UNIVERSIDADE FEDERAL DE JUIZ DE FORA

Departamento de Ciência da Computação

Análise de Desempenho das Implementações de Coleta de Lixo da Máquina Virtual Java

Luciano Jerez Chaves

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Orientador: Prof. Marcelo Lobosco

CIÊNCIA DA COMPUTAÇÃO

Juiz de Fora, MG
Dezembro de 2007

Análise de Desempenho das Implementações de Coleta de Lixo da Máquina Virtual Java

Luciano Jerez Chaves

Monografia submetida ao corpo docente do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora, como parte integrante dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Aprovada pela banca constituída pelos seguintes professores:

Prof. Marcelo Lobosco – orientador
D.Sc. em Engenharia de Sistemas e Computação, COPPE/UFRJ, 2005

Prof. Ciro de Barros Barbosa
D.Sc. em Ciência da Computação, University of Twente, UT - Holanda, 2001

Prof. Rodrigo Weber dos Santos
D.Sc. em Matemática, Physikalisch-Technische Bundesanstalt, P.T.B., Alemanha, 2004

Juiz de Fora, MG
Dezembro de 2007

Sumário

Lista de Reduções iii

Lista de Figuras iv

Lista de Tabelas vi

Resumo vii

Capítulo 1 – Introdução 1

1.1. Contextualização 1

1.2. Motivações 1

1.3. Estrutura da monografia 2

Capítulo 2 – A coleta de lixo 3

2.1. Contagem de referências 4

2.2. Algoritmos de rastreamento 6

2.2.1. Coleta por marcação e varredura 7

2.2.2. Coleta por marcação e compactação 7

2.2.3. Coleta com cópia 8

2.3. Algoritmo de rastreamento incremental 8

2.4. Coleta por gerações 11

2.5. Coleta distribuída 13

2.6. Comparativo 14

Capítulo 3 – A coleta de lixo na JVM 16

3.1. A Máquina Virtual Java 16

3.2. As gerações *HotSpot* 17

3.3. Os coletores disponíveis 18

3.3.1. O coletor serial 18

3.3.2. O coletor paralelo 19

3.3.3. O coletor concorrente 21

3.4. Seleções automáticas e otimizações 24

Capítulo 4 – Avaliação de desempenho 32

4.1. Objetivos 32

4.2. O ambiente experimental 34

4.3. O conjunto de avaliação 34

4.4. Análise dos resultados 38

4.4.1. GCOld 38
4.4.2. GCBench 42
4.4.3. JGFCreateBench 47
4.4.4. JGFSerialBench 51
4.4.5. JGFForkJoinBench 56
4.4.6. GeneId 60
4.4.7. JGFSORBench 66
4.4.8. JGFMonteCarloBench 72
4.5. Desempenho dos coletores 79
4.6. Reduzindo os custos da coleta de lixo 82
Capítulo 5 – Conclusões 84
Referências bibliográficas 85

Lista de Reduções

J2SE	Java 2 *Standad Edition*
JVM	*Java Virtual Machine*
MB	*MegaBytes*
SO	Sistema Operacional
SOR	*Successive Over Relaxation*
SPARC	*Scalable Processor ARChitecture*
CMS	*Concurrent Mark-Sweep*
GB	*GigaByte*

Lista de Figuras

Figura 2.1 – Organização de memória utilizando contagem de referência 5

Figura 2.2 – Ciclos no mecanismo de contagem de referencia 6

Figura 2.3a – Coleta de lixo com cópia, configuração inicial 9

Figura 2.3b – Coleta de lixo com cópia, configuração final.....9

Figura 2.4 – Estrutura da memória para o coletor de lixo *Treadmil*.....11

Figura 2.5 – Uso de memória em coletas por gerações 12

Figura 3.1 – Organização de memória na *HotSpot Java Virtual Machine* 18

Figura 3.2 – Comparação entre um coletor serial e dois coletores em paralelo na JVM 20

