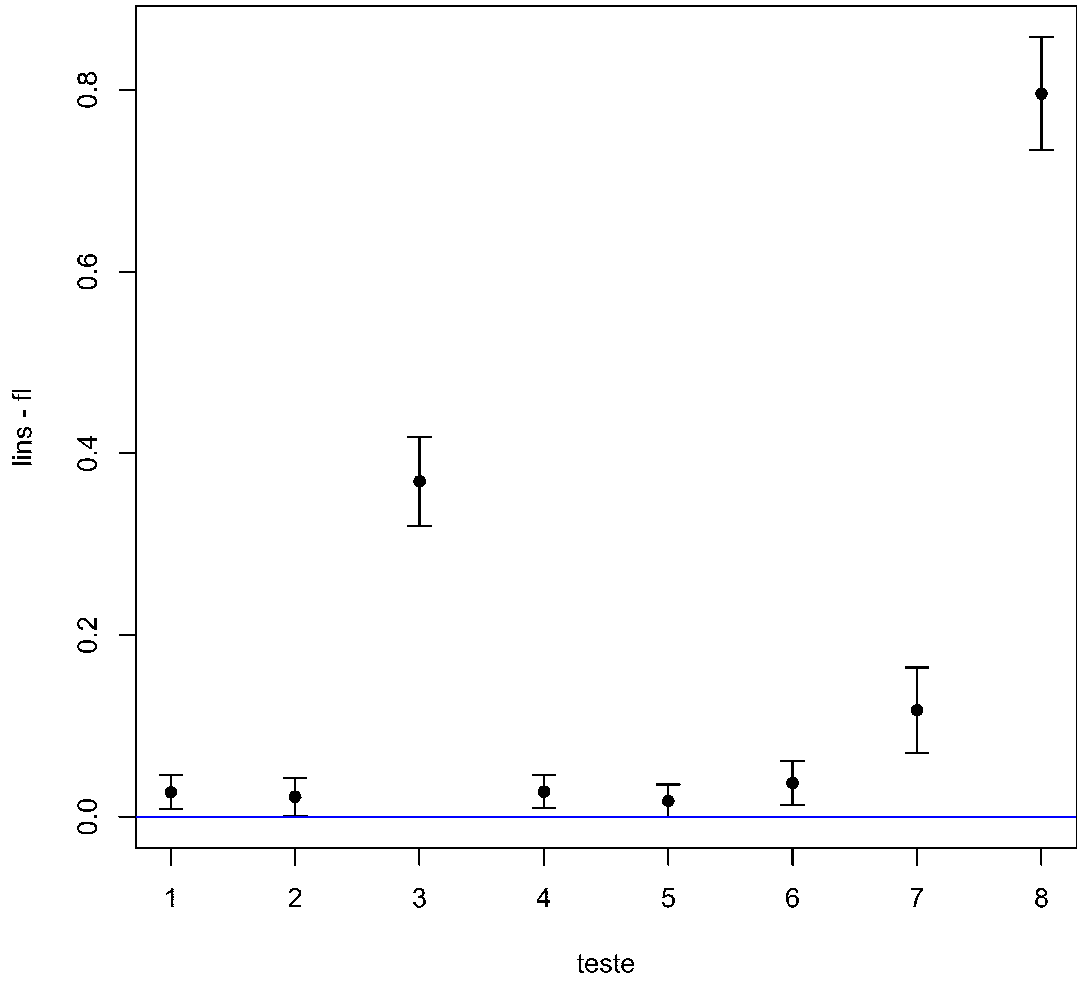

Figura 4.5: *Intervalos de confiança para a diferença entre as médias dos tempos de execução dos programas de teste usando o algoritmo de Lins e o Formiga e Lins. Os programas de teste de 1 a 8 estão na ordem mostrada na Tabela 4.1.*

Uma consideração importante na análise desses resultados é que os programas testados são curtos e apresentam cargas de trabalho sintéticas, sem uma a preocupação de representar cargas que seriam obtidas em aplicações reais. Os programas cujos resultados constam na Tabela 4.1, entretanto, mostram uma tendência consistente, o que indica uma tendência de diferença entre o desempenho dos algoritmos. Para aumentar a confiança nessa última constatação, pode-se realizar uma análise estatística baseada na construção de um intervalo

de confiança para o tempo total de execução dos programas de teste [51].

Como mencionado anteriormente, os tempos mostrados na Tabela 4.1 são médias aritméticas dos valores medidos em 15 execuções repetidas de cada programa. Através da média e do erro padrão calculado através da dispersão da amostra, podemos obter um intervalo de confiança para cada uma das médias, em um nível de confiança selecionado. Segundo o Teorema Central do Limite, médias de medições de variáveis aleatórias tendem a uma distribuição Normal, quando a amostra é grande [96]. Não há uma definição matemática precisa do tamanho da amostra necessária, mas como uma regra prática para medições de desempenho, diz-se que a amostra é suficientemente grande para usar o Teorema Central do Limite quando são feitas 30 ou mais observações (vide o livro de Jain [51], Capítulo 13). Com menos de 30 observações, pode-se calcular o intervalo de confiança usando a distribuição t, mas apenas se a quantidade amostrada seguir uma distribuição Normal.

Como não há nenhuma indicação fundamentada que os tempos medidos seguem uma distribuição Normal, o mais indicado é usar uma técnica de inferência não-paramétrica [96]. No presente caso, foi selecionada uma técnica baseada no método de *bootstrap*, os intervalos de confiança sendo construídos como intervalos pivotais de *bootstrap* em nível de 99% com 1000 replicações [96]. Os cálculos foram realizados com o *software* de computação estatística R [24], que também foi usado para gerar os gráficos que mostram os intervalos de confiança (Figuras 4.5 e 4.6).

Foram realizadas duas comparações diretas: a primeira entre o algoritmo original de Lins com o algoritmo Formiga e Lins; e a segunda entre o algoritmo Formiga e Lins e o algoritmo baseado em fila de atualizações. A Figura 4.5 mostra o resultado da diferença das médias de tempo de execução entre o algoritmo de Lins e o Formiga e Lins, e a Figura 4.6 mostra a comparação entre o algoritmo Formiga e Lins e o algoritmo com a fila de atualização. Como a comparação é feita através de uma comparação entre médias, se o intervalo de confiança incluir o valor 0, pode-se concluir (no nível de confiança de 99%) que não há diferença entre as quantidades medidas. Em ambas figuras de comparação, verifica-se que a maioria dos intervalos calculados não contém o valor 0, o que indica que há, de fato, uma diferença de desempenho nas duas comparações, embora em alguns casos a diferença seja bastante pequena. Isso também se deve ao fato dos programas de teste serem pequenos e executarem em um curto espaço de tempo.

A comparação baseada em programas sintéticos de teste pode chegar a conclusões que

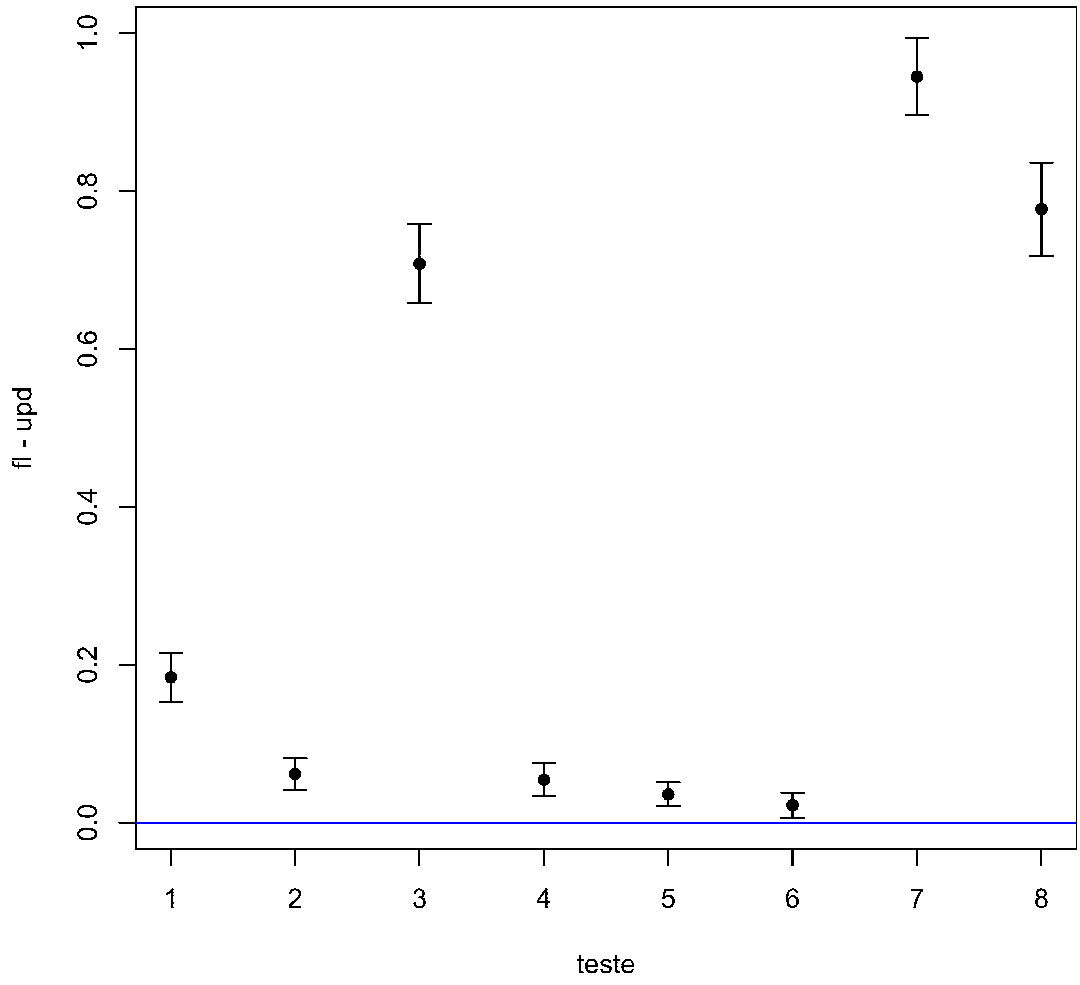

Figura 4.6: *Intervalos de confiança para a diferença entre as médias dos tempos de execução dos programas de teste usando o algoritmo de Formiga e Lins e o algoritmo com fila de atualização. Os programas de teste de 1 a 8 estão na ordem mostrada na Tabela 4.1.*

não são aplicáveis em situações reais de aplicação do sistema. Além disso, seria desejável medir o desempenho dos algoritmos em uma plataforma de uso corrente. Para realizar ambas as ideias e obter dados de desempenho dos algoritmos de contagem de referências cíclicas em aplicações reais, os algoritmos podem ser implementados em plataformas reais, como a máquina virtual Java, e testados com aplicações com carga de trabalho significativa, como o conjunto DaCapo [11]. Entretanto, os algoritmos de contagem de referência para sistemas multiprocessados mostrados neste capítulo não seriam adequados para implementação em uma máquina virtual Java, pois esta permite que os *threads* do usuário sejam, cada uma, uma unidade escalonável pelo sistema operacional, podendo executar em processadores separados. Isso exige que o algoritmo de contagem de referências possa operar com mais de um mutador, embora possa permanecer com apenas um coletor. O capítulo seguinte trata de tal extensão e de sua implementação na máquina virtual Java Jikes RVM [1].

capítulo 5

Sistemas com Vários Mutadores e Um Coletor

O Capítulo 4 apresentou algoritmos para realizar o gerenciamento automático da memória em sistemas multiprocessados usando contagem de referências cíclicas. Os algoritmos adequados para um mutador e um coletor (Seções 4.3.1 e 4.3.2) foram implementados em um sistema de tempo de execução para uma linguagem funcional pura criada para servir como plataforma de testes, como detalhado na Seção 4.4. Entretanto, esses algoritmos não seriam adequados para implementação em uma plataforma similar à máquina virtual Java, como discutido anteriormente, pois esta última possibilita que os programas criem vários *threads* que seriam encarados pelo coletor como mutadores.

Para implementação em muitas plataformas atuais, portanto, o interesse maior recai sobre algoritmos que sejam estruturados em vários mutadores e um coletor. Este capítulo trata de versões do algoritmo de contagem de referências cíclicas que são estruturadas desta forma. Inicialmente, é apresentado um possível algoritmo que foi sugerido anteriormente para esse caso, mas que não se mostrou viável na prática. Em seguida, é descrito um algoritmo para vários mutadores e um coletor baseado no conceito de fila de atualizações (Seção 4.3.2); também é descrita uma variante desse último algoritmo que usa duas filas de atualização por mutador. Esses dois algoritmos foram implementados e testados na máquina virtual de pesquisa Jikes RVM [1] (`www.jikesrvm.org`). A implementação e os testes de desempenho realizados com o algoritmo são mostrados. Os resultados demonstram que o algoritmo baseado em fila de atualizações tem bom desempenho, com uma vazão (*throughput*) algumas vezes

ligeiramente menor que outros coletores, mas impondo tempos de pausa muito curtos, o que se reflete em uma latência mais baixa que os coletores alternativos testados. Isso confirma que é vantajoso usar coletores baseados em contagem de referências em situações onde se deseja manter a latência baixa, com um pequeno impacto no desempenho geral do sistema.

5.1 Modelo de *Threads*

A maioria das plataformas de *software* atuais usa o modelo de *threads* de execução como seu modelo primário de concorrência: o sistema operacional Windows usa *threads* desde suas primeiras versões de 32 bits[1]; os sistemas da família Unix que seguem o padrão POSIX 1003.1c; a Máquina Virtual Java [63]; a plataforma .NET da Microsoft [41]. O modelo de *threads* também foi adotado por vários sistemas de tempo de execução de linguagens recentes que permitem a escrita de programas concorrentes.

No modelo de *threads*, o programador pode criar um número arbitrário[2] de *threads* como unidades de execução independentes, mas compartilhando um espaço de memória do processo. Em sistemas com apenas um processador, o sistema de tempo de execução da linguagem de programação pode gerenciar a execução e o escalonamento dos *threads* sem recorrer ao sistema operacional ou máquina virtual subjacente (esse modelo é normalmente chamado de *green threads*[3]); desta forma, interferências entre *threads* mutadores e o coletor podem ser evitadas, e mesmo a existência de mais de um *thread* mutador pode ser ignorada pelo coletor, possibilitando o uso do algoritmo discutido na Seção 4.3.

5.2 Um Possível Algoritmo para Múltiplos Mutadores

Quando o sistema computacional onde o programa será executado possui mais de um processador, entretanto, é mais vantajoso delegar o gerenciamento dos *threads* para o sistema operacional ou máquina virtual subjacente, a fim de aproveitar os processadores disponíveis. Neste caso de concorrência real, é preciso considerar que o programa poderá criar vários *threads* mutadores, e que portanto o algoritmo descrito na Seção 4.3 não pode ser usado. Uma forma

[1] `http://msdn.microsoft.com/en-us/library/ms684254(VS.85).aspx`

[2] De maneira abstrata, não há limite para o número de *threads* que podem ser criadas, mas a maioria dos sistemas atuais impõem limites a esse número por questões práticas de implementação. Além disso, dependendo da plataforma o número de *threads* que podem ser criados sem causar uma grande degradação das características de desempenho do programa são menores que os limites impostos pelo sistema. Entretanto, esses limites podem ser considerados como permissivos de um número muito grande de *threads*, para os propósitos deste trabalho.

[3] `http://en.wikipedia.org/wiki/Green_threads`

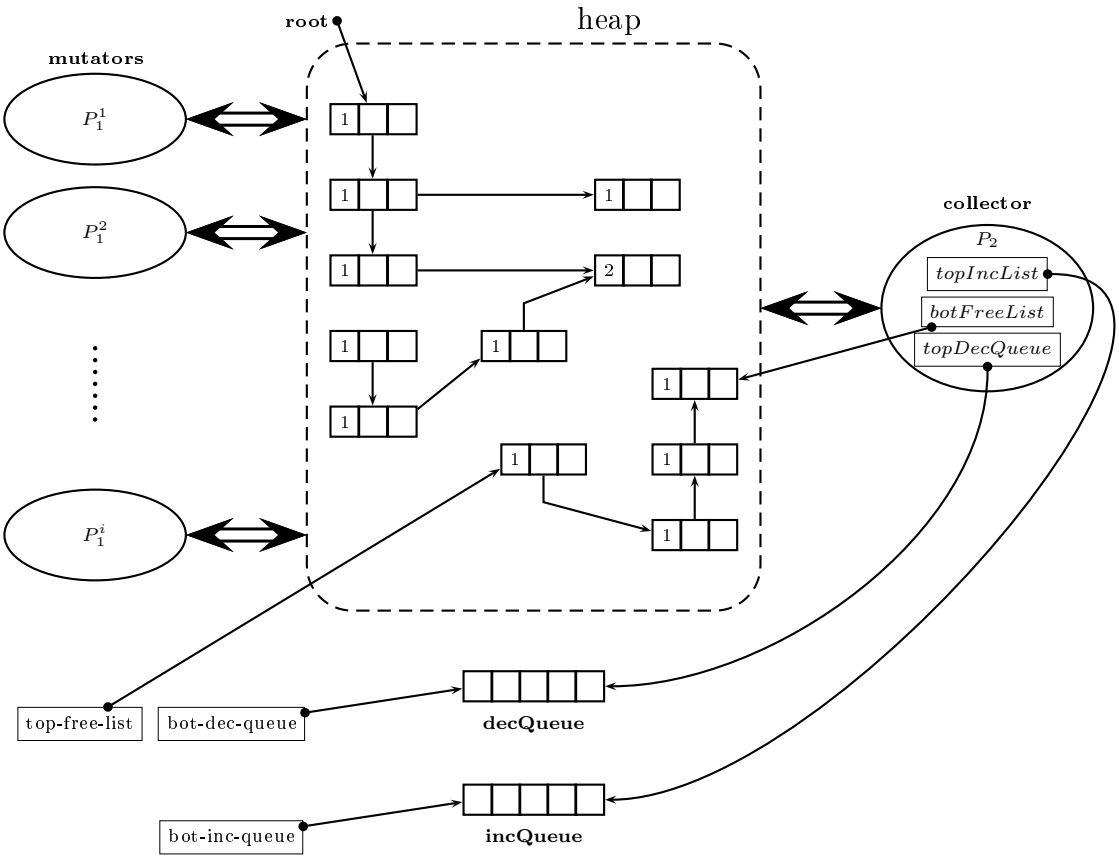

Figura 5.1: *Visão esquemática de uma proposta anterior de algoritmo de contagem de referências cíclicas com um coletor e vários mutadores*

simples de estender esse algoritmo para o uso de vários mutadores, proposta em trabalhos anteriores [39, 68], seria fazer com que os mutadores compartilhem acesso a registradores que guardam referências para o topo da lista livre, e o final das listas de decrementos e incrementos, como mostrado na Figura 5.1. O acesso concorrente aos registradores compartilhados deve ser sincronizado, e as filas de incrementos e decrementos deixam de ter apenas um produtor e um consumidor.

Embora um algoritmo organizado nesta forma (vide Figura 5.1) seja uma solução possível, em teoria, para usar a contagem de referências cíclicas com vários mutadores, no uso de uma implementação concreta o compartilhamento de uma fila entre todos os *threads* mutadores seria um problema. É possível implementar uma fila com sincronização relativamente leve para múltiplos produtores [49], mas com o aumento da contenção pela fila, é inevitável que o desempenho seja prejudicado. Alguns testes de implementação usando essa estratégia, em

moldes similares aos testes relatados na Seção 5.7, demonstraram que um algoritmo seguindo essa estratégia não consegue atingir um desempenho competitivo com outras alternativas.

Uma versão do algoritmo de contagem de referências cíclicas para sistemas multiprocessados com vários mutadores e um coletor que sana esses problemas é descrita a seguir.

5.3 O Algoritmo Proposto para Vários Mutadores e um Coletor

A ideia do algoritmo proposto neste trabalho para usar a contagem de referências cíclicas com vários mutadores e um coletor é separar uma fila para cada mutador, sendo essas filas de atualização, como na Seção 4.3.2, ao invés das filas de incrementos e decrementos. A Figura 5.2 mostra uma representação esquemática dessa ideia.

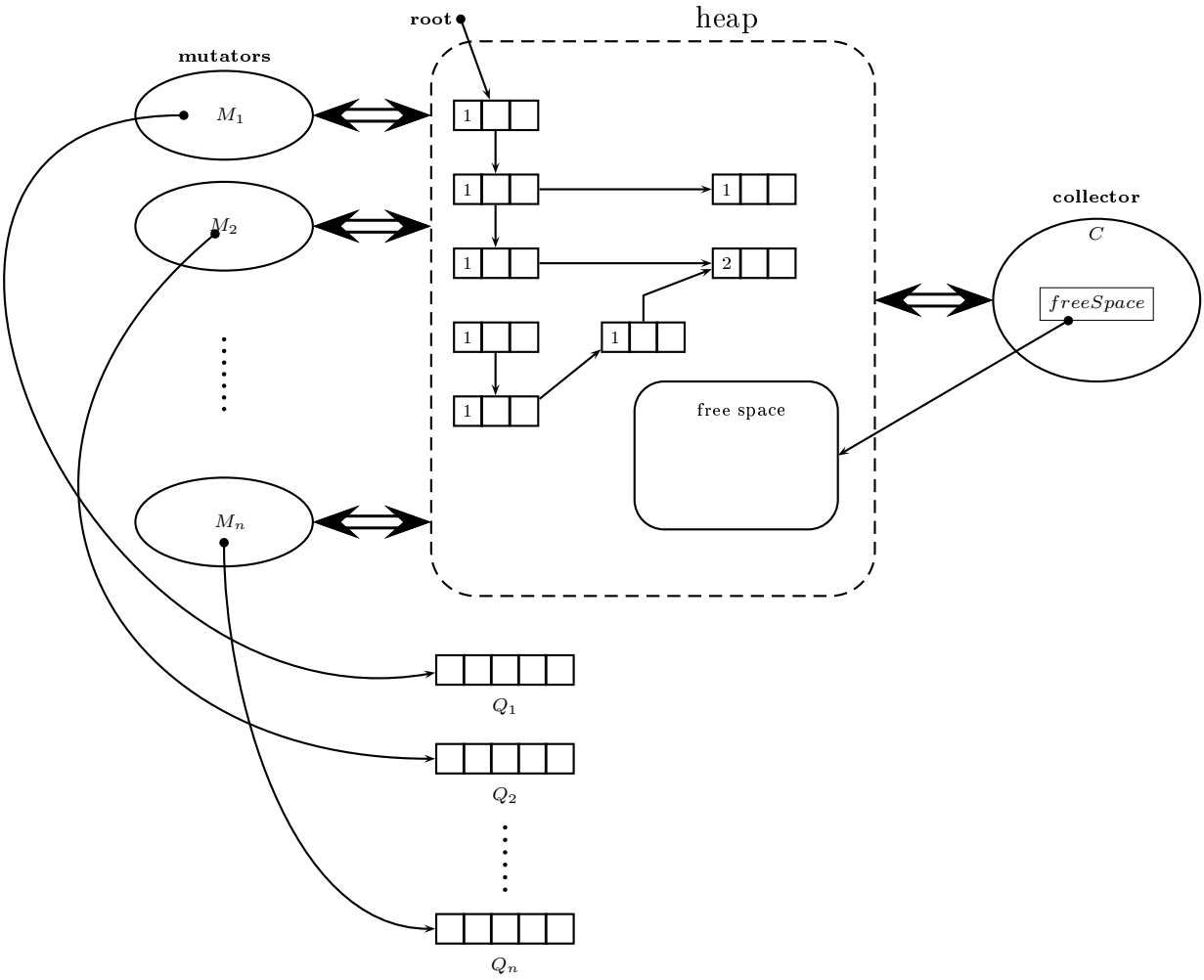

Figura 5.2: *Organização do algoritmo proposto para um coletor e vários mutadores*

As vantagens em usar filas de atualização são similares às vistas na Seção 4.3.2: é possível coordenar os mutadores e o coletor com relativamente pouca sincronização (em relação à versão da Figura 5.1), e reduzindo o número de atualizações dos contadores de referências que precisam efetivamente ser feitas, sem prejuízo na capacidade de detectar e reaproveitar objetos inacessíveis, mesmo que estejam em ciclos. A existência de vários mutadores cria a necessidade de coordenação mais complexa do que no caso de um mutador apenas, mas o algoritmo resultante é *on-the-fly*, tem bom desempenho e impõe apenas pausas pequenas para o funcionamento dos *threads* mutadores, como será visto na Seção 5.7.

O algoritmo proposto para a situação de vários mutadores deve ser adequado para uso em sistemas de tempo de execução e máquinas virtuais de linguagens atuais; especificamente, a implementação do algoritmo para testes foi realizada em uma máquina virtual Java, a Jikes RVM[4]. Portanto, em contraste com as implementações dos algoritmos para um mutador e um coletor, que foram criadas para funcionarem com uma linguagem puramente funcional, o algoritmo da presente seção deve interagir com uma linguagem imperativa. O algoritmo em si muda pouco, mas uma diferença importante em termos de desempenho é ter que diferenciar entre referências na pilha local e referências no *heap*. Referências na pilha estão contidas em objetos locais a algum escopo que ficam armazenados no registro de ativação (*stack frame*) de alguma função ou método, enquanto que referências no *heap* estão contidas em objetos no *heap*. Neste caso, é importante para garantir um bom desempenho não contar com as referências na pilha. Referências na pilha devem ser usadas como raízes para propósitos da coleta de lixo mas, como no algoritmo de Deutsch e Bobrow [28], elas não devem ser utilizadas na contagem de referências.

Como o algoritmo lida com vários mutadores, estes devem ser identificados de alguma maneira. Seguindo a Figura 5.2, a discussão que se segue usa $M_1, M_2, \ldots, M_n$ para representar os mutadores; ou, de maneira equivalente, pelo índice do mutador, de forma que M_i é o mutador i. Em uma implementação, pode-se usar o identificador do *thread* que executa o mutador; praticamente todos os sistemas de *threads* existentes atribuem a cada *thread* um identificador único.

O maior problema na extensão do algoritmo da Seção 4.3.2 para vários mutadores é obter uma visão sincronizada de todas as filas de atualização, sem parar todos os mutadores ao mesmo tempo (caso contrário o coletor seria *stop-the-world* ou, no máximo, concorrente, e

[4] A Jikes RVM era anteriormente conhecida como Jalapeño [1]. Informações sobre a Jikes podem ser encontradas no endereço `http://www.jikesrvm.org/`

os tempos de pausa impostos pelo processo de coleta seriam mais altos). Para resolver isso, usa-se alguns *flags* booleanos na coordenação entre mutadores e coletor. O coletor mantém um *flag* chamado *collect* que indica quando começou um novo ciclo de coleta e as informações das filas estão sendo agregadas. Um mutador, ao ver o *flag collect* com valor verdadeiro durante uma atualização de referência, deve inserir um marcador na sua fila de atualização, para indicar o ponto onde as filas são sincronizadas. Para evitar que mais de um marcador seja inserido, cada mutador i também terá um *flag* $mark(i)$ que será verdadeiro se o marcador já tiver sido inserido na fila Q_i. Depois, o coletor poderá obter os registros de atualização de cada pilha até encontrar o marcador.

Outra situação que deve ser tratada com cuidado na versão com vários mutadores é o estado dos *flags updated*. Esses detalhes serão tratados na descrição do algoritmo, a seguir.

O Mutador As operações relevantes do mutador são as mesmas dos casos anteriores para um mutador e um coletor: a operação NEW do Algoritmo 4.1, e a operação UPDATE. Esta última é bastante similar ao caso do algoritmo com um mutador e um coletor usando fila de atualizações (Algoritmo 4.8), mas com algumas diferenças importantes. Quando UPDATE é chamada pelo mutador de número i, o registro de atualização deve ser inserido na fila de número i. Isso significa que alguma função do sistema deve ser chamada para obter o identificador do *thread*. Além disso, a necessidade de inserir os marcadores na fila, como discutido anteriormente, deve ser observada. A operação UPDATE resultante é mostrada abaixo no Algoritmo 5.1 e utiliza a chamada a GETCURRENTTHREADID para obter o número do mutador. A chamada a ADDUPDATE, então, inclui o número da fila na qual o registro de atualização deve ser inserido. É importante notar que o *flag updated* do objeto atualizado em UPDATE é verificado antes da inclusão do registro de atualização na fila correspondente. Isso é feito para garantir que, em cada ciclo de coleta, só exista um registro de atualização para um determinado objeto. Depois do primeiro registro de atualização inserido para um objeto O, as próximas atualizações não serão inseridas na fila. O importante, para o algoritmo, é que o coletor tenha a informação de quais objetos tiveram referências para ele atualizadas. Ao fim do ciclo de coleta, o *flag updated* de todos os objetos é limpado, possibilitando a inserção de um novo registro de atualização para o próximo ciclo (se o objeto tiver alterações nas referências).

O Coletor O coletor para vários mutadores precisa seguir uma sequência de operação mais complexa que o caso para um mutador. O programa principal do coletor é mostrado no

Algoritmo 5.1 Alteração de referências no algoritmo para n mutadores e um coletor

```
1:  procedure UPDATE(R, S)
2:      if ¬updated(R) then
3:          id := GETCURRENTTHREADID()
4:          if collect ∧ ¬marked(id) then
5:              marked(id) := true
6:              INSERTQUEUEMARK(id)
7:          end if
8:          ADDUPDATE(id, R, S)
9:          updated(R) := true
10:     end if
11:     R := S
12: end procedure
```

Algoritmo 5.2. O primeiro passo é obter as filas de atualização de cada mutador. Para garantir a sincronização do conteúdo lido para as filas, o coletor começa estabelecendo o *flag collect* como verdadeiro. Em seguida, o coletor para cada mutador separadamente e obtem o conteúdo da fila de atualização correspondente até o marcador (se não houver marcador, não ocorreram atualizações após o início do ciclo de coleta, e portanto toda a fila pode ser copiada). CLEARQUEUEUPTOMARK retira itens da fila até achar o marcador (que também é retirado); neste momento o *flag mark* do mutador é estabelecido como verdadeiro para evitar que marcadores sejam inseridos após a fila do mutador ter sido examinada. Em seguida cada mutador pode continuar sua execução. Ao terminar o exame das filas para cada mutador, o coletor desmarca o *flag collect*.

Os dois próximos passos do coletor são limpar os *flags* de marcação de cada mutador e os *flags updated* das referências afetadas. As primeiras não apresentam problema, já que nesse caso o *flag collect* já está desativado e os mutadores não precisam ler nem alterar o valor de $mark(i)$. Entretanto, no caso dos *flags updated*, a situação é mais delicada. Se uma referência r for alterada entre a obtenção dos registros das filas de atualização (*loop* que começa na linha 3) e a chamada a CLEARUPDATEDFLAGS (na linha 11) , o *flag updated*(r) será desativado, mas a referência r já foi atualizada no ciclo atual, e portanto $updated(r)$ deveria continuar verdadeiro. Nesse caso r teve um registro de atualização incluído em alguma das filas $Q_1, \ldots, Q_n$, e isso pode ser usado para reestabelecer os *flags* que forem necessárias. Isso é feito pela operação REFRESHUPDATEDFLAGS.

REFRESHUPDATEDFLAGS precisa examinar a fila de atualizações associada a cada muta-

Algoritmo 5.2 Programa principal do coletor para algoritmo com n mutadores e um coletor

```
1:  procedure CollectorMain
2:      collect := true
3:      for each mutator M_i do
4:          Suspend mutator M_i
5:          CopyQueueUpToMark(i, buffer)          ▷ Copia, eliminando duplicatas
6:          ClearQueueUpToMark(i)
7:          mark(i) := true
8:          Resume mutator M_i
9:      end for
10:     collect := false
11:     ClearMarkFlags()
12:     ClearUpdatedFlags()
13:     RefreshUpdatedFlags()
14:     UpdateReferenceCounters(buffer)
15:     ScanStatusAnalyser()
16:     CollectorMain()
17: end procedure
```

dor, procurando por referências mencionadas em registros de atualização para ativar seus *flags updated*. Para evitar a necessidade de sincronização mais complicada, cada mutador é parado enquanto o coletor copia o conteúdo de sua fila para um *buffer* local chamado *updatedSet*. Em seguida, cada referência presente em algum registro em *updatedSet* tem seu *flag* marcado, recuperando o valor dos *flags* que podem ter sido perdidos quando da chamada a ClearUpdatedFlags. A operação RefreshUpdatedFlags é mostrada no Algoritmo 5.3.

Algoritmo 5.3 Operação para reestabelecer *flags updated* no algoritmo com n mutadores e um coletor

```
1:  procedure RefreshUpdatedFlags
2:      for each mutator M_i do
3:          Suspend mutator M_i
4:          CopyQueueToBuffer(i, updatedSet)
5:          Resume mutator M_i
6:      end for
7:      for each record ⟨r, o_1, o_2⟩ in updatedSet do
8:          updated(r) := true
9:      end for
10: end procedure
```

O resto das operações do coletor continua como antes: UpdateReferenceCounters como no Algoritmo 4.10, e ScanStatusAnalyser como no Algoritmo 3.5, assim como as

operações associadas a ScanStatusAnalyser (ver Capítulo 3) para a identificação e recuperação de ciclos de objetos que se tornaram lixo.

O algoritmo descrito nesta seção foi implementado na máquina virtual Java Jikes RVM e seu desempenho foi testado. Os resultados desses testes são demonstrados na Seção 5.7.

5.4 Um Algoritmo com Duas Filas de Atualização por Mutador

Uma ideia para reduzir o tempo necessário de sincronização entre mutador e coletor é a utilização de duas filas de atualização para cada mutador, ao invés de apenas uma. Desta forma, o mutador pode inserir os registros de atualização em uma das filas, e o coletor pode obter o conteúdo da outra fila durante o ciclo de coleta, sem que seja necessário suspender o mutador para realizar a cópia completa da fila. Essa ideia é representada graficamente na Figura 5.3, na qual as duas filas de cada mutador são denominadas de fila da frente (*front queue*) e fila de fundo (*back queue*). Após o coletor obter o conteúdo da fila de fundo durante um ciclo de coleta, os papeis das duas filas podem ser trocados para iniciar o próximo ciclo.

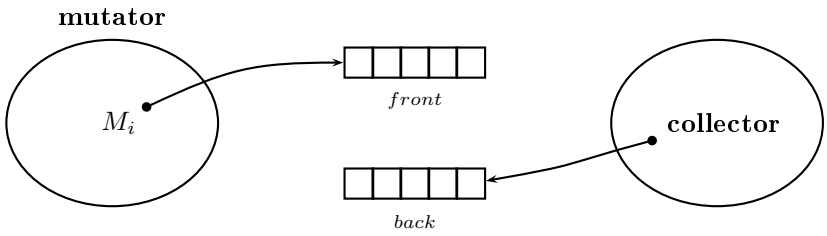

Figura 5.3: *Representação do uso de duas filas de atualização por mutador. Um mutador de número i é mostrado juntamente com o coletor, com as duas filas de atualização entre eles. A cada ciclo de coleta, o papel das duas filas é trocado.*

Entretanto, como o ciclo de coleta pode ocorrer enquanto o mutador executa, ainda é necessário fazer sincronização entre mutador e coletor. Os *flags* são mantidos como na versão anterior, mas há algumas mudanças no uso do *flag mark*. O processo de alteração de referências (Update) para o algoritmo com duas filas de atualização é mostrado no Algoritmo 5.4.

Enquanto não há um ciclo de coleta executando, o mutador deve inserir seus registros de atualização normalmente na fila de frente. Entretanto, se um ciclo de coleta está em andamento, duas situações podem ocorrer:

1. O coletor pode já ter processado o mutador que está executando Update. Nesse caso,

as filas do mutador já foram trocadas e o registro de atualização é inserido na fila da frente.

2. O coletor ainda não processou o mutador que está executando UPDATE. Nesse caso, o ciclo de coleta já começou e o registro de atualização deve ser deixado para o próximo ciclo. Para isso, o mutador insere o registro de atualização na fila de fundo.

Exceto pela mudança na inserção do registro de atualização, o resto da rotina UPDATE permanece como no algoritmo anterior.

Algoritmo 5.4 Alteração de referências para algoritmo com duas filas por mutador.

```
1:  procedure Update(R, S)
2:      if ¬updated(R) then
3:          id := GetCurrentThreadID()
4:          if collect ∧ ¬marked(id) then
5:              AddUpdateToBackQueue(id, R, S)
6:          else
7:              AddUpdateToFrontQueue(id, R, S)
8:          end if
9:          updated(R) := true
10:     end if
11:     R := S
12: end procedure
```

O programa principal do coletor para o algoritmo com duas filas de atualização por mutador é mostrado no Algoritmo 5.5. A maior parte do algoritmo é similar à versão anterior (Algoritmo 5.2), mas o processamento de cada mutador é alterado. Como o coletor executa concorrentemente com os mutadores, fazer a troca das filas do mutador enquanto uma operação UPDATE do mutador em questão poderia criar condições de corrida ou exigir o uso de sincronização baseada em *locks*, o que vai contra os objetivos do algoritmo proposto. Mesmo com apenas um mutador, verificou-se pelos resultados dos experimentos relatados no Capítulo 4 que é mais eficiente evitar sincronização baseada em *locks*. Desta forma, o coletor suspende o coletor para realizar a troca das filas; enquanto o mutador está parado, o *flag* *mark* também é ativado para sinalizar ao mutador que suas filas foram trocadas. Após isso, o mutador pode retomar sua execução e o coletor copia a fila de fundo para seu *buffer*. O resto do algoritmo do coletor, e as outras rotinas, são iguais às versões do algoritmo com fila de atualizações da Seção 5.3.

Esse algoritmo com duas filas de atualização por mutador foi implementado na máquina

Algoritmo 5.5 Algoritmo do coletor para algoritmo com n mutadores e um coletor e duas filas por mutador.

```
1:  procedure CollectorMain
2:      collect := true
3:      for each mutator M_i do
4:          Suspend mutator M_i
5:          Flip back and front queues
6:          mark(i) := true
7:          Resume mutator M_i
8:          CopyBackQueue(i, buffer)              ▷ Copia, eliminando duplicatas
9:          ClearBackQueue(i)
10:     end for
11:     collect := false
12:     ClearMarkFlags()
13:     ClearUpdatedFlags()
14:     RefreshUpdatedFlags()
15:     UpdateReferenceCounters(buffer)
16:     ScanStatusAnalyser()
17:     CollectorMain()
18: end procedure
```

virtual Jikes RVM e seu desempenho foi testado em relação à versão com apenas uma fila por mutador. Os resultados dos testes de desempenho são mostrados na Seção 5.7.6.

5.5 Extensão para Uso de Vários Coletores

Em arquiteturas atuais, nas quais a tendência é que existam poucos processadores disponíveis, provavelmente faz mais sentido usar apenas um coletor, como na Seção 5.3. Para o futuro, se o número de processadores ou núcleos de processamento continuar aumentando, é de se esperar que usar apenas um coletor não será a possibilidade mais eficiente quando o número de mutadores executando concorrentemente em um dado instante de tempo é muito grande. Neste caso, pode compensar ter um algoritmo com vários coletores.

Existem algumas possibilidades que podem ser estudadas para extender o algoritmo de contagem de referências cíclicas para vários coletores, tornando o conjunto um coletor paralelo e *on-the-fly*. Uma delas é dividir áreas do *heap* entre os coletores, ou fazer com que cada um tenha um *status analyser* local, para que várias varreduras locais possam ser executadas em paralelo; cada coletor, ao identificar uma célula candidata para o *status analyser*, adiciona-a à sua versão local da estrutura.

O gerenciamento da área de memória livre também pode ser paralelizada. Uma ideia de segmentação da área livre, assim como do *heap* inteiro, baseada em endereços pode ser utilizada para evitar ou minimizar a necessidade de comunicação entre os coletores. Caso existam desequilíbrios de carga, pode-se usar técnicas de *work stealing* para melhorar o balanceamento [37].

Outra possibilidade seria ter um coletor – com todas as estruturas de dados associadas – para cada N mutadores, sendo o valor desse N determinado experimentalmente. A relação entre mutadores e coletores poderia ser mais um parâmetro ajustável pelo programador, pois pode depender de características específicas do programa. Neste caso também podem ocorrer desbalanceamento na carga de cada coletor, e mais uma vez pode-se usar técnicas conhecidas para o balanceamento.

Entretanto, o estudo dessa modalidade não será detalhado aqui, ficando como possibilidade para trabalhos futuros.

5.6 Implementação e Plataforma de Testes

O algoritmo proposto de contagem de referências cíclicas para vários mutadores e um coletor, baseado no conceito de filas de atualização, foi apresentado na Seção 5.3. Este algoritmo foi implementado e testado na máquina virtual de pesquisa Jikes RVM [1] (`www.jikesrvm.org`).

Esta seção descreve a implementação do algoritmo da Seção 5.3 na Jikes RVM e relata o resultado dos testes realizados para medir o desempenho de tal algoritmo em comparação com outros coletores existentes na máquina virtual Jikes. Os testes foram realizados com os programas do conjunto de *benchmarks* DaCapo 9.12-bach [11, 12], um conjunto de *benchmarks* criado por pesquisadores (com o apoio de grandes empresas) para medir o desempenho de sistemas Java[5].

5.6.1 Jikes RVM

A linguagem Java [5] foi projetada para que programas escritos na linguagem sejam executados em uma máquina virtual com uma arquitetura específica, chamada de máquina virtual

[5] Das publicações listadas no site da Jikes RVM (`http://www.jikesrvm.org/Publications`), que usam a máquina virtual Jikes para propor melhorias na tecnologia de máquinas virtuais Java, a maioria das que foram publicadas nos últimos dois anos usou o alguma versão do conjunto DaCapo como base para medições de desempenho. O conjunto de *benchmarks* DaCapo pode ser encontrado no endereço `http://www.dacapobench.org/`

Java[6] [63] e conhecida pela sigla JVM, do inglês *Java Virtual Machine*. A máquina virtual Java foi projetada para utilizar o gerenciamento automático da memória, incluindo um coletor de lixo desde suas primeiras versões.

Embora o conceito de máquinas virtuais já existisse há algum tempo, a popularidade da linguagem Java ajudou a aumentar o interesse por que esse tipo de arquitetura de *software*. Da mesma forma, o uso de um coletor de lixo para gerenciar automaticamente a memória dos programas se tornou mais difundido graças à popularidade de Java. Naturalmente, esses fatores contribuíram para um maior interesse de pesquisas, tanto na academia quanto na indústria de *software*, que tivessem máquinas virtuais e seus subsistemas – principalmente os coletores de lixo – como objeto de estudo.

Seguindo a especificação para a JVM [63], várias implementações da máquina virtual Java surgiram, algumas comerciais, algumas criadas como projetos de pesquisa. Entretanto, quase todas tinham como objetivo mais importante maximizar o desempenho na execução dos programas, e não a facilidade de compreensão ou modificação da implementação, o que dificultava o seu uso em pesquisas. Para facilitar a pesquisa sobre a tecnologia de máquinas virtuais e coleta de lixo, a IBM iniciou em 1997 um projeto interno chamado Jalapeño, cujo objetivo era desenvolver uma máquina virtual Java, obedecendo à especificação da JVM, mas tendo como objetivo primário servir de plataforma de pesquisa. A máquina virtual Jalapeño foi o resultado desse projeto. No final de 2001, a IBM decidiu disponibilizar a tecnologia desenvolvida pelo projeto Jalapeño para uma comunidade mais ampla de pesquisadores, alterando o nome da máquina virtual para Jikes Research Virtual Machine (Jikes RVM), e tornando o código-fonte disponível sob uma licença de *software* livre[7].

A máquina virtual Jikes é implementada quase totalmente na própria linguagem Java, em contraste com a maioria das outras JVMs, normalmente implementadas em C ou C++. Isso facilita a sua portabilidade e simplifica o modelo da memória para a programação de novas estratégias de gerenciamento automático da memória, incluindo o coletor de lixo. Os módulos da máquina virtual Jikes relacionados ao gerenciamento da memória foram separados em um subsistema chamado de MMTk (*Memory Managemement Toolkit*), para facilitar a criação

[6] Tanto a linguagem de programação quanto a plataforma de execução e máquina virtual têm o mesmo nome: Java. Embora a linguagem e a plataforma tenham sido criadas para trabalharem sempre em conjunto, hoje em dia é cada vez mais comum ver sistemas que executam sobre a plataforma Java, mas que são escritos em linguagens de programação diferentes de Java (por exemplo Scala e Clojure).

[7] Algumas informações históricas sobre o projeto Jalapeño e a Jikes RVM podem ser encontradas no FAQ do *site* oficial do projeto: `http://www.jikesrvm.org/FAQ`

de novos coletores de lixo ou mesmo a adaptação de todo *toolkit* para outras plataformas de *software*. O algoritmo de contagem de referências cíclicas usando vários mutadores e um coletor foi implementado na Jikes RVM usando o MMTk, como descrito a seguir. A versão da Jikes RVM utilizada neste trabalho foi a 3.1.1, lançada em Julho de 2010[8].