Figura 3.3 – Comparação entre um coletor serial e coletor concorrente na JVM 23

Figura 3.4a – Memória na geração estável antes de uma coleta concorrente.....23

Figura 3.4b – Memória na geração estável depois de uma coleta concorrente 23

Figura 3.5 – Divisão de memória entre gerações na *heap* 27

Figura 4.1 – Comparativo entre os *footprints* das execuções do aplicativo GCOld 43

Figura 4.2 – Comparativo entre os *throughputs* das execuções do aplicativo GCOld 43

Figura 4.3 – Comparativo entre os tempos de execução do aplicativo GCOld 43

Figura 4.4 – Utilização de memória e tempos de pausa no aplicativo GCBench utilizando o coletor de lixo cliente paralelo na geração estável 46

Figura 4.5 – Comparativo entre os *footprints* das execuções do aplicativo GCBench.....48

Figura 4.6 – Comparativo entre os *throughputs* das execuções do aplicativo GCBench 48

Figura 4.7 – Comparativo entre os tempos de execução do aplicativo GCBench.....48

Figura 4.8 – Comparativo entre os *footprints* das execuções do aplicativo JGFCreateBench 52

Figura 4.9 – Comparativo entre os *throughputs* das execuções do aplicativo JGFCreateBench ...52

Figura 4.10 – Comparativo entre os tempos de execução do aplicativo JGFCreateBench 52

Figura 4.11 – Comparativo entre os *footprints* das execuções do aplicativo JGFSerialBench 55

Figura 4.12 – Comparativo entre os *throughputs* das execuções do aplicativo JGFSerialBench.....55

Figura 4.13 – Comparativo entre os tempos de execução do aplicativo JGFSerialBench 55

Figura 4.14 – Comparativo entre os *footprints* das execuções do aplicativo JGFForkJoinBench ...61

Figura 4.15 – Comparativo entre os *throughputs* das execuções do aplicativo JGFForkJoinBench.....61

Figura 4.16 – Comparativo entre os tempos de execução do aplicativo JGFForkJoinBench 61

Figura 4.17 – Comparativo entre os *footprints* do aplicativo GeneId para *heap* de 512MB.....67

Figura 4.18 – Comparativo entre os *throughputs* do aplicativo GeneId para *heap* de 512MB.....67

Figura 4.19 – Comparativo dos tempos de execução do aplicativo GeneId para *heap* de 512MB..67

Figura 4.20 – Comparativo entre os *footprints* do aplicativo GeneId para *heap* de 1024MB.....68

Figura 4.21 – Comparativo entre os *throughputs* do aplicativo GeneId para *heap* de 1024MB.....68

Figura 4.22 – Comparativo dos tempos de execução do aplicativo GeneId para *heap* de 1024MB 68

Figura 4.23 – Comparativo entre os *footprints* do aplicativo JGFSORBench.....73

Figura 4.24 – Comparativo entre os *throughputs* do aplicativo JGFSORBench.....73

Figura 4.25 – Comparativo entre os tempos de execução do aplicativo JGFSORBench.....73

Figura 4.26 – Comparativo entre os *footprints* do aplicativo JGFMonteCarloBench.....78

Figura 4.27 – Comparativo entre os *throughputs* do aplicativo JGFMonteCarloBench 78

Figura 4.28 – Comparativo entre os tempos de execução do aplicativo JGFMonteCarloBench 78

Lista de Tabelas

Tabela 2.1 – Comparativo entre dois grupos básicos de coletores 15

Tabela 3.1 – Valores padrão para a JVM rodando no Solaris em plataforma SPARC 28

Tabela 4.1 – Possíveis configurações entre coletores e opções na JVM.....33

Tabela 4.2 – Valores médios obtidos na execução do GCOld para os indicadores avaliados 40

Tabela 4.3 – Valores médios obtidos na execução do GCBench para os indicadores avaliados....45

Tabela 4.4 – Valores médios obtidos na execução do JGFCreateBench para os indicadores avaliados.....50

Tabela 4.5 – Valores médios obtidos na execução do JGFSerialBench para os indicadores avaliados.....54

Tabela 4.6 – Valores médios obtidos na execução do JGFForkJoinBench para os indicadores avaliados.....57

Tabela 4.7 – Valores médios obtidos na execução do GeneId para os indicadores avaliados em uma *heap* de até 512MB 63

Tabela 4.8 – Valores médios obtidos na execução do GeneId para os indicadores avaliados em uma *heap* de até 1024MB 64

Tabela 4.9 – Valores médios obtidos na execução do JGFSORBench para os indicadores avaliados em uma *heap* de até 128MB 70

Tabela 4.10 – Valores médios obtidos na execução do JGFMonteCarloBench para os indicadores avaliados em uma *heap*de até 256MB 75

Resumo

Os mecanismos de coleta de lixo automática surgiram na década de 60. Com o passar dos anos, diversas linguagens de programação como Smalltalk, ML, Lisp, Prolog e, mais atualmente JAVA e C#, adotaram estes mecanismos.

Com o objetivo de livrar o programador da responsabilidade de gerenciar a memória que sua aplicação solicitou, simplificando assim a programação de algoritmos complexos e eliminando os erros por falta de memória ou por referências a objetos que foram liberados antes do momento correto, diversas abordagens para estes mecanismos foram propostas na literatura, cada uma delas com suas peculiaridades.

Ao longo deste trabalho serão abordados os principais conceitos em coleta de lixo, bem como os algoritmos clássicos da literatura e suas principais características. Serão também apresentados e avaliados, através de um conjunto de aplicações distintas, os mecanismos de coleta de lixo existentes na Máquina Virtual Java em suas versões atuais.

Palavras chave

Coleta de lixo, Java, desempenho, comparação, análise, máquina virtual.

Capítulo 1

Introdução

1.1. Contextualização

O gerenciamento de memória é o processo de reconhecer quando objetos alocados não são mais necessários, liberar a memória por eles utilizada e torná-la disponível para alocações subseqüentes. Em algumas linguagens de programação, o gerenciamento de memória é uma responsabilidade do programador. A complexidade desta tarefa conduz a vários erros comuns que podem fazer com que o comportamento da aplicação seja errôneo ou inesperado. Como resultado, uma grande porção do tempo do desenvolvedor é gasta depurando o código e tentando corrigir tais erros.

Alguns dos problemas que normalmente ocorrem em programas com gerenciamento de memória explícito são conhecido por *dangling reference* e *memory leaks*. O primeiro deles acontece quando um objeto é retirado de uma determinada posição da memória enquanto ainda é referenciado pela aplicação. Isso pode provocar erro durante o acesso ao ponteiro com referência inválida. O segundo problema ocorre quando a memória alocada não é mais referenciada e não é liberada, fazendo com que o consumo da mesma seja crescente durante toda a execução.

Uma abordagem alternativa para o gerenciamento de memória que atualmente é utilizada, especialmente pelas modernas linguagens orientadas a objetos, é o gerenciamento automático por um programa chamado coletor de lixo, que habilita o aumento da abstração de interfaces e códigos mais confiáveis.

O coletor de lixo previne o problema das referências inválidas, porque objetos que ainda são referenciados em algum lugar da aplicação não serão eliminados da memória; bem como o problema do vazamento de memória, já que todos os objetos não mais alcançáveis são eliminados assim que possível.

1.2. Motivações

Um coletor de lixo deve ser seguro e compreensivo. Isto é, dados ainda úteis para a aplicação não devem ser erroneamente liberados, e o lixo não deve ser mantido na memória mais do que um pequeno número de ciclos de limpeza. Também é desejável que

os coletores de lixo operem eficientemente, sem introdução de longas pausas para seus processos. Existem diversas abordagens para realizar essas tarefas, cada uma delas com suas vantages e desvantagens.

Como na maioria dos sistemas computacionais, os coletores de lixo também enfrentam os dilemas entre tempo, espaço e freqüência. Por exemplo: se o tamanho da *heap* de uma aplicação for pequeno, as limpezas serão rápidas, mas a *heap* irá saturar logo, requerendo coletas mais freqüentes. Ao contrário, *heaps* grandes caracterizam coletas menos freqüentes, mas provocam pausas maiores durante o processo.