5.6.2 MMTk

MMTk (*Memory Management Toolkit*) é o componente da Jikes RVM voltado ao desenvolvimento e teste de sistemas de gerenciamento automático de memória dinâmica, especialmente para coletores de lixo[9] [13].

Como está implementado em Java, o MMTk está organizado em uma hierarquia de classes projetada para acomodar diferentes estratégias de gerenciamento da memória. O grande número de classes envolvidas no funcionamento do MMTk – o componente corresponde a cerca de um terço do código completo da Jikes RVM – reflete sua flexibilidade e complexidade de funcionamento.

Para implementar novas estratégias de gerenciamento de memória, é necessário criar subclasses de algumas classes-base importantes no sistema MMTk. Por exemplo, um conceito importante é o de um *plano*, que é uma visão de alto nível de como a memória será planejada; a classe PLAN, do pacote `org.mmtk.plan`, representa um plano de gerenciamento da memória. Para criar uma nova estratégia é preciso criar uma subclasse de PLAN. A implementação de um plano de gerenciamento não envolve apenas uma subclasse de PLAN, sendo também necessário implementar subclasses de outras classes associadas, como será detalhado posteriormente.

As classes do MMTk estão divididas em quatro pacotes principais:

- **org.mmtk.plan** – Classes relacionadas ao plano de gerenciamento da memória.
- **org.mmtk.policy** – Uma classe POLICY determina a política de gerenciamento do espaço de memória no *heap*, incluindo a política de alocação e o algoritmo de coleta de lixo. A classe-base principal deste pacote é a classe SPACE, que representa um espaço de memória, ou seja, uma região do *heap*.
- **org.mmtk.utility** – Agrupa classes utilitárias usadas pelas outras partes do sistema MMTk. Estruturas de dados usadas no gerenciamento da memória, como listas de

[8] `http://docs.codehaus.org/display/RVM/2010/07/04/Jikes+RVM+3.1.1+Released`

[9] Alguns aspectos do MMTk, como os *sistemas de alocação* de memória no *heap*, não estão diretamente relacionados à coleta de lixo.

objetos livres, listas encadeadas e filas sincronizadas, são definidas neste pacote.

- ▷ **org.mmtk.vm** – Classes que operam na interface entre o MMTk e a máquina virtual a qual o sistema MMTk está associado. Na maioria dos casos essa máquina virtual é uma plataforma Java, mas o MMTk pode ser usado com outras máquinas virtuais.

O plano de gerenciamento é o que coordena todos os outros componentes em conjunto, reunindo-os em uma estratégia de gerenciamento da memória. É possível inclusive dividir o *heap* em diferentes espaços, que podem usar diferentes algoritmos de alocação e coleta de lixo. O plano também é responsável por guardar os dados e estatísticas do sistema de gerenciamento da memória durante a execução, principalmente para análise do desempenho e diagnóstico.

Na Jikes RVM, a escolha do plano de gerenciamento da memória que deve ser usado é refletida em diferentes *configurações* da máquina virtual. É necessário, ao compilar uma versão da Jikes RVM, especificar qual configuração deverá ser usada. Por exemplo, a configuração `BaseBaseMarkSweep`, que seleciona um coletor de lixo *stop-the-world* de marcação e varredura (*mark-sweep*), determina o uso do plano de gerenciamento `org.mmtk.plan.marksweep.MS`; esta classe é uma subclasse de `org.mmtk.plan.StopTheWorld`, e inclui o uso de espaços do tipo `org.mmtk.policy.MarkSweepSpace` e de várias classes nos pacotes `org.mmtk.utility` e `org.mmtk.vm`.

5.6.3 Implementação na Jikes RVM

Como visto anteriormente, definir um novo plano de gerenciamento da memória na Jikes RVM, usando o MMTk, envolve a criação de um conjunto de classes que representam esse plano. As duas partes principais desse conjunto são o plano em si – subclasses de Plan e classes associadas – e uma subclasse de Space, que determina a organização do espaço do *heap* e as políticas de alocação e coleta de lixo.

Os planos presentes na distribuição de código-fonte da Jikes RVM, versão 3.1.1, incluem apenas estratégias *stop-the-world* sequenciais ou paralelas de gerenciamento da memória. Portanto, para implementar o algoritmo *on-the-fly* de contagem de referências apresentado na Seção 5.3 foi necessário criar uma infra-estrutura ainda não existente (ou, ao menos, não publicamente disponível) na Jikes RVM. Isso incluíu a criação de um pacote `org.mmtk.plan.onthefly`, com classes relacionadas à uma estratégia geral de gerenciamento *on-the-fly*, e um pacote `org.mmtk.plan.otfrc` para classes específicas para o algoritmo de contagem de

referências *on-the-fly* com vários mutadores e um coletor, como apresentado na Seção 5.3. Também foram criadas classes que implementam um espaço de memória adequado para o algoritmo de contagem de referências, incluindo um algoritmo de alocação baseado em lista de objetos livres e especificando a política de coleta de lixo do algoritmo de contagem de referências. Uma implementação de fila FIFO com sincronização eficiente para um produtor e um consumidor foi incluída no pacote `org.mmtk.utility`.

Além das classes adicionadas, algumas classes-base do MMTk tiveram de ser alteradas para incluir referências às novas classes. Por exemplo, a classe PLAN do pacote `org.mmtk.plan` guarda referências aos planos disponíveis; neste caso, a classe foi alterada para incluir referências aos novos planos no pacote `org.mmtk.plan.otfrc`. O Apêndice C inclui mais detalhes sobre a implementação do algoritmo na Jikes RVM.

5.7 Testes e Resultados

Para analisar o desempenho do algoritmo da Seção 5.3, conforme implementado na Jikes RVM (Seção 5.6), foi usado o conjunto de *benchmarks* DaCapo, versão 9.12-bach [11, 12], em uma série de experimentos. Nesses experimentos, o algoritmo da Seção 5.3 foi comparado com dois coletores paralelos implementados na Jikes RVM.

5.7.1 O conjunto de *benchmarks* DaCapo

O conjunto de *benchmarks* DaCapo contém uma coleção de programas criados para avaliação de desempenho (*benchmarks*). Esses programas de teste são baseados, em sua maioria, em programas reais, de vários níveis de tamanho e complexidade, usados comumente pela comunidade Java. Para as medições de desempenho relatadas aqui, todos os programas de teste do conjunto foram usados, conforme descrição abaixo.

avrora Simula vários programas executando em um *grid* de microcontroladores AVR;

batik Produz um conjunto de imagens SVG (*Scalable Vector Graphics*) baseadas nos testes de unidade do Apache Batik;

eclipse Executa os testes de desempenho da JDT (*Java Development Tools*), um componente da IDE Eclipse;

fop Realiza análise sintática de um arquivo XML-FO e formata uma saída em PDF baseado neste arquivo;

h2 Executa um teste de desempenho baseado nos testes do JDBCbench;

jython Usa o interpretador Jython (implementa Python na máquina virtual Java) para executar o *benchmark* pybench;

luindex Usa o Lucene para indexar um conjunto de documentos que inclui as obras completas de Shakespeare e a Bíblia do Rei James (*King James Bible*);

lusearch Usa o Lucene para fazer uma busca textual em um conjunto de documentos que inclui as obras completas de Shakespeare e a Bíblia do Rei James (*King James Bible*);

pmd Realiza análise estática de um conjunto de classes Java;

sunflow Renderiza um conjunto de imagens usando *ray-tracing*;

tomcat Executa o servidor de aplicações Web Tomcat, enviando para o servidor um conjunto de *requests*, e verificando as páginas web retornadas;

tradebeans Executa o *benchmark daytrader* usando JavaBeans;

tradesoap Executa o *benchmark daytrader* usando SOAP;

xalan Transforma um conjunto de documentos XML em HTML usando folhas de estilo;

A Tabela 5.1 mostra dados relativos à memória consumida por cada programa de teste durante sua execução, usando um coletor de cópia sequencial como referência. As três colunas à esquerda indicam a memória utilizada em *megabytes*: quantidade total de memória alocada durante toda a execução, quantidade de memória máxima efetivamente usada ao mesmo tempo, e a razão entre essas duas últimas quantidades. As três colunas seguintes à direita apresentam a mesma informação, mas em número de objetos. A última coluna à direita mostra o tamanho médio dos objetos alocados, em *kilobytes* (calculado a partir de uma média aritmética entre o tamanho médio dos objetos alocados e o tamanho médio dos objetos utilizados). Essas informações ajudam a caracterizar a carga de trabalho imposta pelos programas de teste. Por exemplo, é possível ver que o programa xalan cria e descarta um grande número de objetos – para representação das estruturas de arquivos XML e HTML – nunca mantendo um grande número deles na memória ao mesmo tempo.

5.7.2 Arranjo Experimental

Os coletores testados na Jikes RVM para comparação foram:

- **pargencopy**: coletor de cópia paralelo generacional, com duas gerações

Tabela 5.1: *Características de uso da memória para os programas de teste*

	MBs			Objetos			
Teste	alocados	útil	aloc/útil	alocados	utilizados	aloc/útil	tam. médio (kb)
avrora	241.7	2.7	89.52	4288113	15977	268.4	59.1
batik	1310.2	52.5	24.96	33597434	149735	224.4	40.8
eclipse	5580.0	36.6	152.46	104355149	471433	221.3	56.0
fop	100.2	7.7	13.01	2433671	178392	13.6	43.2
h2	143.5	75.4	1.90	4576965	3273279	1.39	32.9
jython	1198.3	0.2	5990.0	25976592	2887	8997.8	48.4
luindex	201.9	1.1	183.55	7293149	19477	374.4	29.0
lusearch	1793.3	10.9	164.52	15845173	35092	451.5	118.7
pmd	781.8	13.7	57.07	34176618	420175	81.3	23.9
sunflow	911.6	63.8	14.29	33716349	899773	37.4	28.4
tomcat	934.7	78.0	11.98	31198735	904473	34.5	31.4
tradebeans	17365.9	211.7	82.03	38734190	433910	89.3	470.1
tradesoap	21327.4	274.1	77.81	92617730	1171940	79.0	241.6
xalan	60254.6	26.8	2248.31	161078103	170117	946.9	389.3

▷ **pargenms**: coletor de marcação e varredura paralelo generacional, com duas gerações

Nas tabelas, o coletor baseado no algoritmo da Seção 5.3 é referenciado com o nome **rc-conc-n1**. Os coletores acima listados (**pargencopy** e **pargenms**) estão entre os de melhor desempenho dentre os coletores implementados na Jikes RVM, versão 3.1.1. Ambos os coletores utilizam todos os processadores disponíveis para executar, sendo portanto mais similares para comparação do que um coletor *stop-the-world*.

No total, foram realizados quatro experimentos com os algoritmo proposto na Jikes RVM, sendo três experimentos projetados para determinar o desempenho do coletor proposto na Seção 5.3 em relação aos coletores paralelos distribuídos com a Jikes RVM, e um experimento para comparar o desempenho das variantes do algoritmo proposto com uma e duas filas de atualização:

1. O primeiro consistiu em executar todos os programas do conjunto DaCapo em uma máquina virtual Jikes RVM com os mesmos parâmetros de memória, tendo o *heap* de tamanho fixo em 300 Mb.

2. Para o segundo experimento, os programas do conjunto DaCapo foram divididos em classes de acordo com o perfil de consumo de memória de cada um (como pode ser visto na Tabela 5.1), e os grupos formados por essas classes foram executados em máquinas

virtuais com configurações de memória diferentes: três classes de programas foram identificadas, por ordem crescente da utilização do *heap*, e os três grupos correspondentes foram executados em máquinas virtuais com tamanho de *heap* fixo em 60Mb, 100Mb e 300Mb.

3. O terceiro experimento consistiu em testar as características dos algoritmos à medida em que o número de processadores usados cresce. Para isso, os programas foram todos executados em uma máquina virtual com mesma configuração de memória, mas cada programa foi testado executando em um, dois e quatro processadores.

4. O quarto experimento teve como objetivo comparar o desempenho das duas variantes do algoritmo proposto: a versão com uma fila de atualizações por mutador (Seção 5.3) e a versão com duas filas de atualizações por mutador (Seção 5.4).

Em todos os experimentos, o tamanho da entrada para cada *benchmark* do conjunto DaCapo foi mantido no tamanho padrão (*default*), como recomendado para testes de desempenho [11]. Todas os programas testados foram executado 15 vezes, e os resultados mostrados são a média dos parâmetros medidos.

O *hardware* utilizado nos dois primeiros experimentos foi um computador com processador Intel Core 2 Duo de 2.1GHz com 2Gb de RAM e sistema operacional Ubuntu Linux versão 10.04. Para os experimentos 3 e 4, foi utilizado um computador com processador Intel Core i5-750 de 2.66GHz com 4Gb de RAM e sistema operacional Ubuntu Linux versão 10.10.

Os programas de teste foram executados na máquina virtual Jikes (executável `rvm`) utilizando os seguintes parâmetros de linha de comando:

- ▷ `-X:gc:variableSizeHeap=false` Estabelece que o *heap* terá tamanho fixo durante toda a execução do programa;
- ▷ `-Xms<tam>M` Estabelece o tamanho do *heap*, onde `tam` é o tamanho em *megabytes*;
- ▷ `-X:gc:xmlStats=true` Determina que os números e figuras de desempenho do coletor de lixo serão reportadas em formato XML;
- ▷ `-X:processors=<n>` Configura o coletor de lixo para usar `n` processadores;
- ▷ `-jar dacapo-9.12-bach.jar` Determina que o programa a executar está no arquivo JAR do conjunto DaCapo, versão 9.12;
- ▷ `<teste>` Nome do programa de teste do conjunto DaCapo;

- ▷ `-c MMTkCallback` Determina que os programas do conjunto DaCapo chamem um método do MMTk durante a execução, para que seja possível obter figuras de desempenho detalhadas do coletor de lixo.

Assim, como exemplo, para executar o teste *jython* com um *heap* de 60Mb e com o coletor de lixo utilizando dois processadores, a linha de comando utilizada foi:

```
./rvm -X:gc:variableSizeHeap=false -Xms60M -X:gc:xmlStats=true -X:processors=2 \
      -jar dacapo-9.12-bach.jar jython -c MMTkCallback
```

Os quatro experimentos e seus resultados são apresentados nas seções a seguir.

5.7.3 Resultados: Primeiro Experimento

O primeiro experimento consistiu em executar todos os programas do conjunto DaCapo em uma máquina virtual Jikes configurada com os mesmos parâmetros de memória: *heap* de tamanho fixo e igual a 300Mb, tamanho suficiente para execução de todos os programas do conjunto. Dessa forma, é possível obter um panorama geral do desempenho de execução dos programas usando os três coletores de lixo diferentes. O objetivo foi medir dois parâmetros de desempenho: vazão (*throughput*) e latência, o primeiro através do tempo total de execução dos programas, e o segundo através do tempo máximo de pausa que o coletor impõe aos mutadores. Os testes foram executados em um computador com processador Intel Core 2 Duo de 2.1GHz, que contém dois núcleos de processamento.

A Tabela 5.2 mostra o resultado do tempo de usuário total medido na execução dos programas de teste do conjunto DaCapo para os três coletores comparados. Em todos os casos, é possível ver que o algoritmo de contagem de referências cíclicas tem desempenho menos de 10% pior que os coletores alternativos, em alguns casos até apresentando desempenho melhor que os outros (por ex. *xalan* e *pmd*). Os resultados da Tabela 5.2 são mostrados graficamente na Figura 5.4, onde pode ser visto que a diferença relativa no tempo entre os três tipos de coletores é pequena.

Os resultados para a latência, medida em termos do tempo máximo de pausa imposto pelo sistema coletor aos mutadores, são mostrados na Tabela 5.3. Pode-se ver facilmente que os tempos de pausa impostos pelo algoritmo de contagem de referências são muito menores em relação aos coletores comparados. Enquanto o coletor por contagem de referências pausa por apenas 50-80 ms, no máximo, os coletores paralelos de rastreamento chegam a pausar algum mutador por vários segundos.

Tabela 5.2: *Tempos totais de execução para os programas de teste, em segundos.* heap *de 300Mb.*

Teste	*pargencopy*	*pargenms*	*rc-conc-n1*
avrora	54.630	53.351	54.250
batik	52.310	52.511	53.318
eclipse	502.271	500.812	503.732
fop	21.521	22.872	23.702
h2	77.234	75.473	77.298
jython	121.133	122.274	128.208
luindex	32.731	33.146	33.764
lusearch	106.803	110.904	108.109
pmd	69.234	67.473	66.029
sunflow	71.302	73.071	76.098
tomcat	79.176	78.047	77.131
tradebeans	99.182	100.040	100.981
tradesoap	278.114	287.332	289.954
xalan	66.941	67.413	65.031

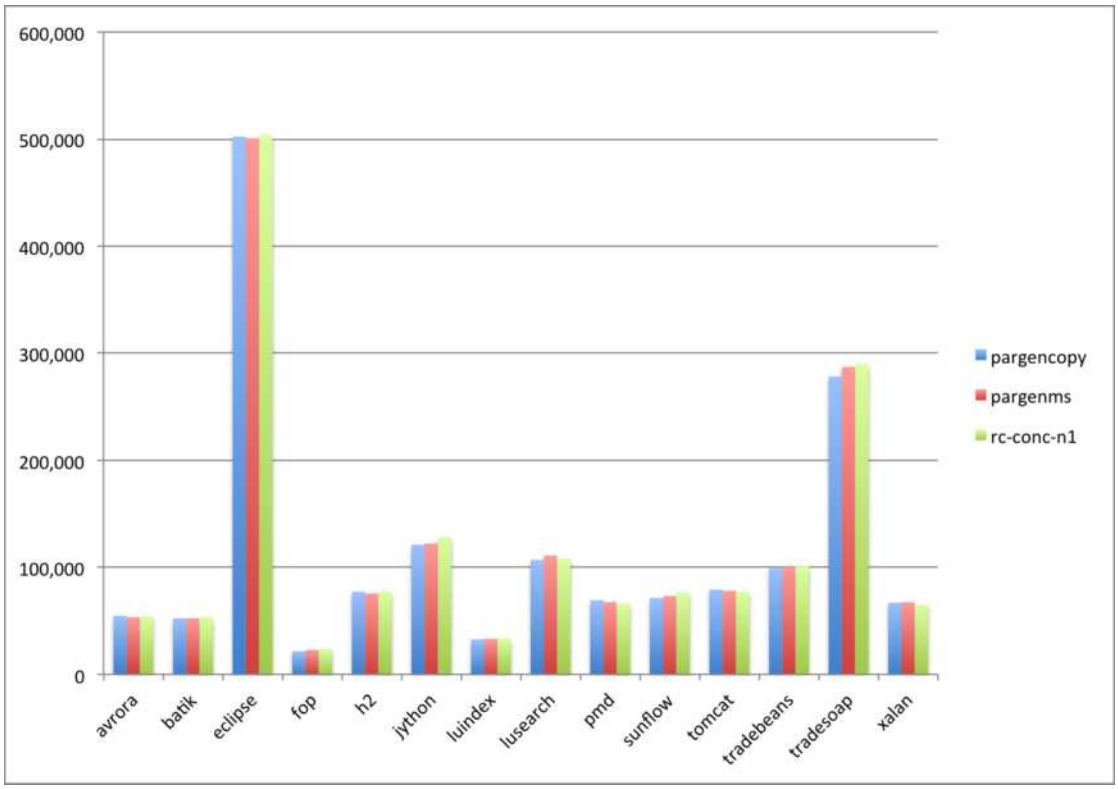

Figura 5.4: *Tempos de execução (em segundos) para os* benchmarks *DaCapo nos três coletores considerados.*

Tabela 5.3: *Tempo máximo de pausa dos mutadores, em milisegundos.* Heap *de 300Mb.*

Teste	*pargencopy*	*pargenms*	*rc-conc-n1*
avrora	3130	3162	40
batik	3501	3501	33
eclipse	7343	6812	67
fop	3102	3218	30
h2	6284	6319	55
jython	8226	8266	88
luindex	3782	3726	44
lusearch	5017	5092	52
pmd	6645	6599	69
sunflow	3212	3189	31
tomcat	6577	6502	58
tradebeans	7002	6984	66
tradesoap	7219	7192	70
xalan	5166	5201	48

Para medir as tendências da vazão e latência à medida em que o espaço reservado para o *heap* aumenta, o primeiro experimento foi repetido para máquinas virtuais configuradas com outros tamanhos de *heap*: 600Mb e 1200Mb. Como a quantidade de memória disponível tende a continuar crescendo, é importante ter uma ideia de como o desempenho dos coletores é afetado com o aumento do tamanho do *heap*. Para o tempo total de execução dos testes, os resultados coletados continuam mostrando pouca diferença relativa entre os três coletores testados, embora o desempenho de todos tenha sido pior (por um fator pequeno), provavelmente devido à perda de localidade. Entretanto, com relação ao tempo máximo de pausa, é possível ver uma tendência de aumento da vantagem do coletor baseado na contagem de referências em relação aos coletores paralelos. A Tabela 5.4 mostra um sumário dos resultados, apresentando uma média dos tempos máximos de pausa obtidos em todos os programas para um dado coletor e um tamanho de *heap*. Para um *heap* de 300Mb, o valor mostrado na Tabela 5.4 é uma média dos valores individuais mostrados na Tabela 5.3 para cada teste. Observa-se que o coletor baseado em contagem de referências impõe tempos de pausas cerca de duas ordens de magnitude menores do que os coletores paralelos, em todos os casos. Para visualizar a tendência de variação do tempo de pausa à medida em que o tamanho do *heap* cresce, são mostradas na Figura 5.5 curvas baseadas nos dados da Tabela 5.4. Para que fosse possível visualizar as três curvas em uma escala similar – já que o objetivo é apenas ver a tendência

Tabela 5.4: *Tempo máximo de pausa dos mutadores (média de todos os programas de teste), em milisegundos, para três tamanhos de* heap.

Tamanho do *heap* (MB)	*pargencopy*	*pargenms*	*rc-conc-n1*
300	5443	5411	53
600	6859	7145	58
1200	9628	10837	74

de variação, e não os valores específicos – os tempos de pausa dos coletores paralelos foram divididos por 100. Na Figura 5.5, vê-se que o tempo de pausa imposto pelo coletor baseado em contagem de referências aparenta crescer mais lentamente com o aumento do *heap* em relação aos coletores paralelos.