O objetivo desta monografia é avaliar situações como as apresentadas acima para algoritmos de coleta distintos em diferentes classes de aplicativos. Para tanto, será realizado um estudo detalhado e comparativo entre os mecanismos de limpeza existentes na Máquina Virtual Java, com o intuito de obter as informações necessárias e apresentar os pontos característicos de cada classe de coletores. Além disso, verificaremos algumas opções existentes que podem melhorar o desempenho destes coletores para algumas métricas.

1.3. Estrutura da monografia

A presente monografia está dividida da seguinte forma: o capítulo 2 aborda os algoritmos de coleta de lixo comuns na literatura, as principais características, as diversas vantagens e as possíveis dificuldades decorrentes do uso de cada paradigma apresentado; o capítulo 3 é voltado para a Máquina Virtual Java e apresenta os mecanismos de limpeza implementados na mesma, com informações de utilização e opções para o gerenciamento de memória; o capítulo 4 contém as informações das avaliações realizadas e dos resultados obtidos, além de sugestões para modificações de parâmetros da máquina virtual; e, finalmente, o capítulo 5 conclui a presente monografia e apresenta sugestões para trabalhos futuros.

Capítulo 2

A coleta de lixo

Segundo JONES (1996), coleta de lixo é uma forma automática de realizar gerenciamento de memória. Com este recurso, é possível recuperar zonas de memória que o programa não utiliza mais e torná-las disponíveis novamente. Entretanto, antes de fazer uso desses mecanismos, três perguntas precisam ser respondidas para que seja possível definir qual tipo de coletor deve-se utilizar, já que existem variações do mesmo, cada uma delas com características específicas focadas em determinados tipos de aplicações.

A primeira pergunta é: como identificar o lixo? A definição dos objetos que serão liberados da memória pode ser explicitada pelo usuário e sujeita a erros, como acontece nas linguagens C e C++; ou através da descoberta automática, utilizando algoritmos capazes de fazer a identificação dos objetos inúteis, ao custo de mais processamento.

Quando a escolha for possível, a melhor opção vai depender do tipo de aplicação a ser executada. Uma liberação explícita pode ser válida para situações onde existe uma alta taxa de criação e destruição de objetos, bem como em situações onde esses objetos são úteis por pouco tempo, caracterizando uma vida curta. Nestes casos, a tendência é que a memória fique cheia de lixo com grande facilidade e as rotinas para limpeza sejam invocadas freqüentemente, resultando em grande perda de desempenho da aplicação principal, já que esse processo automático vem acompanhado de custos, como veremos a diante.

Depois da definição do que coletar, também é importante saber quando coletar o lixo. Para WITHINGTON (1991), duas políticas básicas podem ser abordadas: esperar até o último momento, quando não houver mais memória disponível, para que a execução da limpeza reduza ao máximo à interferência na aplicação; ou optar por uma coleta contínua, que acarreta em um melhor desempenho ao custo de uma programação mais árdua em conjunto com desperdício de processamento.

Ao responder essa pergunta, é importante considerar que, independente de quando a coleta seja executada, ela virá acompanhada de um custo e uma queda no desempenho da aplicação. Como uma tentativa de amenizar esse prejuízo, WILSON (1993) propõe a procura por momentos menos cruciais da execução da aplicação principal, no qual o processador fica ocioso, como em acessos aos dispositivos de entrada/saída. O objetivo é que as pausas para a limpeza sejam pequenas e pouco freqüentes.

A última e não menos importante entre as três perguntas inicialmente propostas é: como fazer a limpeza? Existem muitos algoritmos para realizar essa tarefa. Alguns deles se propõem a eliminar os objetos inúteis e deixar o espaço livre para que um novo objeto do mesmo tipo possa ser alocado neste local. Estes são os chamados mecanismos de reuso. Em contrapartida, existem os métodos de reciclagem, que recuperam toda memória liberada e a torna disponível para qualquer nova solicitação de espaço por parte da aplicação.

Para este trabalho, a primeira pergunta já está respondida. Assumindo a identificação do lixo de forma automática, é possível apresentar algumas respostas para a segunda e terceira perguntas. Dos diversos algoritmos existentes para tal processamento, cada um tem suas características individuais. Mesmo assim, é possível dividi-los em dois grupos básicos: algoritmos de contagem de referências (*reference counting*) e algoritmos de rastreamento (*tracing*). Cada um desses grupos será abordado nas seções seguintes (seções 2.1 e 2.2, respectivamente), onde o objetivo é avaliá-los com maior profundidade e direcioná-los à classe de aplicações em que se comportariam da melhor forma.

Após a análise destes dois grupos básicos, algumas considerações sobre outras técnicas são apresentadas, como algoritmos de rastreamento incrementais (seção 2.3), coleta por gerações (seção 2.4) e mecanismos distribuídos (seção 2.5). Ao final do capítulo é exibido um comparativo (seção 2.6) entre os métodos apresentados.

2.1. Contagem de referências

Os algoritmos de contagem de referências se caracterizam por manter um contador para cada objeto em memória indicando quantos ponteiros existem para este objeto. Toda vez que uma referência de um objeto ao outro é criada ou destruída, os contadores são atualizados. Dessa forma, se um desses contadores chegar a zero, é possível concluir que nenhum outro objeto referencia o mesmo, sendo assim, este é considerado lixo e pode ser liberado da memória. A figura 2.1 representa um modelo de memória com o mecanismo de contagem de referências.

Esse tipo de algoritmo tem a característica de executar em paralelo com a aplicação principal, o que o torna incremental e atende as características de um coletor de tempo real. É importante lembrar que sempre que um objeto é liberado, as referências deste objeto para outros são removidas, o que pode gerar mais lixo, fazendo com que mais objetos sejam desalocados. Assim, se existir uma lista encadeada simples de objetos na memória e um

objeto for considerado lixo, a referência para o próximo objeto será destruída. Se não existirem ponteiros externos para esta lista, toda ela será removida; já que cada vez que retiramos um objeto, destruímos a referência para o próximo e este se torna lixo também. Esse tipo de situação pode gerar uma pausa maior para a coleta e refletir diretamente na aplicação principal.

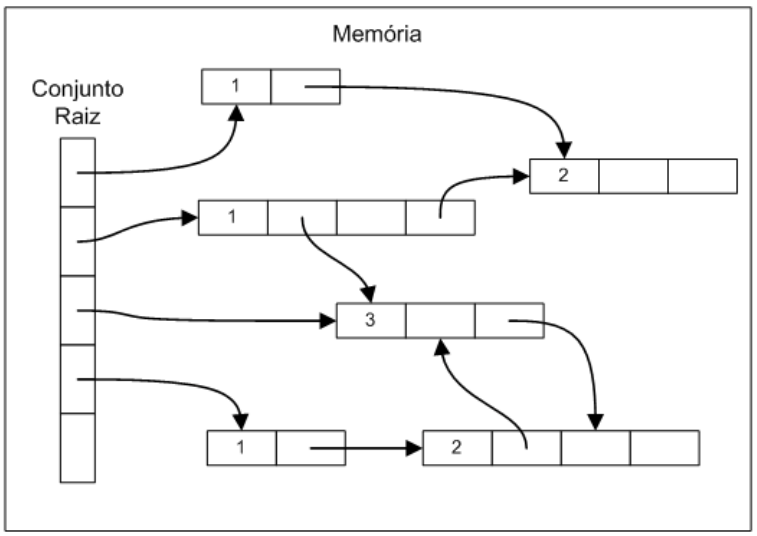

Figura 2.1 – Organização de memória utilizando contagem de referência

WILSON (1992) apontou dois problemas com os algoritmos de contagem de referência que podem ser destacados: é difícil que eles sejam eficientes e eles não são sempre efetivos.

O problema da efetividade é causado pela não detecção de ciclos. Se os ponteiros de um grupo de objetos criarem um ciclo direto, o contador de referencias do objeto nunca será reduzido à zero, mesmo que não exista um caminho a partir do conjunto raiz para esse ciclo.