Isso permite concluir que o algoritmo de contagem de referências mostrado na Seção 5.3 é competitivo com as alternativas em termos de desempenho, mas atinge uma latência muito menor, e a diferença de latência tende a se acentuar com o crescimento do *heap*. Em situações onde é desejável limitar o tempo de pausa imposto pelo coletor ao programa sendo executado – por exemplo, sistemas de tempo real, sistemas interativos para usuários finais ou servidores que devem limitar o tempo de resposta por transação – o uso de coletores de referências cíclicas pode atingir esse objetivo sem penalidade significativa no desempenho. O bom comportamento da latência com o crescimento do *heap*, observado no coletor por contagem de referências, indica que essa técnica se torna mais vantajosa à medida que a quantidade de memória disponível para o sistema cresce – por exemplo, como acontece com servidores.

5.7.4 Resultados: Segundo Experimento

O primeiro experimento foi executado com uma máquina virtual Jikes configurada com um tamanho de *heap* fixo e único para todos os programas do conjunto DaCapo. Embora isso seja adequado para obter um panorama geral do desempenho relativo dos três coletores testados, os programas do conjunto DaCapo apresentam diferentes perfis de carga com relação ao uso da memória, como pode ser visto na Tabela 5.1. No caso de alguns programas do conjunto, um *heap* de 300Mb pode ser grande demais, fazendo com que sua execução exija pouco do coletor.

Para obter informações sobre o comportamento dos programas de teste sujeitos a condições de maior uso do coletor, os programas do conjunto DaCapo foram divididos em três classes de acordo com o uso da memória. Os programas de cada classe foram executados em máquinas

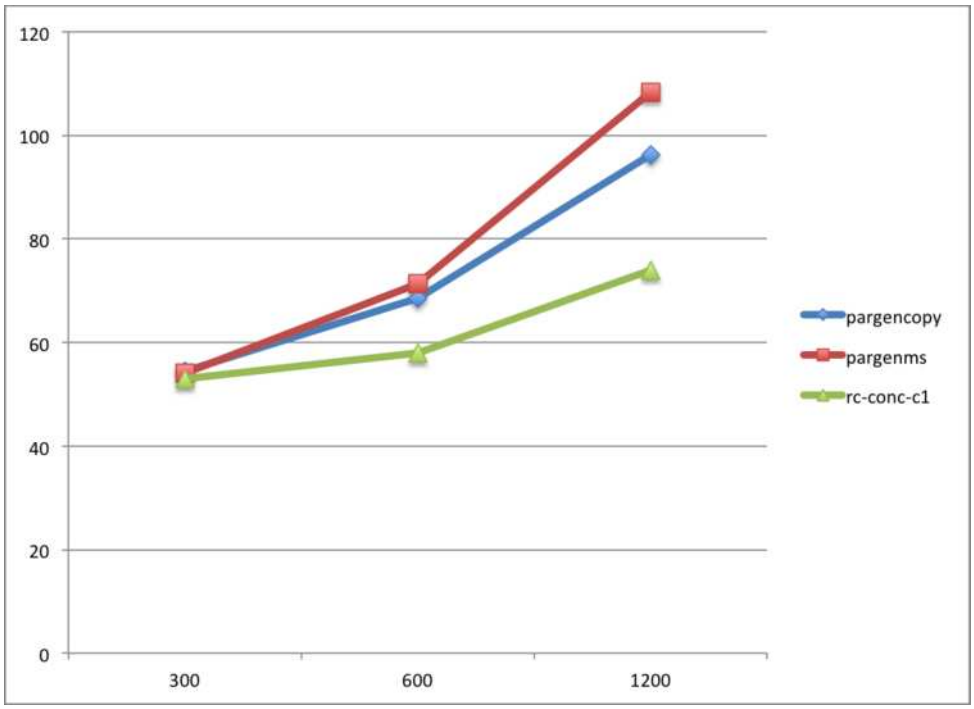

Figura 5.5: *Tendência do tempo de pausa (eixo Y) em relação ao tamanho do* heap *(eixo X, em Mb). Os tempos no eixo Y foram alterados para compatibilizar a escala.*

virtuais com diferentes tamanhos de *heap*, sendo então configurados três tamanhos diferentes: 60Mb, 100Mb e 300Mb. Na Tabela 5.5 são mostradas as características de ocupação da memória dos três grupos de programas, com quantidades de memória medidas em *megabytes*. O Grupo 1, associado a um tamanho de *heap* de 60Mb, é composto por sete programas do conjunto; o Grupo 2 está associado a um *heap* de 100Mb e é composto por três programas; finalmente, o Grupo 3 está associado a um *heap* de 300Mb e é composto por quatro programas do conjunto DaCapo. As informações presentes nessas tabelas são um subconjunto dos dados constantes na Tabela 5.1, incluindo apenas a quantidade máxima de memória alocada pelo programa (*aloc*), a quantidade máxima de memória utilizada simultaneamente pelo programa (*útil*), e a proporção entre as duas últimas quantidades. Os programas foram separados em grupos de acordo com os valores da coluna *útil*.

Os programas em todos os grupos foram executados em um computador com processador Intel Core 2 Duo de 2.1GHz, que contém dois núcleos de processamento.

Os resultados para a execução dos programas de teste dos três grupos são mostrados na Tabela 5.6. Nos três grupos, pode-se observar que as tendências apresentadas nos resultados do primeiro experimento (Seção 5.7.3) também são apresentadas neste segundo experimento.

Tabela 5.5: *Perfil de ocupação da memória dos três grupos de programas. Memória máxima alocada e memória máxima utilizada, em* megabytes.

Grupo 1

Teste	aloc	util	aloc/util
jython	1198.3	0.2	5991.50
luindex	201.9	1.1	183.55
avrora	241.7	2.7	89.52
fop	100.2	7.7	13.01
lusearch	1793.3	10.9	164.52
pmd	781.8	13.7	57.07
xalan	60254.6	26.8	2248.31

Grupos 2 e 3

Teste	aloc	util	aloc/util
eclipse	5580	36.6	152.46
batik	1310.2	52.5	24.96
sunflow	911.6	63.8	14.29
h2	143.5	75.4	1.90
tomcat	934.7	78	11.98
tradebeans	17365.9	211.7	82.03
tradesoap	21327.4	274.1	77.81

Os tempos totais de execução continuam similares entre os três coletores, mas os tempos máximos de pausa dos mutadores é cerca de duas ordens de magnitude menor para o coletor proposto. Mesmo que os programas de teste executem em uma configuração em que o coletor de lixo é mais exigido, as tendências mais notáveis observadas no primeiro experimento não são alteradas, o que aumenta a confiança nas conclusões obtidas.

Para obter maiores detalhes sobre a execução dos programas nos três grupos, foram levantados perfis de execução de cada programa, utilizando funcionalidades presentes na Jikes RVM. Os resultados podem ser vistos nas Tabelas 5.7 (Grupo 1), 5.8 (Grupo 2) e 5.9 (Grupo 3). As tabelas mostram, para cada programa e cada coletor testado, o tempo total de execução em segundos, o número de execuções do coletor (no caso dos coletores paralelos) ou de ciclos de coleta (no caso do coletor baseado em contagem de referências), e a porcentagem do tempo total da execução do programa que foi gasta na execução do coletor de lixo. Observa-se que, na maioria dos programas de teste, o coletor é chamado várias dezenas de vezes, e que a porcentagem do tempo total utilizada pelo coletor é relativamente baixa, ficando na maior parte dos casos em menos de 10% do tempo total de execução. Isso indica que o tempo de execução dos programas é dominado pelos mutadores e não pelo coletor, e ajuda a explicar a pouca diferença medida nos tempos totais de execução para os três coletores. Entretanto, isso não significa que o desempenho do coletor é pouco significante para o desempenho total da plataforma – como pode ser visto comparando qualquer um dos três coletores em estudo nesse experimento com algum dos coletores mais básicos que são distribuídos com a Jikes RVM. Pode-se imaginar que, à medida em que o coletor se torna mais eficiente e toma uma menor porcentagem do tempo total de execução do programa, chega-se em uma situação de "rendi-

Tabela 5.6: *Resultados de execução dos programas nos três grupos. Tempo total em segundos e pausa máxima em milissegundos.*

	pargencopy		pargenms		rc-conc-n1	
Teste	tempo	pausa	tempo	pausa	tempo	pausa
Grupo 1 (*heap* de 60Mb)						
avrora	57.98	3212	58.11	3109	59.01	42
fop	23.61	3101	24.02	3008	24.37	32
jython	152.20	8396	150.74	8412	160.11	91
luindex	31.48	3825	32.88	3716	33.82	47
lusearch	108.77	5297	109.91	5381	109.20	58
pmd	69.21	6834	70.35	6870	73.02	73
xalan	66.81	5217	68.55	5188	70.01	50
Grupo 2 (*heap* de 100Mb)						
batik	55.11	3601	54.38	3108	54.72	32
eclipse	542.18	7422	546.20	6913	551.55	69
sunflow	71.33	2954	70.21	3010	72.11	26
Grupo 3 (*heap* de 300Mb)						
h2	76.23	6379	74.41	6184	79.01	57
tomcat	74.43	3771	77.73	3580	79.23	28
tradebeans	97.37	5120	99.24	5089	104.29	44
tradesoap	263.08	5373	270.15	5306	276.47	49

mentos decrescentes", na qual melhorias adicionais no desempenho do coletor têm impacto pequeno no desempenho geral do sistema. Os coletores paralelos e o coletor proposto baseado em contagem de referências aparentam, pelos testes realizados, estar na mesma região aproximada com relação à vazão (*throughput*), uma região em que vale a situação de rendimentos decrescentes.

Tabela 5.7: *Perfil de execução dos programas no Grupo 1: tempo total (segundos), número de execuções ou ciclos do coletor, e porcentagem do tempo de execução do coletor em relação ao tempo total.*

	pargencopy			pargenms			rc-conc-n1		
Teste	tempo	gc	gc%	tempo	gc	gc%	tempo	gc	gc%
avrora	57.98	38	1.83	58.11	36	1.79	59.01	22	1.77
fop	23.61	27	1.21	24.02	29	1.17	24.37	21	1.24
jython	152.20	913	18.87	150.74	931	18.79	160.11	92	20.07
luindex	31.48	11	2.41	32.88	11	2.39	33.82	14	2.40
lusearch	108.77	535	5.81	109.91	544	5.83	109.20	82	5.97
pmd	69.21	76	10.48	70.35	78	10.32	73.02	39	9.88
xalan	66.81	162	6.11	68.55	166	6.19	70.01	44	6.33

Tabela 5.8: *Perfil de execução dos programas no Grupo 2: tempo total (segundos), número de execuções ou ciclos do coletor, e porcentagem do tempo de execução do coletor em relação ao tempo total.*

	pargencopy			pargenms			rc-conc-n1		
Teste	tempo	gc	gc%	tempo	gc	gc%	tempo	gc	gc%
batik	55.11	37	2.31	54.38	37	2.29	54.72	20	2.39
eclipse	542.18	291	6.67	546.20	296	6.71	551.55	198	6.32
sunflow	71.33	70	0.91	70.21	68	0.89	72.11	37	0.96

Diferenças pequenas no desempenho medido, como observadas nos resultados das Tabelas 5.2 e 5.6, podem ter origem em flutuações aleatórias e não serem estatisticamente significativas. Uma análise estatística das diferenças pode determinar, em um certo nível de confiança, se existe uma diferença real de desempenho entre os diferentes coletores. Com esse intuito, considerando que os valores relatados nas Tabelas 5.2 e 5.6 são médias de um conjunto de execuções repetidas dos programas de teste, foi realizada uma análise da diferença entre as médias dos coletores, calculando-se intervalos de confiança para as diferenças. O método uti-

Tabela 5.9: *Perfil de execução dos programas no Grupo 3: tempo total (segundos), número de execuções ou ciclos do coletor, e porcentagem do tempo de execução do coletor em relação ao tempo total.*

	pargencopy			pargenms			rc-conc-n1		
Teste	tempo	gc	gc%	tempo	gc	gc%	tempo	gc	gc%
h2	76.23	74	9.52	74.41	79	9.13	79.01	68	10.03
tomcat	74.43	68	1.14	77.73	64	1.09	79.23	43	1.31
tradebeans	97.37	96	4.46	99.24	100	4.79	104.29	74	5.17
tradesoap	263.08	104	4.78	270.15	101	5.2	276.47	126	5.31

lizado foi o mesmo relatado na Seção 4.5.1[10]: cálculo de intervalos pivotais de *bootstrap* em nível de 99% de confiança, com 1000 replicações [96].

Duas comparações foram realizadas: a primeira entre os resultados do coletor **rc-conc-n1** (baseado no algoritmo proposto de contagem de referências) e o coletor **pargencopy** (paralelo generacional de cópia, distribuído com a Jikes RVM); a segunda entre o coletor **rc-conc-n1** e o coletor **pargenms** (paralelo generacional de marcação e varredura, também distribuído com a Jikes RVM).

Na Figura 5.6 são mostrados os intervalos de confiança calculados para a diferença **rc-conc-n1 − pargencopy**. Dos intervalos calculados para os 14 testes (numerados de acordo com a ordem dos programas na Tabela 5.6), cinco contêm o valor 0, o que indica que não há uma diferença estatisticamente significativa, no nível de confiança de 99%, entre o desempenho dos dois algoritmos. Entretanto, os demais nove intervalos não contêm o valor 0, demonstrando que há uma diferença de desempenho significativa nesse nível de confiança entre os dois coletores, em favor do coletor paralelo de cópia. Entretanto, como já mencionado antes, a diferença é pequena em relação ao tempo total de execução, ficando em menos de 10% em todos os testes.

Os intervalos de confiança calculados para a comparação entre o coletor **rc-conc-n1** e o coletor **pargenms** são mostrados na Figura 5.7. Dos 14 intervalos calculados, seis contêm o valor zero e representam testes para os quais, no nível de confiança de 99%, não houve diferença estatisticamente significativa. Em um dos testes, o teste 5 (*lusearch*) houve uma diferença significativa em favor do coletor **rc-conc-n1**, enquanto nos demais sete testes houve uma diferença significativa em favor do coletor **pargenms**. Mais uma vez, as diferenças observadas são pequenas em relação ao tempo total de execução, ficando em menos de 10%.

[10] Na mesma seção é apresentada uma justificativa para o uso de tal método específico

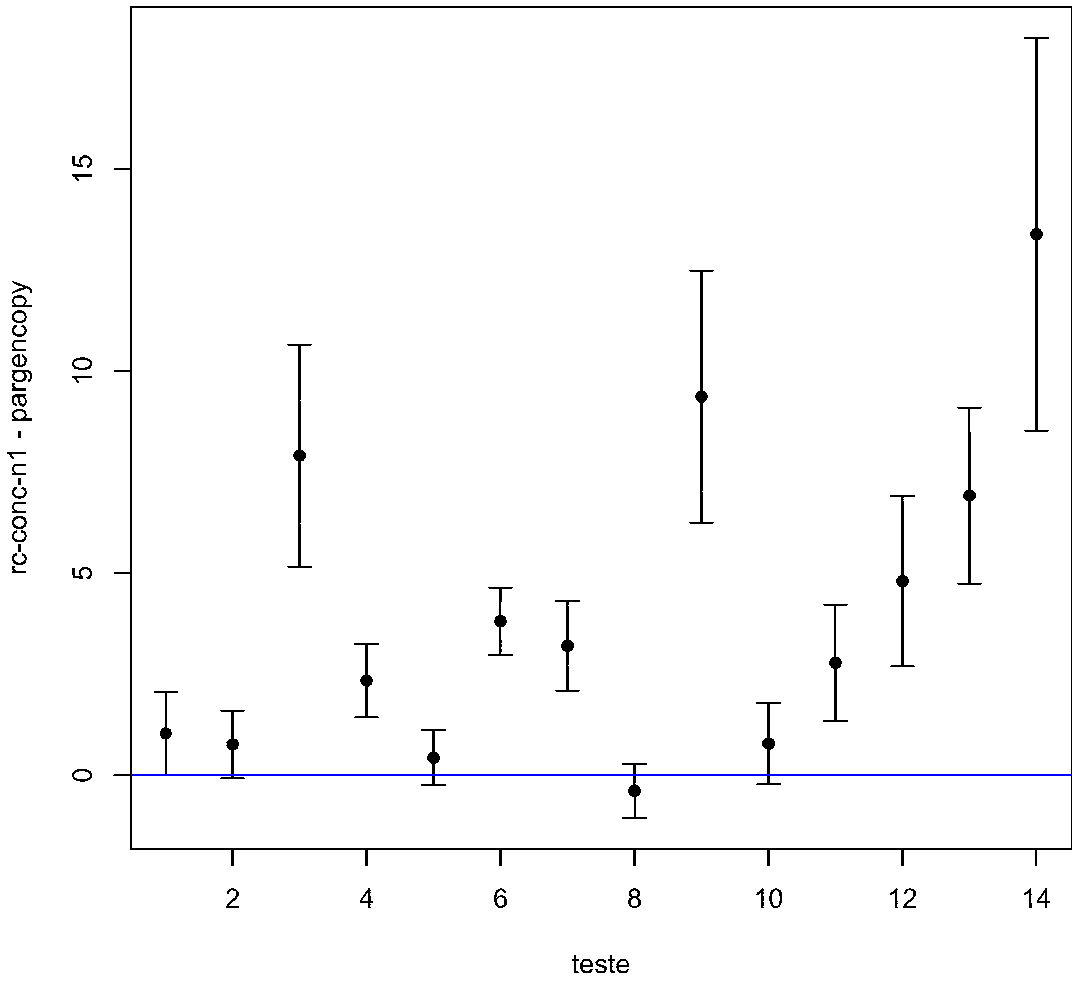

Figura 5.6: *Intervalos de confiança para a diferença entre as médias dos tempos de execução dos programas dos três grupos usando o coletor* ***rc-conc-n1*** *e o coletor* ***pargencopy****. Os programas de teste de 1 a 14 estão na ordem mostrada na Tabela 5.6.*

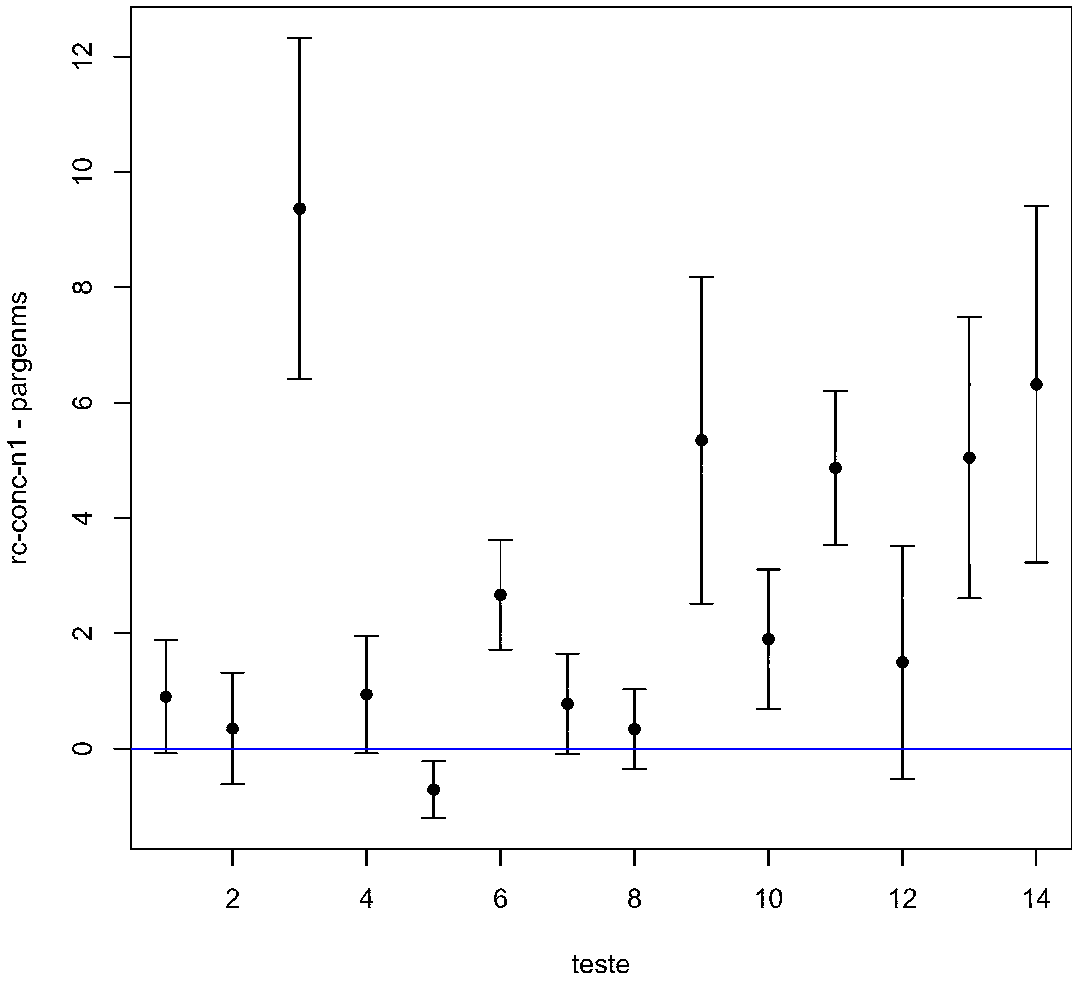

Figura 5.7: *Intervalos de confiança para a diferença entre as médias dos tempos de execução dos programas dos três grupos usando o coletor* ***rc-conc-n1*** *e o coletor* ***pargenms****. Os programas de teste de 1 a 14 estão na ordem mostrada na Tabela 5.6.*

Podemos concluir que, embora existam testes para os quais o coletor baseado no algoritmo de contagem de referências da Seção 5.3 apresenta desempenho (em termos de *throughput*) menor que os coletores paralelos, as diferenças são pequenas, em alguns casos sendo estatisticamente insignificantes no nível de confiança de 99%. Isso reforça a confiança na conclusão que o coletor proposto, baseado no novo algoritmo de contagem de referências para múltiplos mutadores, tem desempenho competitivo com coletores alternativos para sistemas multiprocessados, mas possui a vantagem de impor tempos de pausa bem menores aos mutadores. Como as diferenças entre os tempos máximos de pausa medidos foram consideráveis, chegando a duas ordens de magnitude, julgou-se que tais diferenças não poderiam ser explicadas apenas por flutuações aleatórias de execução e medição, e portanto não foi feita uma análise estatística sobre essas diferenças.