A figura 2.2 exemplifica uma situação de ciclo em memória. Os objetos no interior do círculo não são mais alcançáveis pela aplicação, mesmo assim seus contadores são diferentes de zero. É normal pensar que ciclos não são comuns, mas na realidade os mesmos ocorrem com grande freqüência, principalmente em estrutura de árvores, onde nodos filhos podem manter ponteiros para nodos pais, facilitando a travessia da mesma (MCBETH, 1963).

Coletores por contagem de referência geralmente incluem algum outro coletor, para que quando o número de ciclos crescerem o suficiente para encher a memória, o outro método possa solucionar o problema.

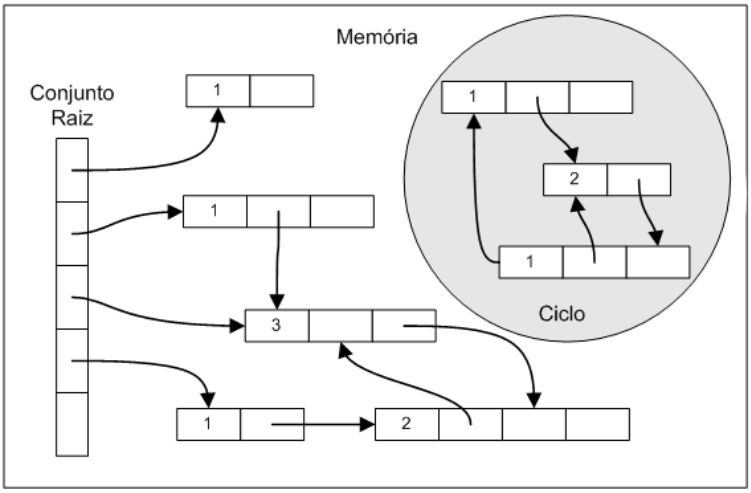

Figura 2.2 – Ciclos no mecanismo de contagem de referencia

A eficiência está relacionada ao custo das atualizações dos contadores dos objetos a cada manipulação de referências da aplicação principal. Esse tipo de situação se torna crítica quando a vida do objeto é muito pequena, como no caso de ponteiros em passagem de parâmetro de funções: os ponteiros são criados e destruídos logo em seguida. Nesta situação, muito trabalho é despendido, resultando no atraso da aplicação principal. MCBETH (1963) propôs algumas soluções para esse problema, como um mecanismo de atualização postergada, onde mudanças em um curto período de tempo não seriam analisadas.

2.2. Algoritmos de rastreamento

Outra alternativa para a coleta de lixo são os algoritmos de rastreamento. Estes algoritmos partem de uma idéia diferente e tão simples quanto a apresentada anteriormente.

Esses algoritmos podem ser divididos em duas etapas: a detecção do lixo e a liberação do mesmo, que pode ser efetuada de diferentes maneiras. Na prática, essas etapas podem ser temporariamente e funcionalmente intercaladas (BARROS, 2006).

Para a execução destes algoritmos, é necessário o conhecimento do grafo de relações entre objetos, onde as arestas são as referências de um objeto a outro na memória. A partir de um conjunto de vértices iniciais, conhecido por conjunto raiz (*root set*), o coletor faz uma busca por todo o grafo, em largura ou profundidade, marcando todos os objetos pelos quais percorrer. Estes são os objetos considerados alcançáveis e serão chamados de objetos vivos ao longo deste trabalho. Ao final desta primeira etapa, temos todos os objetos vivos marcados. A partir daí, uma inspeção completa na memória é

executada à procura dos objetos que não foram marcados para que possam ser eliminados. Esta segunda etapa pode ser executada de diferentes formas, como veremos agora.

2.2.1. Coleta por marcação e varredura (*mark-sweep collection*)

Neste tipo de coleta, o algoritmo percorre toda a memória e, ao encontrar algum objeto que não esteja marcado, simplesmente o remove liberando seu espaço.