5.7.5 Resultados: Terceiro Experimento

À medida em que o nível de integração e o número de transistores por pastilha em arquiteturas *multicore* cresce, a tendência é que o número de núcleos de processamento continue a crescer. Desta forma, é importante determinar como o desempenho do coletor proposto muda com uma variação no número de processadores usados. O terceiro experimento foi criado para medir as características de desempenho dos coletores testados. Os programas de teste foram executados em um computador com processador Intel Core i5-750 de 2.66GHz, que possui quatro núcleos de processamento. Cada teste foi repetido três vezes (cada teste consistindo em 15 execuções): uma vez para o programa executando em apenas um processador, uma vez com dois processadores e uma última com quatro processadores. A máquina virtual Jikes RVM foi configurada com um *heap* de 300Mb. Foram medidos o tempo total de execução e o tempo máximo de pausa dos mutadores.

Os resultados para o tempo total de execução são apresentados na Tabela 5.10. Os tempos de execução indicam que o aumento no número de processadores tem um impacto considerável no desempenho dos programas, o que pode ser explicado pelo fato de todos os programas utilizarem vários *threads* na sua execução. Entretanto, o desempenho relativo dos coletores não apresenta tendências diferentes dos resultados vistos anteriormente para dois processadores. Os tempos medidos ainda são bastante próximos, em geral não diferindo por mais de 10%. Na Figura 5.8 são mostrados os resultados para quatro processadores em gráfico de barras. Observando a Figura 5.8, pode-se notar a evidência visual da proximidade nos tempos medi-

Tabela 5.10: *Tempo total de execução dos programas de teste usando 1, 2 e 4 CPUs (*cores*).*

	pargencopy			pargenms			rc-conc-n1		
Teste	1	2	4	1	2	4	1	2	4
avrora	71.11	39.23	27.93	75.65	33.30	26.55	78.42	35.56	25.92
fop	31.72	13.46	9.12	34.06	13.16	10.07	30.85	14.36	9.75
jython	207.30	115.10	78.03	206.06	114.45	77.08	209.83	106.76	79.73
luindex	44.86	16.26	10.87	44.39	18.34	11.37	44.96	17.35	10.39
lusearch	141.43	75.09	52.09	137.09	67.59	49.93	141.64	68.90	51.69
pmd	98.40	42.98	28.81	90.46	46.18	27.47	89.48	45.08	29.13
xalan	86.13	48.28	31.40	90.28	42.46	29.92	91.22	46.77	31.36
batik	71.77	29.53	19.94	78.25	31.35	20.43	75.52	31.91	21.04
eclipse	687.72	402.33	318.14	666.37	382.39	305.51	673.64	408.66	323.09
sunflow	101.97	40.91	29.62	96.35	47.35	31.33	89.65	40.82	29.85
h2	101.67	50.40	28.10	102.97	46.48	29.84	105.09	49.10	30.49
tomcat	107.76	39.54	29.31	98.71	44.33	28.65	101.66	48.54	27.99
tradebeans	141.92	51.17	31.99	139.86	69.03	33.03	149.67	64.05	34.89
tradesoap	352.83	173.52	106.63	369.92	159.01	113.84	365.02	166.04	110.69

dos.

Apesar dos resultados indicarem a continuação da tendência de similaridade entre o desempenho dos coletores, observada nos resultados dos experimentos anteriores, os números parecem, em uma primeira observação, mais favoráveis ao coletor de contagem de referências quando a execução usa quatro processadores. Uma comparação mais detalhada e levando em considerações as flutuações estatísticas na obtenção dos dados requer o cálculo de intervalos de confiança na comparação entre as médias, como realizado no segundo experimento (Seção 5.7.4). Na Figura 5.9 são mostrados os intervalos de confiança calculados seguindo o mesmo método da Seção 5.7.4 – intervalos pivotais de *bootstrap* com nível de confiança de 99% e 1000 replicações. Nesse nível de confiança, pode-se ver que há mais intervalos que contêm o valor 0: nove na comparação entre **rc-conc-n1** e **pargencopy** e oito na comparação entre **rc-conc-n1** e **pargenms**. Isso indica que, em mais da metade dos testes, não houve diferença estatisticamente significativa entre os coletores, no nível de confiança indicado. Também há mais intervalos mostrando uma vantagem estatisticamente significativa para o coletor proposto baseado em contagem de referências. Isso indica que o coletor proposto tem uma pequena melhoria de desempenho relativo aos coletores paralelos usados como comparação. Entretanto, para concluir isso com maior confiança seriam necessários mais testes, usando um

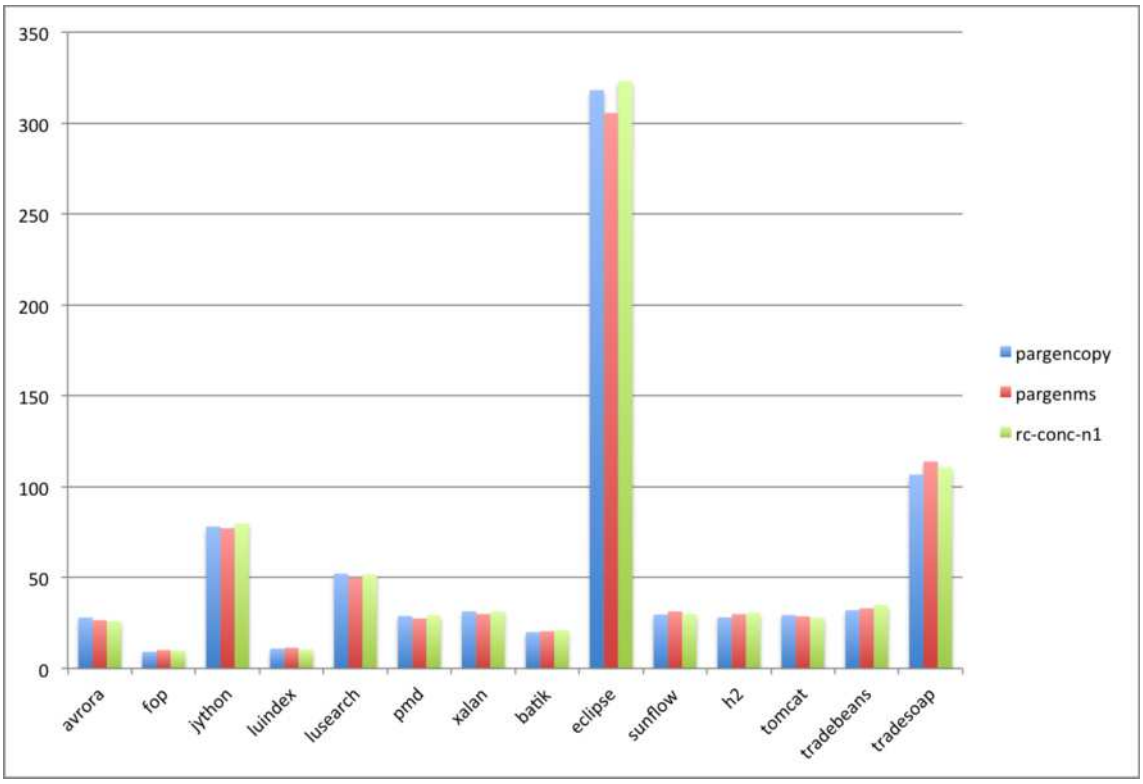

Figura 5.8: *Tempos de execução (em segundos) para os* benchmarks *DaCapo nos três coletores considerados, com quatro processadores.*

maior número de processadores.

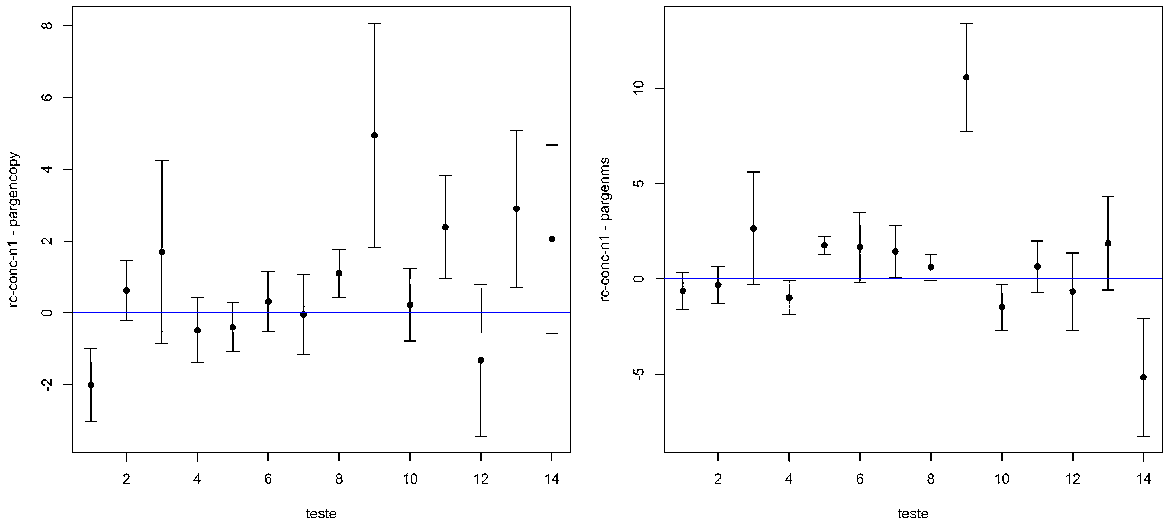

Figura 5.9: *Intervalos de confiança (em nível de 99%) calculados para a diferença entre as médias dos tempos de execução entre os coletores* ***rc-conc-n1*** *e* ***pargencopy*** *(esquerda) e entre* ***rc-conc-n1*** *e* ***pargenms*** *(direita).*

Os tempos máximos de pausa do mutador medidos no terceiro experimento são mostrados na Tabela 5.11. Mais uma vez, observa-se uma grande diferença no tempo máximo de pausa imposto pelo coletor proposto baseado em contagem de referências e pelos coletores paralelos. O tempo de pausa do coletor proposto é cerca de duas ordens de magnitude menor que o dos coletores paralelos.

A mudança no número de processadores, entretanto, parece indicar uma tendência de pequena melhoria no tempo de pausa dos coletores paralelos em relação ao coletor proposto. Na Figura 5.10 é mostrada a evolução dos tempos de pausa dos três coletores de acordo com o número de processadores usados na execução. Assim como na Figura 5.5, os tempos de pausa dos coletores paralelos foram divididos por 100 para compatibilizar a escala. De fato, como pode ser visto na Figura 5.10, há uma aparente melhoria no tempo de pausa dos coletores paralelos em relação ao coletor proposto. Provavelmente o número maior de processadores torna o processo de coleta mais rápido, reduzindo o tempo que os coletores paralelos precisam parar o mutador. No coletor proposto, os tempos de pausa já são relativamente pequenos, e caem menos com o aumento do número de processadores. É importante ressaltar que, embora haja uma aparente melhoria relativa no tempo de pausa dos coletores paralelos, a diferença a favor do coletor proposto ainda é bastante significativa.

Tabela 5.11: *Tempo máximo de pausa dos mutadores (em milissegundos) nos programas de teste executando em 1, 2 e 4 CPUs (*cores*).*

	pargencopy			pargenms			rc-conc-n1		
Teste	1	2	4	1	2	4	1	2	4
avrora	4896	3167	2208	4931	3205	2199	42	36	28
fop	3755	2804	1936	3789	2916	1977	34	31	25
jython	9352	6164	4704	9199	5934	4662	79	54	48
luindex	4001	2844	1893	3867	2633	1845	31	26	22
lusearch	6948	5176	3812	6690	5195	3781	58	48	40
pmd	9224	6434	4941	9373	6533	5022	97	80	66
xalan	4221	3086	2077	4358	3297	2184	42	36	29
batik	4966	3944	3114	5024	4162	3182	48	42	36
eclipse	8770	6324	4690	8709	6078	4587	81	62	50
sunflow	3995	3046	2368	3871	2925	2306	36	28	25
h2	8287	6015	4379	8290	5877	4321	77	55	49
tomcat	8439	6193	5113	8511	6383	5204	82	61	54
tradebeans	8483	6344	4961	8502	6206	4879	80	61	56
tradesoap	5573	4524	3452	5668	4742	3547	52	42	38

Portanto, à medida que o número de processadores usados na execução cresce, verificou-se que as tendências gerais do desempenho dos coletores não se alterou muito, mas duas tendências secundárias, de menor efeito, foram identificadas:

1. A vazão (*throughput* do coletor proposto de contagem de referências parece melhorar ligeiramente em relação aos coletores paralelos.
2. A latência (medido como tempo de pausa do mutador) dos coletores paralelos parece melhorar em relação ao coletor proposto.

Entretanto, os efeitos secundários observados ainda são preliminares e precisariam de mais experimentos para serem confirmados. Especialmente, seria importante testar os coletores em questão em um maior número de processadores.

5.7.6 Resultados: Quarto Experimento

Neste experimento os programas do conjunto DaCapo foram executados em um computador com processador Intel Core i5-750 de 2.66GHz, com 4Gb de RAM, alterando o algoritmo de coleta entre as duas variantes do algoritmo proposto: com uma fila de atualização por mutador (Seção 5.3) e com duas filas de atualização por mutador (Seção 5.4). O objetivo foi

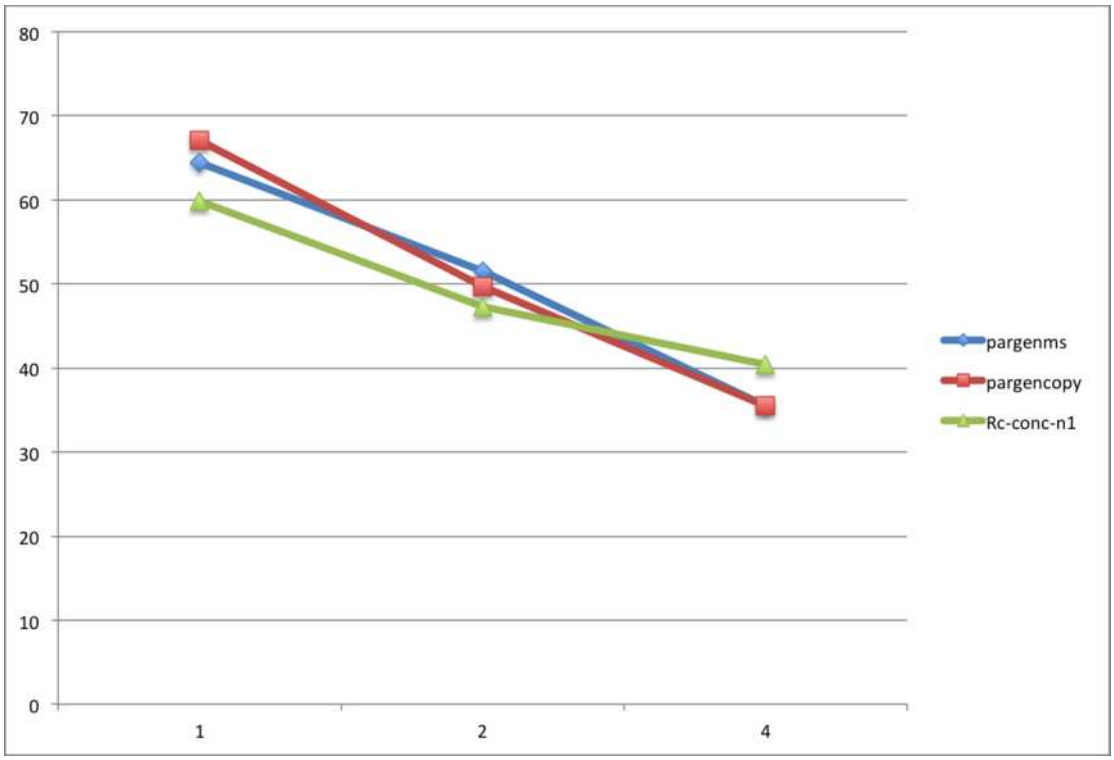

Figura 5.10: *Tendência do tempo de pausa (eixo Y) em relação ao número de processadores usados na execução. Os tempos no eixo Y foram alterados para compatibilizar a escala.*

comparar as variantes e verificar se, como pretendido, a variante com duas filas por mutador apresentava menores tempos de pausa. Todos os testes foram executados usando os quatro núcleos de processamento do processador Core i5.

Entretanto, observando os algoritmos que mudam entre as duas variantes (as rotinas UPDATE e COLLECTORMAIN), é possível notar que não parece haver grande diferenças, apenas um mecanismo diferente de sincronização das filas do mutador com o *buffer* do coletor. Como as questões de concorrência no algoritmo são delicadas, esse nível de sincronização é necessário em ambas as versões. Isso indica que pode não haver diferença significativa entre o desempenho das duas versões, indicação que é confirmada pelos resultados experimentais, como mostrado a seguir.

Os resultados dos testes são mostrados na Tabela 5.12, na qual estão indicados os tempos totais de execução em segundos e o tempo máximo de pausa dos mutadores, em milissegundos, para cada programa de teste nas duas variantes do algoritmo. Pode-se perceber que os resultados são bastante similares. Na maioria dos testes a variante com duas filas parece apresentar tempo de pausa menor que a variante com uma fila, mas as diferenças são menores

que 10% em todos os casos, e menores que 5% na maioria.

Tabela 5.12: *Tempo total de execução (segundos) e tempo máximo de pausa do mutador (milissegundos) para os programas de teste usando as duas variantes do algoritmo proposto.*

	1 fila		2 filas	
Teste	tempo	pausa	tempo	pausa
avrora	25.92	28	25.76	27
fop	9.75	25	10.11	26
jython	79.73	48	77.96	45
luindex	10.39	22	10.28	21
lusearch	51.69	40	52.30	41
pmd	29.13	66	28.49	63
xalan	31.36	29	32.03	28
batik	21.04	36	21.90	36
eclipse	323.09	50	326.77	48
sunflow	29.85	25	29.52	24
h2	30.49	49	31.47	50
tomcat	27.99	54	28.10	51
tradebeans	34.89	56	32.99	54
tradesoap	110.69	38	114.24	39

Para obter uma caracterização mais adequada para as diferenças entre as variantes, levando em consideração as flutuações aleatórias que ocorrem no processo experimental, foram calculados intervalos de confiança para as diferenças entre as médias dos tempos de execução e tempos máximos de pausa, em nível de 99%, utilizando o mesmo método dos experimentos anteriores (*bootstrap* com 1000 replicações).

Os intervalos de confiança calculados são mostrados na Figura 5.11, onde pode-se ver que, no caso dos tempos medidos, nove dos 14 intervalos de confiança incluem o zero. Dos cinco que não incluem o zero, três apresentam vantagem para a variante com uma fila de atualização (v1), e dois apresentam vantagem para a variante com duas filas (v2). Isso indica que, no nível de confiança de 99%, há pouca diferença na vazão das duas variantes, não sendo essa diferença estatisticamente significante na maioria dos casos e, mesmo quando a diferença é significativa, não há uma das variantes que seja claramente superior à outra.

As diferenças são ainda menores nos intervalos calculados para os tempos de pausa. Dos 14 intervalos, apenas dois não incluem a origem. Portanto, no nível de confiança utilizado, parece claro que não há uma diferença estatisticamente significativa entre os tempos de pausa observados nas duas variantes. Como observado, as necessidades de sincronização entre co-

letor e mutador são bastante similares nas duas variantes, seja usando uma ou duas filas de atualização por mutador. Com isso, não há vantagem aparente em usar duas filas de atualização por mutador, já que mais memória é necessária para cada mutador ao se utilizar duas filas. O algoritmo proposto que usa apenas uma fila de atualização por mutador utiliza menos memória e tem desempenho similar, tanto em termos de vazão quanto de latência, à variante com duas filas por mutador.

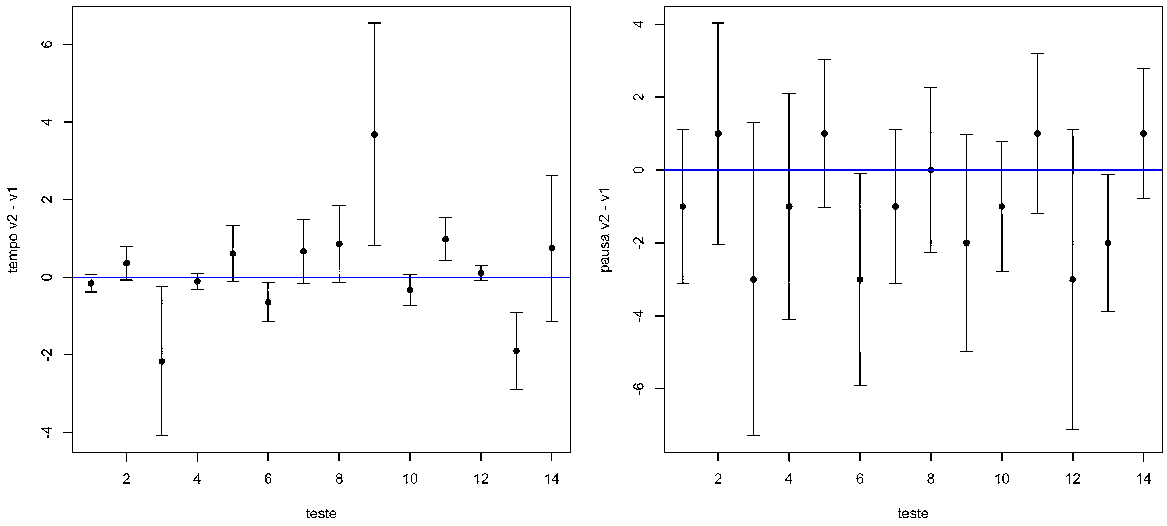

Figura 5.11: *Intervalos de confiança (em nível de 99%) calculados para a diferença entre as médias dos tempos de execução entre as variantes com uma e duas filas de atualização por mutador (esquerda) e os intervalos calculados para a diferença entre o tempo máximo de pausa das duas variantes (direita).*

Embora os testes realizados com a implementação do coletor proposto tenham demonstrado que:

- ▷ O coletor funciona no conjunto de programas de teste DaCapo;
- ▷ O coletor tem desempenho competitivo com os coletores paralelos presentes na Jikes em relação à vazão e é consideravelmente superior em relação à latência,

é importante determinar a corretude do algoritmo de coleta de lixo proposto nesta tese. A corretude do algoritmo é considerada a seguir.

capítulo 6

Considerações Sobre a Corretude dos Algoritmos

Os algoritmos de contagem de referências apresentados nos capítulos 4 e 5 se mostraram adequados para uso em sistemas multiprocessados, com bom desempenho em relação às alternativas disponíveis, e com o diferencial de imporem pausas ao programa do usuário que podem ser ordens de magnitude menores que em outros algoritmos (ver Seção 5.7).

Entretanto, um bom desempenho e pequenos tempos de pausa não importam se o algoritmo não funciona corretamente de acordo com sua especificação. Neste capítulo são feitas considerações sobre a corretude dos algoritmos apresentados para contagem de referências cíclicas em sistemas multiprocessados, com o objetivo final de apresentar uma prova informal da corretude do algoritmo para vários mutadores e um coletor da Seção 5.3.

As seções a seguir analisam a corretude de vários algoritmos apresentados nos Capítulos 2 a 5, observando a manutenção da propriedade de segurança e da propriedade de vivacidade em cada um. Antes disso, são feitas algumas considerações sobre a dificuldade de obter uma prova formal sobre a corretude dos algoritmos de contagem de referências cíclicas.

6.1 Dificuldades para uma Prova Formal

Os algoritmos baseados em contagem de referências foram desenvolvidos e pesquisados ao longo de décadas, como detalhado no Capítulo 2. O primeiro algoritmo para contagem

de referências cíclicas, que resolve o problema dos ciclos de lixo, foi publicado em 1990 por Martínez, Wachenchauzer e Lins [72]. Após isso, várias outras versões do algoritmo de contagem de referências cíclicas foram propostas, trazendo melhorias de desempenho consideráveis [64, 67, 69, 71].

Esses algoritmos foram implementados em sistemas experimentais e em sistemas usados em produção, como a máquina virtual Jalapeño da IBM [1]. Provas informais da corretude dos algoritmos podem ser encontradas em alguns dos artigos originais que os propõem. Entretanto, é desejável ter uma prova formal da corretude dos algoritmos a partir de suas especificações.

A necessidade de uma prova formal se torna ainda mais importante para os algoritmos criados para multiprocessadores. Defeitos relacionados à concorrência entre processos que compartilham memória frequentemente são sutis e difíceis de detectar [49].

Juntamente com o desenvolvimento dos novos algoritmos, foram realizadas tentativas de provar os algoritmos de contagem de referências cíclicas formalmente, tanto pelo autor da presente tese como por outros pesquisadores. Essas tentativas envolveram vários métodos formais: redes de Petri [88], CCS [80], *π-calculus* [81] e outras álgebras de processo [38]. Também considerou-se utilizar métodos baseados em asserções, como a lógica de Owicki-Gries-Hoare [3], ou especificar o sistema com algum método baseado em lógicas temporais, como a TLA+ de Lamport [59].

A maior dificuldade experimentada pelo autor deste trabalho ao tentar criar uma prova formal da corretude dos algoritmos de contagem de referências cíclicas foi a modelagem do *heap*. Para simplificar a modelagem, pode-se assumir que o *heap* é infinito, mas o estado do algoritmo depende da conectividade do grafo da memória. A princípio, não há um limite prévio para o número de referências que um objeto pode ter apontando para ele, e portanto o contador de referências pode atingir valores arbitrários, embora na prática a maioria dos objetos tenham contadores de referências com pequenos valores [61]. Não é fácil modelar características essenciais ao algoritmo, como o fato de um objeto estar transitivamente conectado à raiz, sem modelar um *heap* complexo, contendo um grafo direcionado de conectividade arbitrária, incluindo ciclos.

Por isso mesmo, as provas informais criadas para os algoritmos tendem a trabalhar com a manutenção de invariantes, e a prova da manutenção desses invariantes muitas vezes é feita de maneira indireta, por *reductio ad absurdum*. A experiência do autor desta tese indicou que uma prova direta, embora possa ser mais instrutiva e mais fácil de formalizar, é mais difícil

de criar para esse tipo de algoritmo.

6.2 Propriedades de Segurança e Vivacidade

Quando se discute sobre propriedades temporais de algoritmos, é comum separar essas propriedades em dois tipos: propriedades de *segurança* (*safety*) e propriedades de *vivacidade* (*liveness*). Em termos gerais, se um algoritmo possui uma propriedade de segurança, existe uma garantia de que algo "ruim" (algum estado indesejável) nunca pode ocorrer; para as propriedades de vivacidade, a garantia é que algum estado desejável será eventualmente atingido pelo algoritmo.

Para algoritmos de coleta de lixo, pode-se destacar uma propriedade importante de cada tipo: a propriedade de segurança é que os objetos ou células retornados à lista livre são efetivamente lixo, ou seja, não estão transitivamente conectados à raiz; a propriedade de vivacidade é que os objetos ou células que são lixo são eventualmente retornados à lista livre. Um algoritmo de coleta de lixo que não cumpre a propriedade de segurança pode retornar à lista livre objetos ou células ainda em uso, que poderão então ser alocados novamente; isso pode ser a origem de defeitos (*bugs*) difíceis de serem detectados no programa. Se o algoritmo de coleta não satisfaz a propriedade de vivacidade, alguns objetos ou células que não estão mais em uso ficarão indisponíveis para o programa, configurando um vazamento de memória (*space leak*). Embora as consequências da violação da segurança pareçam mais sérias, vazamentos de memória, se frequentes, podem degradar significantemente o desempenho do programa (devido às trocas de páginas no sistema de memória virtual) ou até mesmo fazer com que o programa seja terminado pelo sistema operacional. Desta forma, para algoritmos de coleta de lixo gerais, é importante ter as duas propriedades satisfeitas.

6.3 Algoritmos Sequenciais

A análise da maioria dos algoritmos baseados em contagem de referência é simplificada se for utilizada a seguinte propriedade invariante:

- ▷ **Invariante**: quando o programa do usuário (mutador) está executando, o contador de referências de um objeto O tem valor igual ao número de objetos fora da lista livre que têm uma referência para O.

Algoritmo básico Com relação ao algoritmo básico de contagem de referências, apresentado no Capítulo 2 (Algoritmos 2.8 e 2.9), uma observação importante aqui é que a invariante não diz que o contador de referências de O tem valor igual ao número de referências para O que estão transitivamente conectadas à raiz; esse invariante seria inválido quando O estivesse no fecho transitivo de algum ciclo que não estivesse transitivamente conectado à raiz. Esse é o problema já mencionado do algoritmo de contagem de referências: não detectar o isolamento – em relação à raiz – de objetos no fecho transitivo de algum ciclo.

O algoritmo é sequencial; durante a ação do coletor, o invariante pode ser transitoriamente inválido – as atualizações no contador ocorrem em tais momentos – mas ao fim de cada operação o invariante é reestabelecido corretamente. Desta forma, sempre que o mutador está executando, o invariante é mantido. A manutenção da propriedade de segurança pode ser concluída através do invariante: se um objeto O tem seu contador de referências com valor zero, significa que as únicas referências que podem existir para O são de objetos na lista livre; portanto, O não está em uso pelo programa e pode ser reaproveitado (adicionado à lista livre).[1]

Entretanto, o algoritmo básico de contagem de referências não garante a propriedade de vivacidade. O contra-exemplo é a existência de ciclos que podem perder suas conexões com a raiz e não serem liberados, pois as referências internas vindas de objetos do ciclo fazem com que o contador de referências dos objetos no fecho transitivo desse ciclo nunca chegue a ter o valor zero.

Contagem de referências cíclicas No algoritmo de contagem de referências cíclicas do Capítulo 3, o invariante acima ainda é mantido. Durante as operações do coletor, que incluem as operações de rastreamento local para busca de ciclos (ScanStatusAnalyser, Algoritmo 3.5), o contador de referências pode ter seu valor alterado de maneira que o invariante seja momentaneamente quebrado, mas quando o mutador executa, o invariante sempre é mantido. A análise das propriedades de segurança e vivacidade, entretanto, é afetada pela existência do *status analyser*. Qualquer objeto cujo contador de referências tenha valor maior que um em algum momento será adicionado ao *status analyser* – ou estará transitivamente

[1] Um caso limite é se um objeto O_1 aponta para O, e essa referência será transferida de forma que, ao final, O_2 esteja apontando para O, e não O_1. Mesmo que a referência de O_1 seja excluída antes da referência em O_2 ser criada, a única forma que o programa tem de estabelecer uma referência de O_2 para O é se essa referência estiver guardada em algum objeto ativo (uma variável temporária, em muitos casos). Portanto, o contador de referências de O nunca terá valor 0 durante a transferência.

conectado a um objeto no *status analyser*.

Segurança Para analisar a segurança, note-se que objetos podem ser liberados apenas em DELETE (Algoritmo 3.4) ou em COLLECT (Algoritmo 3.8). COLLECT é chamada pela rotina SCANSTATUSANALYSER caso, após passar por MARKRED e SCANGREEN, a célula ainda esteja marcada com a cor vermelha. Em MARKRED (Algoritmo 3.6), vê-se que os objetos no *status analyser* são pintados de vermelho e têm seus contadores de referências subtraídos de acordo com o número de outros objetos vermelhos que apontam para ele (linha 5). Depois da marcação recursiva de objetos para a cor vermelha e redução dos contadores de referências para descontar as referências internas ao ciclo (ou objetos no fecho transitivo do mesmo), objetos que ainda possuem o contador de referências com valor maior que zero estão conectados transitivamente à raiz, e são adicionados ao *status analyser* para serem analisados posteriormente (linhas 11 e 12 de MARKRED, Algoritmo 3.6). Portanto, se um objeto analisado pelo rastreamento local estiver conectado transitivamente à raiz, ele será adicionado ao *status analyser*; de volta a SCANSTATUSANALYSER, se um objeto tiver a cor vermelha e o contador de referências com valor maior que zero, o mesmo será passado para a rotina SCANGREEN (Algoritmo 4.7), como pode ser visto nas linhas 9 e 10 de SCANSTATUSANALYSER. Em SCANGREEN, o objeto é pintado de verde (linha 2) e os contadores de referência no seu fecho transitivo são reestabelecidos para incluir as referências desconsideradas em MARKRED. Portanto, um objeto que está transitivamente conectado à raiz, mesmo que seja analisado em SCANSTATUSANALYSER, não será liberado, garantindo a propriedade de segurança.

Vivacidade Para a propriedade da vivacidade, o caso que não é coberto pelo algoritmo básico de contagem de referências é o de ciclos isolados da raiz, nos quais pelo menos um objeto no seu fecho transitivo permanece com contador de referências com valor maior que um devido às referências internas originadas do ciclo. Como esse é o único caso em que a propriedade de vivacidade não é observada no algoritmo básico, esse é o caso que será analisado aqui. Como já mencionado no Capítulo 3, um ciclo será isolado quando a última referência originada (transitivamente) da raiz para algum objeto O do ciclo for eliminada. Nesse caso, o objeto O deverá ter, antes da eliminação, um contador de referências com valor no mínimo igual a dois, sendo uma dessas referências a referência externa, e as demais são possíveis referências internas ao ciclo, das quais deve existir pelo menos uma para que seja um ciclo. A eliminação da última referência externa ao ciclo faz com que O seja passado à rotina DELETE

(Algoritmo 3.4). Como o contador de referências de O nesse caso é maior que um, a parte *else* do comando condicional da linha 2 será executada, a partir da linha 9. Isso faz com que o objeto O seja adicionado ao *status analyser* e marcado com a cor preta. Como não há outras referências para objetos no fecho transitivo do ciclo, todos esses objetos permanecerão no mesmo estado até que o objeto O no *status analyser* seja analisado; o mutador não tem referências aos objetos do fecho transitivo do ciclo para realizar alterações no estado dos mesmos. Assim, eventualmente O será analisado em ScanStatusAnalyser, e como sua cor é preta, MarkRed será chamada (linhas 6 e 7 de ScanStatusAnalyser). MarkRed marcará, recursivamente, todos os objetos no fecho transitivo de O como vermelhos e reduzirá os contadores de referência dos mesmos objetos para descontar as referências internas. Como não há objetos no fecho de O que estejam conectados transitivamente à raiz, ao final do processamento de MarkRed, todos os objetos no fecho de O terão contador de referências com valor zero. Portanto, nenhum desses objetos será adicionado ao *status analyser* (linhas 11 e 12 de MarkRed) e ScanGreen nunca será chamada em nenhum deles (linhas 9 e 10 de ScanStatusAnalyser). O resultado final é que os objetos no fecho transitivo de O permanecerão com a cor vermelha, e eventualmente a execução de ScanStatusAnalyser chegará na chamada a Collect tendo O como argumento (linhas 13 e 14 de ScanStatusAnalyser). Collect executará a liberação de O (linhas 8 e 9) e irá recursivamente percorrer os objetos no fecho de O, liberando-os. Isso garante que a propriedade de vivacidade é mantida, pois os ciclos são eventualmente liberados pelo procedimento de rastreamento local iniciado a partir de ScanStatusAnalyser.

6.4 Corretude do Algoritmo com Um Mutador

O algoritmo para um mutador e um coletor usando fila de atualizações (Seção 4.3.2) usa rotinas para rastreamento local de ciclos que são bastante similares às do algoritmo de contagem de referências cíclicas analisado na seção anterior. Entretanto, as operações do mutador (especificamente Update) não alteram o valor dos contadores de referência diretamente, apenas incluindo registros na fila de atualizações. O coletor processa a fila de atualizações periodicamente e realiza as alterações necessárias nos contadores de referência, e isso ocorre sem nenhuma interferência do mutador.

A diferença mais importante ocorre no processo de rastreamento local, pois enquanto o coletor realiza o rastreamento, o mutador pode alterar referências de, ou para, objetos que

estão sendo rastreados. Uma observação aqui torna a verificação da vivacidade mais simples: a propriedade de um objeto ser *lixo*, ou seja, não estar transitivamente conectado à raiz, é uma propriedade estável com relação às operações do mutador. Isso significa que um objeto que seja lixo, ou um ciclo isolado de lixo, não pode deixar de ter tal propriedade devido a nenhuma operação do mutador. Como observado anteriormente, o mutador não possui mais referências a objetos que sejam lixo, e portanto não pode alterá-los. Assim, se um ciclo for isolado, o algoritmo da Seção 4.3.2 irá recuperar os objetos no fecho transitivo do ciclo, como analisado no caso sequencial. Isso garante que a propriedade da vivacidade é mantida para objetos em ciclos de lixo.

6.5 Corretude do Algoritmo com Vários Mutadores

Esta seção analisa a corretude do algoritmo para um coletor e vários mutadores, apresentado na Seção 5.3 e implementado na máquina virtual Java Jikes RVM, como descrito na Seção 5.6.

O algoritmo para vários mutadores da Seção 5.3 é baseado no algoritmo com fila de atualizações da Seção 4.3.2, cuja corretude é analisada na Seção 6.5. A questão da segurança é similar ao caso do algoritmo com apenas um mutador. A diferença é que o coletor reúne todas as informações nas filas de atualização de cada mutador no início do ciclo de coleta, e precisa integrar essas informações para atualizar os contadores de referências de todos os objetos que aparecem nas filas de atualização.

Uma questão importante na integração entre os conteúdos das filas de atualização é garantir que apenas um registro de atualização esteja presente para cada objeto cujas referências para ele foram atualizadas no ciclo de coleta atual. Assim, o contador de referências de cada objeto é atualizado apenas uma vez em cada ciclo. Isso é garantido na rotina UPDATE (Algoritmo 5.1) através do uso do *flag updated*. Em UPDATE, o *flag updated* de cada objeto é verificado antes da inserção de um registro na fila de atualizações; se o objeto já foi atualizado no ciclo atual, um registro não é inserido. Isso garante que só há um registro para um dado objeto em cada ciclo.

Segurança No coletor, o procedimento UPDATEREFERENCECOUNTERS (Algoritmo 4.10) tem a responsabilidade de atualizar os valores dos contadores de referências, baseado no conteúdo do *buffer* construído a partir das filas de atualização dos mutadores. Se uma referência

a um objeto foi removido, UpdateReferenceCounters chama RecDel (Algoritmo 4.5), que realiza a remoção recursiva de referências. Assim como no algoritmo para um mutador e um coletor, RecDel garante que apenas objetos que não estão mais vivos serão efetivamente removidos para o espaço livre. Isso garante a propriedade da segurança.

Vivacidade Como mencionado, UpdateReferenceCounters chama RecDel para contabilizar a remoção de referências a um objeto. Por sua vez, RecDel detecta quando a remoção da referência ao objeto tiver o potencial de isolar um ciclo – verificando se o contador de referência tem valor maior que um; nesse caso, o objeto é adicionado ao *status analyser*, que fará a verificação de ciclos assim como nas versões anteriores do algoritmo. O processo todo de varredura de ciclos locais não é diferente dos algoritmos previamente examinados, e desta forma a propriedade da vivacidade está garantida.

capítulo 7

Conclusões e Considerações Finais

Este trabalho apresentou um estudo realizado sobre algoritmos de contagem de referências cíclicas adequados para uso em sistemas multiprocessados, detalhando os algoritmos em si, implementações dos mesmos foram realizadas em plataformas de testes, e resultados de testes de desempenho realizados.

Dois novos algoritmos foram propostos: um algoritmo para uma arquitetura baseada em um mutador e um coletor, baseado no conceito de fila de atualizações (Seção 4.3.2), e um algoritmo adequado para arquiteturas com vários mutadores e um coletor, usando também o conceito de fila de atualizações (Capítulo 5). Esses novos algoritmos foram implementados e testados em conjuntos de programas de *benchmarking* para análises de desempenho. Os resultados mostrados são promissores: ambos os algoritmos têm desempenho competitivo em relação às alternativas disponíveis e, no caso do algoritmo para vários mutadores, apresenta tempos de pausa muito menores do que as alternativas.

Os novos algoritmos apresentados nos Capítulos 4 e 5 respondem positivamente às Questões de Pesquisa 1 e 2 apresentadas na Seção 1.3.1. Em especial, com relação à Questão 2, o uso da fila de atualizações não só reduz a necessidade de sincronização entre coletor e mutador, como também reduz o número de atualizações necessárias nos contadores de referências dos objetos, resultando em uma melhora considerável no desempenho do algoritmo.

O algoritmo do Capítulo 5 se presta bem à implementação em arquiteturas atuais baseadas em máquinas virtuais como a Java Virtual Machine ou sistemas de tempo de execução geren-

ciados como o Common Language Runtime (CLR) da plataforma .NET da Microsoft. Em relação a algoritmos de coleta de lixo usados nessas plataformas, o algoritmo do Capítulo 5 tem desempenho menos de 10% pior em todos os testes realizados, mas tem a grande vantagem de impor tempos de pausa menores ao programa executado; de fato, os tempos de pausa no algoritmo de contagem de referências cíclicas são ordens de magnitude menores que os dos algoritmos concorrentes. Isso é uma evidência da resposta positiva dada à Questão de Pesquisa 1 da Seção 1.3.1. Os testes realizados com os programas do conjunto DaCapo, que são baseados em aplicações reais escritas em Java, respondem à Questão de Pesquisa 3 de maneira satisfatória.

Também foi feito um conjunto de testes preliminares para determinar o comportamento do algoritmo proposto para vários mutadores com relação ao crescimento do número de processadores disponíveis, como pode ser visto na Seção 5.7.5. Os testes realizados com um computador com processador de 4 núcleos mostram que o algoritmo proposto para vários mutadores continua tendo bom desempenho tanto em *throughput* como em relação à latência. Isso responde parcialmente à Questão de Pesquisa 4, mas mais investigação é necessária para observar tendências que apareçam quando o número de processadores cresce ainda mais. Essa ideia e outras relacionadas podem ser exploradas em trabalhos futuros, como será detalhado adiante.

Esta característica do algoritmo proposto no Capítulo 5 pode ser usada com vantagem em situações em que a latência e o tempo de resposta do programa são tão ou mais importantes que o desempenho, por exemplo:

- ▷ Sistemas de tempo real, nos quais o programa deve responder às solicitações dentro de prazos fixos. Algoritmos de coleta de lixo que impõem pausas maiores podem tornar imprevisível o tempo de resposta do sistema, pois um ciclo de coleta pode começar enquanto existe uma ou mais solicitações pendentes.
- ▷ Sistemas de interface com o usuário final. Usuários tendem a preferir sistemas que respondam aos seus comandos, mesmo que o tempo total para realizar cada comando seja um pouco maior.
- ▷ Servidores ou Serviços Web que devem responder às solicitações dentro de prazos estipulados por algum contrato de serviço.

Em aplicações onde o tempo total de processamento é mais importante, como em aplicações científicas, pode ser mais desejável usar métodos alternativos à contagem de referências.

7.1 Sugestões para Trabalhos Futuros

O trabalho de investigação relatado aqui suscita uma grande possibilidade de outros estudos relacionados que podem ser realizados. Alguns exemplos são sugeridos a seguir:

- ▷ As filas usadas pelos algoritmos baseados em filas de atualização têm um impacto importante no desempenho resultante. Existe um ferramental teórico baseado em *teoria das filas* que poderia ser usado para analisar matematicamente algumas características do desempenho dessas filas, e as informações dessa análise poderiam ser usadas para determinar a relação entre alguns parâmetros associados (por exemplo, tamanho máximo da fila) e o desempenho final do algoritmo.
- ▷ O maior ponto de contenção no algoritmo para vários mutadores do Capítulo 5 é o acesso à lista de espaço livre na memória: todos os *threads* acessam a mesma lista. Uma possível alteração que poderia reduzir a contenção e melhorar o desempenho do algoritmo seria ter listas de espaço livre separadas para cada *thread* ou grupo de *threads*. Seria necessário estudar qual a melhor solução para casos como um *thread* usando todo o espaço de memória na sua lista livre, enquanto as listas de outros *threads* possuem espaço disponível. Seria necessário testar se a alteração compensa a maior complexidade com uma melhoria significativa no desempenho do algoritmo.
- ▷ O algoritmo para vários mutadores (Capítulo 5) suspende cada mutador por algum tempo enquanto copia sua fila de atualizações. Uma possibilidade para reduzir as pausas ainda mais seria usar uma estrutura diferente para a fila de atualização. Um exemplo é usar duas filas para cada *thread*, filas F_1 e F_2. Apenas uma das filas estaria ativa em cada momento. Inicialmente, pode-se usar a fila F_1 como ativa. Durante o início do ciclo de coleta, em que o coletor precisa obter o conteúdo das filas de atualização de cada mutador, a fila ativa para o mutador passa a ser F_2, enquanto o conteúdo de F_1 é utilizado pelo coletor; registros de atualização seriam incluídos apenas na fila ativa. Essas alterações, se não criarem problemas de sincronização ou interferência entre coletor e mutadores, talvez reduzam ou mesmo eliminem as pausas impostas pelo coletor.
- ▷ Os algoritmos testados neste trabalho usam apenas um processo ou *thread* coletor, o que é adequado para um computador com poucos processadores. Entretanto, à medida que o número de núcleos de processamento cresce, provavelmente será mais

vantajoso ter vários coletores funcionando em paralelo. Algumas possibilidades para arquiteturas com mais de um coletor foram sugeridas no Capítulo 5, mas nenhum algoritmo seguindo essas possibilidades foi implementado. Uma sugestão de trabalho é desenvolver e implementar um algoritmo com vários coletores em alguma plataforma de *hardware* com um maior número de processadores, ou através de simulação, e verificar o desempenho das soluções propostas.

- Os algoritmos propostos nos Capítulos 4 e 5 podem ser adaptados para uso com sistemas distribuídos de memória compartilhada. Como não há a necessidade de varrer todo o espaço de memória compartilhada, como ocorre nos algoritmos de coleta por cópia ou coleta por marcação e varredura, os algoritmos que usam contagem de referência tendem a ter melhor desempenho em arquiteturas distribuídas. Embora a sincronização envolvida seja mais complexa, os algoritmos funcionam com relativamente poucas operações de sincronização explícita. No algoritmo para vários mutadores do Capítulo 5, as filas de atualização já são obtidas pelo coletor de uma forma similar a um *snapshot* distribuído.

Referências

[1] B. Alpern and *et al.*, "The jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–215, Março 2000.

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. AFIPS Spring Joint Computer Conference*, Atlantic City, NJ, USA, Abril 1967, pp. 483–485.

[3] G. R. Andrews, *Concurrent Programming: Principles and Practice.* Addison-Wesley Professional, Julho 1991.

[4] ——, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley Professional, Julho 1999.

[5] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language (4th Edition).* Addison-Wesley Professional, Agosto 2005.

[6] ArsTechnica, "Multicore, dual-core, and the future of intel," 2004. [Online]. Available: http://arstechnica.com/articles/paedia/cpu/intel-future.ars/1

[7] Azatchi, Hezi, Levanoni, Yossi, Paz, Harel, and Petrank, Erez, "An on-the-fly mark and sweep garbage collector based on sliding views," in *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications.* New York, NY, USA: ACM, 2003, pp. 269–281. [Online]. Available: http://dx.doi.org/10.1145/949305.949329

[8] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The fortran automatic coding system," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, ser. IRE-AIEE-ACM '57 (Western). New York, NY, USA: ACM, 1957, pp. 188–198.

[9] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith, "Java without the coffee breaks: a nonintrusive multiprocessor garbage collector," in *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation.* New York, NY, USA: ACM, 2001, pp. 92–103.

[10] N. Benton, "Abstracting allocation: The new new thing," in *Computer Science Logic (CSL 2006)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2006.

[11] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications.* New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[12] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake up and smell the coffee: evaluation methodology for the 21st century," *Commun. ACM*, vol. 51, pp. 83–89, August 2008.

[13] S. Blackburn, R. Garner, and D. Frampton, "Mmtk: The memory management toolkit," JikesRVM, Tech. Rep., 2006. [Online]. Available: http://cs.anu.edu.au/~Robin.Garner/mmtk-guide.pdf

[14] S. Blazy and X. Leroy, "Formal verification of a memory model for C-like imperative languages," in *International Conference on Formal Engineering Methods (ICFEM 2005)*, ser. Lecture Notes in Computer Science, vol. 3785. Springer-Verlag, 2005, pp. 280–299.

[15] H. Boehm, "Destructors, finalizers, and synchronization," in *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, vol. 38, no. 1. New York, NY, USA: ACM Press, Janeiro 2003, pp. 262–272.

[16] ——, "The space cost of lazy reference counting," in *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, vol. 39, no. 1. New York, NY, USA: ACM Press, Janeiro 2004, pp. 210–219.

[17] H. Boehm and A. Demers, "Garbage collection in an uncooperative environment," *Software Practice & Experience*, vol. 9, no. 18, pp. 195–244, 1988.

[18] E. Chailloux, P. Manoury, and B. Pagano, *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[19] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, pp. 677–678, November 1970.

[20] J. Chirimar, C. A. Gunter, and J. G. Riecke, "Reference counting as a computational interpretation of linear logic," *Journal of Functional Programming*, vol. 6, no. 2, pp. 195–244, 1996. [Online]. Available: http://citeseer.ist.psu.edu/chirimar96reference.html

[21] W. F. Clocksin and C. S. Mellish, *Programming in Prolog, 5th edition*. Springer, 2003.

[22] G. E. Collins, "A method for overlapping and erasure of lists," *Communications of the ACM*, vol. 3, no. 12, pp. 655–657, Dezembro 1960.

[23] D. Coward, "Java vm: Trying a new garbage collector for jvm 7," 2008. [Online]. Available: http://blogs.sun.com/theplanetarium/entry/java_vm_trying_a_new

[24] M. J. Crawley, *The R Book*. Wiley, 2007.

[25] D. L. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," pp. 37–48, 2004. [Online]. Available: http://dx.doi.org/10.1145/1029873.1029879

[26] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, "Lock-free reference counting," *Distributed Computing*, vol. 15, no. 4, pp. 255–271, Dezembro 2002.

[27] J. DeTreville, "Experience with concurrent garbage collectors for Modula-2+," DECSRC, Tech. Rep. 64, Agosto 1990.

[28] P. L. Deutsch and D. G. Bobrow, "An efficient, incremental, automatic garbage collector," *Commun. ACM*, vol. 19, no. 9, pp. 522–526, Setembro 1976.

[29] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation," *Commun. ACM*, vol. 21, no. 11, pp. 966–975, Novembro 1978.

[30] D. Doligez and G. Gonthier, "Portable, unobtrusive garbage collection for multiprocessor systems," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[31] D. Doligez and X. Leroy, "A concurrent, generational garbage collector for a multi-threaded implementation of ML," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[32] R. K. Dybvig, *The Scheme Programming Language, 3rd. edition.* MIT Press, 2003.

[33] R. R. Fenichel and J. C. Yochelson, "A lisp garbage-collector for virtual-memory computer systems," *Commun. ACM*, vol. 12, pp. 611–612, November 1969.

[34] A. J. Field and P. Harrison, *Functional Programming.* Addison-Wesley, Julho 1988.

[35] J. A. S. Filho, "Algoritmos para contagem de referências cíclicas," Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Fevereiro 2002.

[36] D. Flanagan, *JavaScript: The Definitive Guide, 5th edition.* O'Reilly, 2006.

[37] C. Flood, D. Detlefs, N. Shavit, and C. Zhang, "Parallel garbage collection for shared memory multiprocessors," in *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, 2001.

[38] W. Fokkink, *Introduction to Process Algebra.* Springer, Fevereiro 2000.

[39] A. Formiga and R. D. Lins, "A New Architecture for Concurrent Lazy Cyclic Reference Counting onMulti-Processor Systems," *Journal of Universal Computer Science*, vol. 13, no. 6, 2007.

[40] ——, "Efficient removal of noisy borders of monochromatic documents," in *Image Analysis and Recognition*, ser. Lecture Notes in Computer Science, M. Kamel and A. Campilho, Eds. Springer Berlin / Heidelberg, 2009, vol. 5627, pp. 158–167.

[41] E. Fuentes and E. Eilebrecht, "Thread Management in the CLR," *MSDN Magazine*, no. 6, 2008. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd252943.aspx

[42] J. Y. Girard, "Linear logic: Its syntax and semantics," in *Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University, June 1993)*, J. Y. Girard, Y. Lafont, and L. Regnier, Eds., no. 222. Cambridge University Press, 1995.

[43] B. Goetz, "Java theory and practice: Garbage collection in the hotspot jvm," *IBM developerWorks*, 2003. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-jtp11253/index.html

[44] W. Gruener, "Intel aims for 32 cores by 2010," *TG Daily*, 2006. [Online]. Available: http://www.tgdaily.com/2006/07/10/intel_32_core_processor/

[45] S. P. Harbison, *Modula-3.* Prentice-Hall, 1992.

[46] A. Hejlsberg, M. Torgensen, S. Wiltamuth, and P. Golde, *The C# Programming Language, 4th edition.* Addison-Wesley Professional, 2010.

[47] A. Hejlsberg, S. Wiltamuth, and P. Golde, *C# Programming Language, The (2nd Edition) (Microsoft .Net Development Series).* Addison-Wesley Professional, Junho 2006.

[48] M. P. Herlihy and J. E. B. Moss, "Lock-free garbage collection for multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 3, pp. 304–311, 1992. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=139204

[49] M. P. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, Dezembro 2008.

[50] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, Novembro 2000.

[51] R. Jain, *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[52] K. Jensen, N. Wirth, A. B. Mickel, and J. F. Miner, *Pascal User Manual and Report: ISO Pascal Standard.* Springer, Setembro 1991.

[53] R. Jones and R. Lins, *Garbage Collection : Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Setembro 1996.

[54] S. P. Jones and J. H. (editors), Tech. Rep.

[55] B. W. Kernighan and D. M. Ritchie, *C A Linguagem de Programação, Padrão ANSI.* Campus, 1989.

[56] D. E. Knuth, *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition).* Addison-Wesley Professional, Julho 1997.

[57] S. G. Kochan, *Programming in Objective-C 2.0, 2nd edition.* Addison-Wesley Professional, 2009.

[58] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional, Julho 2002.

[59] ——, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2003.

[60] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," 2001. [Online]. Available: citeseer.ist.psu.edu/article/leijen01parsec.html

[61] Y. Levanoni and E. Petrank, "An on-the-fly reference-counting garbage collector for java," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 1, pp. 1–69, Janeiro 2006. [Online]. Available: http://dx.doi.org/10.1145/1111596.1111597

[62] I. Limited, *Occam Programming Manual.* Prentice Hall, 1984.

[63] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification, 2nd. edition.* Prentice-Hall, 1999.

[64] R. D. Lins, "Lazy cyclic reference counting," *Journal of Universal Computer Science*, vol. 9, no. 8, pp. 813–828, Agosto 2003.

[65] R. Lins, B. Ávila, and A. Formiga, "Bigbatch – an environment for processing monochromatic documents," in *Image Analysis and Recognition*, ser. Lecture Notes in Computer Science, A. Campilho and M. Kamel, Eds. Springer Berlin / Heidelberg, 2006, vol. 4142, pp. 886–896. [Online]. Available: http://dx.doi.org/10.1007/11867661_80

[66] R. D. Lins, "A shared memory architecture for parallel cyclic reference counting," *Microprocessing and Microprogramming*, vol. 34, pp. 31–35, Setembro 1991.

[67] ——, "Cyclic reference counting with lazy mark-scan," *Inf. Process. Lett.*, vol. 44, no. 4, pp. 215–220, Dezembro 1992.

[68] ——, "A multi-processor shared memory architecture for parallel cyclic reference counting," *Microprocessing and Microprogramming*, vol. 35, pp. 563–568, 1992.

[69] ——, "An efficient algorithm for cyclic reference counting," *Inf. Process. Lett.*, vol. 83, no. 3, pp. 145–150, Agosto 2002.

[70] ——, "A new multi-processor architecture for parallel lazy cyclic reference counting," in *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 35–43.

[71] ——, "Cyclic reference counting," *Inf. Process. Lett.*, vol. 109, no. 8, pp. 71–78, Dezembro 2008.

[72] A. D. Martínez, R. Wachenchauzer, and R. D. Lins, "Cyclic reference counting with local mark-scan," *Inf. Process. Lett.*, vol. 34, no. 1, pp. 31–35, Fevereiro 1990.

[73] Y. Matsumoto, *Ruby In A Nutshell*. O'Reilly Media, Inc., Novembro 2001.

[74] G. Mattos, R. D. Lins, A. Formiga, and F. M. Junqueira Martins, "Bigbatch: a document processing platform for clusters and grids," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 434–441.

[75] G. d. O. Mattos, A. Formiga, R. D. Lins, F. de Carvalho Júnior, and F. J. Martins, "Comparative aspects between the cluster and grid implementations of bigbatch," *Journal of Universal Computer Science*, vol. 14, no. 18, pp. 3031–3050, 2008.

[76] G. d. O. Mattos, R. D. Lins, A. Formiga, and F. H. d. Carvalho Jr., "A comparison of cluster and grid configurations executing image processing tasks in a local network," *International Conference on Networking*, vol. 0, pp. 408–414, 2008.

[77] H. J. Mcbeth, "Letters to the editor: on the reference counter method," *Commun. ACM*, vol. 6, no. 9, Setembro 1963.

[78] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Communications of the ACM*, vol. 3, no. 3, pp. 184–195, Março 1960.

[79] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, Dezembro 1978.

[80] ——, *Communication and Concurrency*. Prentice Hall, Inc., Dezembro 1989.

[81] ——, *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, Junho 1999.

[82] M. Minsky, "A lisp garbage collector algorithm using serial secondary storage," Cambridge, MA, USA, Tech. Rep., 1963.

[83] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=658762

[84] G. C. Necula, "Proof-carrying code," in *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, Janeiro 1997, pp. 106–119. [Online]. Available: http://citeseer.ist.psu.edu/49901.html

[85] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 1996, pp. 2–11.

[86] W. Partain, "The nofib benchmark suite of haskell programs," in *Workshops in Computing – Proceedings of the 1992 Glasgow Workshop on Functional Programming*. London, UK: Springer Verlag, 1993, pp. 195–202.

[87] D. A. Patterson and J. L. Hennessy, *Organização e Projeto de Computadores: A Interface Hardware/Software, 3 ed.* Editora Campus, 2005.

[88] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Inc., Junho 1981.

[89] S. L. Peyton-Jones, *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Series)*. Prentice Hall, Maio 1987.

[90] G. L. Steele, "Multiprocessing compactifying garbage collection," *Commun. ACM*, vol. 18, no. 9, pp. 495–508, Setembro 1975.

[91] H. Sutter and J. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, Setembro 2005. [Online]. Available: http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=332

[92] A. S. Tanenbaum, *Modern Operating Systems, 3rd. ed.* Prentice Hall, Dezembro 2007.

[93] S. Thompson, *Haskell: The Craft of Functional Programming, 2 ed.* Addison-Wesley, Março 1999.

[94] J. D. Valois, "Implementing lock-free queues," in *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, Outubro 1994, pp. 64–69. [Online]. Available: citeseer.ist.psu.edu/valois94implementing.html

[95] G. Van Rossum, *The Python Language Reference Manual.* Network Theory Ltd., Setembro 2003.

[96] L. Wasserman, *All of Statistics: A Concise Course in Statistical Inference.* Springer, Dezembro 2003.

[97] J. Weizenbaum, "Symmetric list processor," *Commun. ACM*, vol. 6, no. 9, pp. 524–536, Setembro 1963.

APÊNDICE A

Plataforma de Testes Para Algoritmos com Um Mutador

Para realizar os testes da arquitetura descrita no Capítulo 4, foi implementado um compilador para uma linguagem funcional *lazy* simples, para que fosse possível ter todo o controle sobre a implementação do coletor de lixo de maneira concorrente. Vários compiladores e sistemas de tempo de execução atuais não são preparados para aproveitarem múltiplos processadores, se presentes. Isto motivou a criação de uma implementação completa, do compilador ao sistema de tempo de execução, para aproveitar os sistemas multiprocessados.

A linguagem implementada segue o paradigma funcional e é similar a uma versão simplificada da linguagem Haskell [54, 93]. O compilador analisa a sintaxe e verifica os tipos do programa, depois realiza o processo de *lambda-lifting* nas funções do programa, traduzindo o resultado para código da máquina G; este código é então traduzido para código nativo da arquitetura Intel de 32 bits. O resultado final é ligado com o sistema de tempo de execução e o coletor de lixo, que foram escritos em C. As técnicas de compilação utilizadas são conhecidas e podem ser encontradas, por exemplo, no livro de Peyton-Jones [89].

Para a implementação das *threads* concorrentes, foi utilizada a biblioteca *POSIX Threads*, cujas versões mais recentes no sistema operacional Linux aproveitam e utilizam vários processadores, se presentes no computador. Com isto, fica garantido que *threads* diferentes podem ser agendadas para processadores diferentes, dependendo da disponibilidade dos mesmos.

As seções seguintes descrevem a implementação em maiores detalhes.

Módulo	Descrição
AbsSyntax	Definições para a representação em sintaxe abstrata dos programas de entrada
Compiler	Módulo principal do compilador
ELambda	Definições e funções para representação do programa como uma expressão em um λ-*calculus* extendido
Environ	Funções para usar e manipular ambientes (*environments*)
GCode	Tradução do programa para o código de entrada da máquina G (*código G*)
GPrims	Funções primitivas da linguagem implementadas em código G
Parser	Analisador sintático
Primitives	Descrições de alto nível das funções primitivas da linguagem
SuperComb	*Lambda-lifting* e tradução do programa para super-combinadores
Type	Verificação e reconstrução de tipos, seguindo o sistema de tipos Hindley-Milner
Util	Funções utilitárias usadas em vários outros módulos
X86	Gerador de código para a arquitetura Intel x86

Tabela A.1: *Módulos do compilador e suas descrições*

A.1 Sintaxe

A Figura A.1 mostra a sintaxe da linguagem de teste utilizada, que é a linguagem de entrada do compilador implementado. Pode-se perceber que a sintaxe é similar à da linguagem Haskell; entretanto, decidiu-se usar ponto-e-vírgula como terminador de linhas obrigatório, para evitar a implementação de uma regra de *layout* complexa (vide definição da linguagem Haskell [54]).

A.2 Compilador

O compilador para a linguagem de teste foi escrito em Haskell, linguagem que se mostrou bastante adequada para a tarefa: o compilador completo tem pouco mais de duas mil linhas de código, incluindo o analisador sintático. Ao total, o compilador é composto por 12 módulos, como mostrado na Tabela A.1.

O módulo principal, *Compiler*, contém a interface de linha de comando e realiza a coordenação entre os outros módulos para gerar o código final. As opções na linha de comando determinam que etapas da compilação serão realizadas, ou seja, que estágios do compilador serão utilizados. Por exemplo, é possível fazer com que o compilador apenas faça a análise sintática, ou apenas a verificação e reconstrução de tipos, ou chegue até a geração do código G, e assim em diante. A Figura A.2 mostra a função principal do compilador; seu funcionamento

```
# Syntaxe (em EBNF)

# id e typevar são identificadores de variáveis, começando com uma letra minúscula
# typeid é um identificador de construtor de tipo
# consid é um identificador de construtor de dados
# tanto typeid quanto consid devem começar com uma letra maiúscula

<program> := <decl>*

<decl> := [<typeattr> ;] <valdecl> ;    # declaração de um valor
        | <typedecl> ;                  # declaração de um tipo

<typeattr>  := id :: <type>

<type> := typeid <type>*                        # tipos construídos
       | [<type>]                               # tipos de lista
       | <type> -> <type>                       # tipos de função
       | (<type>{,<type>}*)                     # tipos produto (tuplas)
       | typevar | Int | Bool

<typedecl> := data typeid typevar* = <variant> { | <variant> }*
            | type typeid typevar* = <type> | typevar

<variant> := consid <type>*

<valdecl> := id <pat>* = <exp>

<pat> := id

<exp> := id
       | <integer> | <boolean> | <exp> + <exp> | <exp> * <exp> | <exp> - <exp>
       | <exp> / <exp> | - <exp> | <exp> && <exp> | <exp> || <exp> | not <exp>
       | <exp> = <exp> | <exp> /= <exp> | <exp> > <exp> | <exp> < <exp>
       | <exp> >= <exp> | <exp> <= <exp> | (<exp>)
       | consid <exp>*                  # construtor arbitrário
       | <exp> : <exp>                  # construção de lista
       | if <exp> then <exp> else <exp> # condicional
       | let <pat> = <exp> in <exp>     # expressão qualificada
       | let id <pat>* = <exp> in <exp>
       | <exp> <exp>* # aplicação
       | (<exp>{,<exp>}+)               # tupla
       | [<exp>{,<exp>}*]               # lista

<boolean>   := True
             | False
```

Figura A.1: *Sintaxe para a linguagem de entrada do compilador em notação BNF*

consiste em obter os argumentos e opções da linha de comando e determinar que estágio do compilador deve ser chamado, em seguida chamando a função `compile` para executar o estágio selecionado.

```
main = do args <- getArgs
          prog <- getProgName
          (o, n) <- getOptions args prog
          (stage, o') <- compileStage o
          if null n then putStrLn (usage prog)
           else compile stage o' (head n) prog
```

Figura A.2: *Função principal do compilador*

A função `compile` está mostrada na Figura A.3, que consiste em realizar a análise sintática no programa (através da função `parse`), seguida pela verificação e reconstrução de tipos (função `typeCheck`) e, ao final, chamando a função `dispatch` para encaminhar o programa para o estágio selecionado do compilador.

```
compile :: CompilerStage -> [Flag] -> String -> String -> IO ()
compile stage opts file prog = do e <- parse file
                                  t <- typeCheck e opts
                                  o <- getOutput opts
                                  dispatch stage e t o
                                  hClose o
```

Figura A.3: *Função `compile`*

O analisador sintático foi construído utilizando a biblioteca *Parsec*, que implementa combinadores monádicos de analisadores sintáticos [60]. A verificação e reconstrução de tipos utiliza o algoritmo de reconstrução de Milner [79].

A função `dispatch` chama o estágio adequado do compilador, segundo as opções utilizadas pelo usuário. A definição da função `dispatch` é mostrada na Figura A.4, juntamente com algumas funções auxiliares. Como o objetivo final do compilador é gerar código *assembly* para que seja compilado para linguagem de máquina, é este estágio final que será descrito.

A entrada da função `dispatch` é composta pela representação do programa em sintaxe abstrata (variável `e`), o tipo determinado para o programa (variável `t`) e um fluxo de saída, que pode ser um arquivo ou o console (variável `out`). O último caso para a função `dispatch` trata o estágio `Assembly`, cujo objetivo é gerar código *assembly* para o programa. Vê-se facilmente

```
intForm e = ELambda.translateProg e
superclift i = SuperComb.translate i
supercomb i = SuperComb.translate i
gcode s = GCode.translate s
asm gi = X86.translate (GPrims.primitiveCode ++ gi)

dispatch TypeAnalysis e t out = hPutStrLn out (showType t)
dispatch IntermForm e t out = hPutStrLn out (ELambda.prettyPrint $ intForm e)
dispatch SuperComb e t out = let sc = superclift $ intForm e in
                               hPutStrLn out (SuperComb.printSCProg sc)
dispatch G_Code e t out = let is = gcode $ supercomb $ intForm e in
                          hPutStrLn out (GCode.printInstructions is)
dispatch Assembly e t out = let a = asm $ gcode $ supercomb $ intForm e in
                            hPutStrLn out (X86.printInstructions a)
```

Figura A.4: *Função* `dispatch`

que o seguinte processo é seguido nesse caso:

1. A representação em sintaxe abstrata é traduzida para uma forma intermediária baseada em um λ-*calculus* extendido;
2. Esta forma intermediária passa pelo processo de *lambda-lifting*, gerando um conjunto de super-combinadores;
3. Os super-combinadores resultantes são traduzidos para código G, o código de entrada da máquina G;
4. O código G é traduzido para código *assembly* para processadores Intel da arquitetura x86.

O código *assembly* final é compatível com o programa AS, o *assembler* do projeto GNU.

A.3 Suporte de tempo de execução

O suporte de tempo de execução para os programas compilados consiste basicamente de duas partes:

1. A máquina de redução de grafos
2. O coletor de lixo

A máquina de redução de grafos é uma implementação da máquina G, e suas operações estão incluídas no código *assembly* gerado ao final da compilação. O coletor de lixo, por sua vez, foi escrito em linguagem C de forma a cooperar com o programa assembly gerado pelo compilador.

A.3.1 Formato das células

Os objetos manipulados pela máquina G são células de tamanho fixo. O coletor de lixo utiliza esse fato para realizar o gerenciamento da memória, como foi mencionado nos Capítulos 2 e 3. A Figura A.5 mostra graficamente a estrutura dessas células, e a Figura A.6 mostra a mesma estrutura como expressa no programa em linguagem C. A célula é formada por quatro campos de 32 bits cada:

- O campo *gc* é reservado para o uso do coletor de lixo
- O campo *tag* é utilizado como um identificador do tipo do objeto que ocupa a célula
- Os campos *field1* e *field2* têm significados diferentes dependendo do tipo do objeto que ocupa a célula, ou seja, as funções destes campos dependem do valor do campo *tag*

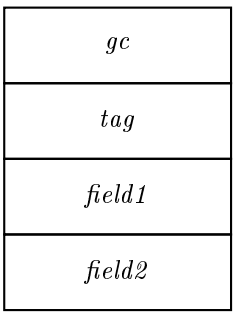

Figura A.5: *Formato das células*

A reserva de um campo de 32 bits para o coletor de lixo permite integrar ao código gerado pelo compilador coletores utilizando qualquer uma das técnicas mostradas no Capítulo 2. O campo *tag* é utilizado não só para identificar o tipo do objeto, como também para apontar para uma *tabela de despacho*, que contém os endereços de versões específicas para o tipo em questão de rotinas genéricas na linguagem. Este uso é similar à implementação original da máquina G – vide Capítulo 19 do livro de Peyton-Jones [89]. Os tipos disponíveis na implementação – cada um possui uma *tag* correspondente – são:

```
typedef struct tagCell
{
  unsigned gc;
  unsigned tag;
  void *   field1;
  void *   field2;
} Cell;
```

Figura A.6: *Declaração da estrutura das células em C*

- ▷ inteiro
- ▷ valor *booleano*
- ▷ função
- ▷ aplicação
- ▷ construção de lista (célula *cons*)

A.3.2 Organização

O suporte de tempo de execução realiza a inicialização do programa como um todo, chamando a função de inicialização do coletor de lixo e iniciando a execução do código gerado pelo compilador. A função principal do sistema de suporte é mostrada na Figura A.7. As funções `Initialize` e `CleanUp` são parte do coletor de lixo, enquanto que a função `faul_main` é declarada pelo código gerado pelo compilador. Além da função principal, o suporte de execução inclui algumas funções auxiliares necessárias durante a execução do programa, como algumas rotinas de impressão simples.

```
int main(void)
{
  Initialize();

  faul_main(stack_end);

  CleanUp();
}
```

Figura A.7: *Função principal do suporte de tempo de execução*

A.3.3 Coletor de lixo

Outra função do suporte de tempo de execução é o gerenciamento da memória, que é realizado pelo coletor de lixo. Para possibilitar que vários coletores diferentes pudessem ser integrados ao programa, foi definida uma interface simples que deve ser implementada por qualquer coletor. Esta interface é mostrada na Figura A.8.

```
int main(void)
void Initialize(void);
Cell * New();
void Update(void *ptr, Cell *c);
void Delete(Cell *c);
void CleanUp(void);
```

Figura A.8: *Interface para o coletor de lixo*

Foram implementadas quatro versões do coletor de lixo:

1. Um coletor que não faz nenhuma tarefa de gerenciamento da memória; na verdade, isto significa que não há coletor de lixo
2. Um coletor baseado na contagem de referências simples, como mostrada no Capítulo 2; este coletor não recupera ciclos de células inativas
3. Um coletor baseado na contagem de referências cíclicas, seguindo a especificação mostrada no Capítulo 3
4. Um coletor concorrente que executa em um processo separado do único processo mutador, correspondendo à versão da arquitetura com um coletor e um mutador do Capítulo 4

A função `Initialize` para o coletor concorrente cria uma nova *thread* para o coletor chamando a função `pthread_create` da biblioteca *POSIX Threads* ou *pthreads*. As pilhas de incremento e decremento são implementadas como listas duplamente encadeadas, com mutador e coletor tendo acesso a pontas opostas de cada uma, como explicado no Capítulo 4. Embora não seja necessário utilizar sincronização explícita nessa arquitetura para as operações do coletor, a Figura A.9 mostra a implementação de uma operação atômica do tipo *compare-and-swap* para a arquitura Intel x86. Esta função pode ser extendida para trabalhar com outras arquiteturas sem maiores dificuldades, já que todas as arquiteturas atuais possuem alguma operação do tipo *compare-and-swap* no seu conjunto de instruções.

```
bool_t CompareAndSwap(IN void ** ptr, IN void * old, IN void * new)
{
   unsigned char ret;

   __asm__ __volatile__
   (
      "  lock\n"
      "  cmpxchgl %2,%1\n"
      "  sete %0\n"
      : "=q" (ret), "=m" (*ptr)
      : "r" (new), "m" (*ptr), "a" (old)
      : "memory"
   );

   return ret;
}
```

Figura A.9: *Implementação da operação* compare-and-swap

APÊNDICE B

PROGRAMAS DE TESTE

Aqui são apresentados e descritos os programas utilizados para testar o desempenho dos algoritmos propostos para um coletor e um mutador, mostrados no Capítulo 4. Os resultados dos testes realizados são mostrados no mesmo capítulo. Os programas foram escritos na linguagem descrita no Apêndice A.

B.1 Função de Ackermann

Este programa implementa a *função de Ackermann*, um exemplo de função computável que não apresenta recursão primitiva. A função recebe dois números naturais como entrada e produz outro número natural. O texto do programa é mostrado na Figura B.1.

```
-- Funcao de Ackermann: acker
acker m n = if m == 0 then (n + 1) else
            if m > 0 && n == 0 then acker (m - 1) 1 else
            acker (m - 1) (acker m (n - 1));

main = acker 3 4;
```

Figura B.1: *Programa* acker

Este programa é interessante pois gera muitas chamadas recursivas, o que testa a capacidade de recuperar ciclos. Por outro lado, o número de chamadas cresce muito rapidamente, o que provoca um estouro na pilha do programa para valores de m tão pequenos quanto 5. Muitas implementações de linguagens de programação, quando usam uma pilha de tamanho

padrão, também encontram erros ao tentar calcular valores maiores desta função.

B.2 Concatenação de listas

O programa na Figura B.2 chama repetidamente uma função para concatenar listas, aplicando várias funções sobre seus componentes. Ao utilizar recursividade e listas, geram-se várias referências a células que devem ser gerenciadas pelo coletor de lixo.

```
-- conctwice
conc l1 l2 = if null l1 then l2 else (head l1) : (conc (tail l1) l2);
map f l = if null l then [] else (f (head l)) : (map f (tail l));
twice f x = f (f x);
sq x = x * x;
succ x = x + 1;
pred x = x - 1;
fib n = if n < 2 then 1 else (fib (n - 1)) + (fib (n - 2));
main = conc (conc (conc (map fib [10, 11, 12, 13, 14]) (map sq [4, 5, 6, 7]))
                  (map (twice succ) [2, 3, 4, 5]))
    (map (twice pred) [2, 3, 4, 5]);
```

Figura B.2: *Programa* conctwice

B.3 Números de Fibonacci

Este programa calcula vários números da seqüência de Fibonacci, utilizando uma função recursiva. As duas chamadas recursivas na função `fib` tornam sua complexidade exponencial, gerando um grande número de ciclos no grafo da memória, e exigindo bastante do coletor de lixo. O programa é mostrado na Figura B.3.

```
-- fiblista
map f l = if null l then [] else (f (head l)) : (map f (tail l));
fib n = if n < 2 then 1 else (fib (n - 1)) + (fib (n - 2));
main = map fib [16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26];
```

Figura B.3: *Programa* fiblista

B.4 Cálculo do fatorial

Este programa cria um padrão de chamadas recursivas similar ao do programa para cálculo do número de Fibonacci, mas usando o cálculo do fatorial. Neste caso, os números crescem muito mais rapidamente, causando um estouro da capacidade de palavras de 32 bits. Por esta razão o programa apenas repete o cálculo para um mesmo argumento, como mostrado na Figura B.4.

```
-- recfat
map f l = if null l then [] else (f (head l)) : (map f (tail l));
fat n = if n < 2 then 1 else (n * (fat (n - 1)));
recfat n = if n <= 0 then (fat 0) else (fat n) + (recfat (n - 1)) + (recfat (n - 2));
run f x n = if n == 0 then [] else (f x) : (run f x (n - 1));
main = run recfat 12 12;
```

Figura B.4: *Programa* recfat

B.5 Somatório recursivo

Um programa para calcular recursivamente o somatório de números variando de m até n, com incrementos de 1. A Figura B.5 lista todo o texto do programa, que aplica a função de somatório a vários pares de números.

```
-- somatorio
map f l = if null l then [] else (f (head l)) : (map f (tail l));
somatorio m n = if n == m then n else (m + (somatorio (m + 1) n));
main = map (somatorio 1) [150, 160, 170, 180, 190, 200, 210, 220, 230, 240];
```

Figura B.5: *Programa* somatorio

B.6 Somatório de listas

Este programa calcula os somatórios S_n de todos os números entre 1 e n, para sucessivos valores de n. O texto se encontra na Figura B.6.

```
-- somamap
map f l = if null l then [] else (f (head l)) : (map f (tail l));
gl l x = if x == 0 then l else (gl (x : l) (x - 1));
somatorio m n = if n == m then n else (m + (somatorio (m + 1) n));
main = map (somatorio 1) (gl [] 100);
```

Figura B.6: *Programa* somamap

B.7 Programa *tak*

Este programa foi retirado do conjunto de *benchmarks* **nofib**, criado por Partain [86] para medir o desempenho de implementações de linguagens funcionais *lazy*. O programa não calcula nada em específico, mas gera um número de chamadas recursivas grande, mas menor que o da função de Ackermann, resultando na possibilidade de execução com argumentos maiores.

```
tak x y z = if not(y < x) then z
            else tak (tak (x-1) y z)
                     (tak (y-1) z x)
                     (tak (z-1) x y);

main = tak 17 16 8;
```

Figura B.7: *Programa* tak

B.8 Problema das N rainhas

O programa que calcula o número de soluções para o problema das N rainhas também foi inspirado em um programa similar incluído no conjunto de *benchmarks* **nofib**, mas precisou de muitas adaptações pois o original usa muitas características da linguagem Haskell que não estão disponíveis na linguagem dos testes.

```
map f l = if null l then [] else (f (head l)) : (map f (tail l));

filter p l = if null l then []
             else let lh = head l in
                  let lt = tail l in
                  if (p lh) then (lh : (filter p lt)) else (filter p lt);

range n = let aux i = if i > n then [] else (i : (aux (i+1))) in aux 1;

safe d l = if null l then True
             else let x  = head l in
                  let lh = head (tail l) in
                  let lt = tail (tail l) in
                  x /= lh && x /= lh+d && x /= lh-d && (safe x (d+1) lt);

safe1 l = safe 1 l;

cons q l = q : l;

gen n nq = if n == 0 then [[]]
           else let f l = (let g x = (x : l) in map g (range nq)) in
           filter safe1 (map f (gen (n-1) nq));

main = len (gen 10 10);
```

Figura B.8: *Programa* queens

APÊNDICE C

Plataforma de Testes para Algoritmos com Vários Mutadores

Neste apêndice será descrita em maiores detalhes a implementação do algoritmo com vários mutadores baseado em filas de atualizações (descrito no Capítulo 5) na máquina virtual Java Jikes RVM.

A implementação do algoritmo depende diretamente do componente MMTk (*Memory Management Toolkit*) da Jikes RVM. O MMTk compreende uma hierarquia de classes que podem ser especializadas para a implementação de diferentes algoritmos de gerenciamento da memória dinâmica. Em geral, é necessário criar subclasses de algumas subclasses centrais no MMTk para implementar um novo coletor de lixo.

C.1 Estrutura do MMTk

Os componentes centrais do MMTk podem ser divididos em duas partes principais: as classes relacionadas ao *plano* (*plan*) de gerenciamento da memória, e as classes relacionadas à *políticas* (*policy*).

A entidade de mais alto nível no MMTk, e que tem a responsabilidade de orquestrar os componentes do sistema para gerenciar a memória, é um *plano*, realizado no sistema por um objeto da classe `Plan` (ou uma subclasse da mesma), do pacote `org.mmtk.plan`. Um objeto da classe `Plan` (ou uma subclasse da mesma) deve realizar as seguintes tarefas [13]:

- ▷ Gerenciamento do *layout* da memória dinâmica, através de um conjunto de objetos da classe `org.mmtk.policy.Space`.
- ▷ Alocação de memória.
- ▷ Coleta de lixo.
- ▷ Coletar e tratar os dados de uso e contadores de desempenho.

Os dois primeiros itens se referem ao gerenciamento do espaço livre no *heap*, e fazem parte dos componentes do MMTk relacionados às políticas de gerenciamento da memória.

Um plano deve dividir a memória disponível para a máquina virtual em um conjunto de *espaços*. Cada espaço é gerenciado separadamente e pode utilizar uma estratégia de alocação e um coletor de lixo diferente. Alguns tipos de espaços vêm pré-definidos nas classes do MMTk, como espaços de memória estática, espaços gerenciados com coleta por cópia ou coleta por marcação e varredura [13]. A estratégia mais comum é usar um espaço gerenciado por um coletor para representar o *heap* de execução do sistema, e usar espaços estáticos adicionais para estruturas de dados utilizadas pelo coletor em si.

Além das classes relacionadas ao plano e as classes com políticas de gerenciamento, o MMTk também inclui classes utilitárias. Estas classes podem implementar estruturas de dados específicas para uso do coletor, como as filas de atualização no algoritmo do Capítulo 5.

C.2 Implementação do Algoritmo com Vários Mutadores

Na implementação de um novo coletor de lixo na Jikes RVM, é possível reutilizar parte do que já foi implementado no componente MMTk, se o coletor a ser implementado tem semelhanças com os coletores já implementados no *toolkit*. Por exemplo, para implementar um novo algoritmo generacional, é possível criar um plano derivado das classes no pacote `org.mmtk.plan.generational`, que foram projetadas para serem suficientemente genéricas para trabalhar com quaisquer coletores sequenciais que sigam o modelo generacional.

Na versão da Jikes RVM utilizada nesta tese (versão 3.1.1, lançada em Junho de 2010) não há nenhum coletor de lixo *on-the-fly* já definido. Há um conjunto de classes associado a um plano para implementação de coletores baseados em contagem de referências, mas esse coletor é baseado no algoritmo básico de contagem de referências, e não ajudaria na implementação do algoritmo *on-the-fly* para vários mutadores.

Dessa forma, poderia-se implementar como básico o plano para um coletor *on-the-fly*, ou o

plano para coletores baseados em contagem de referências cíclicas (ver Capítulo 3). Decidiu-se pela última opção, já que a estratégia baseada em contagem de referências cíclicas pode ser utilizada na implementação de coletores sequenciais, caso seja necessário. Desta forma, criou-se o pacote `org.mmtk.plan.cyclicrc` e as seguintes classes dentro do pacote:

- `CyclicRCCollector`
- `CyclicRCMutator`
- `CyclicRCConstraints`

Esta última é uma subclasse de `org.mmtk.plan.PlanConstraints`, criada para comunicar ao resto da Jikes RVM algumas características importantes do coletor, como por exemplo se são necessárias barreiras de escrita ou leitura.

Baseados nas classes criadas para o plano básico para contagem de referências cíclicas, criou-se um plano para o algoritmo *on-the-fly* que é apresentado no Capítulo 5 e que tem suporte a múltiplos mutadores. Para abrigar as classes desse plano foi criado o pacote `org.mmtk.plan.cyclicrc.otf` e as seguintes classes dentro do mesmo:

- `OTFCyclicRCCollector`
- `OTFCyclicRCMutator`
- `OTFCyclicRCConstraints`

Além disso, foi necessário implementar classes utilitárias para realizar as estruturas de dados necessárias ao coletor, em especial as filas de registros de atualização. Essas filas foram criadas através da classe `UpdateQueue` no pacote `org.mmtk.utility`, através da extensão da classe `DoublyLinkedList`, já definida no mesmo pacote pela Jikes RVM e que implementa uma lista duplamente encadeada que pode ser mantida em espaços de apoio para o coletor de lixo.

Também foi necessário implementar um alocador para o espaço livre (dentro do pacote `org.mmtk.utility.alloc`) e alterar algumas classes já existentes no MMTk para possibilitar o suporte a um coletor *on-the-fly*.

APÊNDICE D

Outros Trabalhos Realizados Durante o Doutorado

Neste apêndice são descritas outras atividades, não relacionadas aos tema da tese, que foram realizadas durante o período do doutorado que resultou nesta tese.

Essas atividades estão concentradas em dois temas principais: processamento de imagens e computação paralela e distribuída.

D.1 BigBatch

BigBatch é uma plataforma para o processamento de grandes quantidades de imagens de documentos digitalizados, inicialmente descrita no artigo de Lins, Ávila e Formiga [65]. Documentos digitalizados quase sempre precisam ser processados por uma série de filtros para eliminar possíveis artefatos – ruídos, bordas, inclinação – introduzidos pelo processo de digitalização. A plataforma BigBatch reúne algoritmos para realizar o processamento das imagens e métodos para distribuir o processamento de grandes conjuntos dessas imagens entre vários computadores, que podem estar organizados em *clusters* ou grades computacionais. A plataforma BigBatch foi descrita mais recentemente no seguinte artigo [74]:

- Giorgia Mattos, Rafael Dueire Lins, Andrei Formiga e Fernando Mário Junqueira Martins, *BigBatch: A document processing platform for clusters and grids*, In: Proceedings of the 2008 ACM symposium on Applied Computing, SAC'08, pp. 434-441,

ACM Press, 2008.

A plataforma BigBatch foi utilizada como estudo de caso em pesquisas realizadas para analisar comparativamente o desempenho de *clusters* e *grids*. Como é possível, usando BigBatch, realizar as mesmas tarefas – processamento de grandes conjuntos de imagens de documentos – usando tanto *clusters* quanto *grids*, foi possível realizar comparações diretas entre os dois tipos de arquiteturas. A pesquisa relacionada a essas comparações, e sua análise, resultou na tese de doutorado de Giorgia Mattos e nos dois artigos a seguir [75, 76]:

- Giorgia Mattos, Rafael Dueire Lins, Andrei Formiga e Francisco Heron de Carvalho Jr., *A Comparison of Cluster and Grid Configurations Executing Image Processing Tasks in a Local Network*, In: International Conference on Networking 2008, Proceedings, pp. 408-414, IEEE Press, 2008.
- Giorgia Mattos, Rafael Dueire Lins, Andrei Formiga, Francisco Heron de Carvalho Jr. e Fernando Mário Junqueira Martins, *Comparative Aspects between the Cluster and Grid Implementations of BigBatch*, Journal of Universal Computer Science, n. 18, vol. 14, pp. 3031-3050, 2008.

D.2 Processamento de Imagens

Enquanto a pesquisa em torno do BigBatch se concentrava principalmente na distribuição das tarefas de processamento de imagens, usando algoritmos previamente desenvolvidos, o autor desta proposta também realizou pesquisa em algoritmos de processamento de imagens. Essas atividades foram concentradas em dois temas: algoritmos para eliminação de bordas ruidosas de imagens de documentos, e detecção e reconhecimento de códigos identificadores bidimensionais – especificamente o código DataMatrix.

A pesquisa sobre remoção de bordas ruidosas de imagens de documentos resultou em um novo algoritmo para essa tarefa, como relatado no seguinte artigo [40]:

- Andrei Formiga e Rafael Dueire Lins, *Efficient Removal of Noisy Borders of Monochromatic Documents*, In: International Conference on Image Analysis and Recognition, 2009, Proceedings of ICIAR 2009, Lecture Notes in Computer Science, vol. 5627, pp. 158-167, Springer, 2009.

A pesquisa relacionada a detecção e reconhecimento de códigos de identificação (similares a códigos de barra) bidimensionais foi realizada no contexto de um projeto de pesquisa do

convênio UFPE/Hewlett-Packard, e resultou no seguinte artigo aceito para o Symposium of Applied Computing 2011 da ACM:

- Andrei Formiga, Rafael Dueire Lins, Steven J. Simske, Gary Dispoto e Marcelo Thielo, *An Assessment of Data Matrix Barcode Recognition under Scaling Rotation and Warping*, aceito para o Symposium of Applied Computing, SAC 2011 (março), ACM.

Sobre o Autor

O autor nasceu em João Pessoa, Paraíba, no dia 25 de agosto de 1977. Formado em Engenharia Elétrica com habilitação em Controle e Automação pela Universidade Federal de Campina Grande (UFCG). Atualmente é Professor Assistente da Universidade Federal da Paraíba (UFPB).

Entre suas áreas de interesse estão a teoria e implementação das linguagens de programação, lógica, programação concorrente e computação paralela.

Endereço: Rua Monteiro Lobato, 773
Tambaú
João Pessoa – PB, Brasil
C.E.P.: 58.039 – 170

e-mail: andrei.formiga@gmail.com

Esta tese foi diagramada usando LaTeX 2_ε[1] pelo autor.

[1] LaTeX 2_ε é uma extensão do LaTeX. LaTeX é uma coleção de macros criadas por Leslie Lamport para o sistema TeX, que foi desenvolvido por Donald E. Knuth. TeX é uma marca registrada da Sociedade Americana de Matemática ($\mathcal{AMS}$). O estilo usado na formatação desta tese foi escrito por Dinesh Das, Universidade do Texas. Modificado em 2001 por Renato José de Sobral Cintra, Universidade Federal de Pernambuco, e em 2005 por André Leite Wanderley.