Três problemas são gerados neste tipo de coleta: a memória se torna altamente fragmentada, já que podem existir objetos de tamanhos variados em posições distintas. Assim, temos que manter listas de posições de memória vazia para novas alocações, o que pode ser custoso na maioria das vezes (WILSON, 1992).

O segundo problema está relacionado ao tempo para execução desta coleta. Se a pilha de memória for muito grande, e o coletor tiver que percorrê-la duas vezes (uma para cada etapa), a aplicação principal ficará muito tempo bloqueada, esperando o fim da coleta. Isso pode ser prejudicial para algumas aplicações, principalmente as de tempo real.

O terceiro problema está relacionado ao princípio da localidade de referência e uso de memória *cache*. Se os novos objetos alocados são espalhados pela memória, a localidade de referência é destruída e uma queda no desempenho será observada com facilidade. HERTZ (2005) aborda esse problema em sua proposta de um mecanismo que evite paginação na memória virtual do SO (Sistema Operacional). O algoritmo intitulado *bookmarking collector* cria um registro de páginas virtuais que estão presentes e são removidas da memória física do sistema. Assim, durante as coletas, somente as páginas residentes na memória física são analisadas, o que diminui o tempo total da limpeza e evita que as páginas em disco tenham que ser carregadas para serem examinadas e limpas. Dessa forma, objetos no disco somente serão analisados quando voltarem à memória. Todo o espaço liberado para novas alocações continua na memória física, o que reduz a paginação e ajuda a manter a localidade de referência.

2.2.2. Coleta por marcação e compactação (*mark-compact collection*)

Esse tipo de coleta visa solucionar dois dos problemas apresentados anteriormente. Na tentativa de eliminar a fragmentação e problemas com localidade de referência, esse tipo de coletor faz uma compactação dos objetos que não são eliminados da memória até que todos se tornem contíguos. Essa abordagem facilita a alocação de novos objetos, que agora se reduz a um incremento no ponteiro da pilha, já que toda memória ao final dela está livre.

Como conseqüência, esse mecanismo é incapaz de realizar todas as suas funções percorrendo apenas uma vez a memória. Um mesmo objeto é analisado em uma primeira etapa que o identifica ou não como lixo e, em uma segunda etapa, é feita a compactação e atualização dos ponteiros. Essa deficiência resulta em mais atraso para a aplicação principal.

2.2.3. Coleta com cópia (*copying collection*)

Os algoritmos de coleta com cópia realizam a limpeza de forma um pouco diferente. Ao invés de procurar pelos objetos que foram marcados como lixo, ao ir percorrendo o grafo de relações, o coletor vai copiando os objetos alcançáveis para uma nova área da memória, onde estes são colocados em seqüência para evitar a fragmentação. Neste caso, a identificação do lixo e a sua liberação de espaço são realizadas em uma única varredura da memória.

Para que isso seja possível, é necessário que a memória seja dividida em duas sub-áreas (origem e destino), na qual somente a origem estará em uso pela aplicação. Quando uma limpeza se fizer necessária, a aplicação é interrompida, os objetos vivos são copiados da origem para o destino e os nomes destas subáreas são trocados, e por fim a aplicação volta a utilizar a origem, já com os objetos compactados. As figuras 2.3a e 2.3b mostram a configuração da memória antes e depois de uma limpeza.

Nesse tipo de abordagem todas as referências a objetos que mudaram de posição na memória têm que ser atualizadas, o que demanda um esforço computacional adicional. Outro fator identificado como problemático para este algoritmo é a necessidade de se ter muita memória disponível para a aplicação, já que metade dela se torna inativa entre duas coletas consecutivas. Uma possível solução para esse problema é apresentada adiante.

2.3. Algoritmo de rastreamento incremental

Para os sistemas verdadeiramente de tempo real, a escolha da técnica para a coleta de lixo tem grande impacto na eficiência da aplicação. Nestes casos, não podemos tratar a limpeza como um processo atômico, que simplesmente interrompe a execução da aplicação principal e realiza sua tarefa. Todo esse processo tem que ser feito de maneira incremental. A identificação do lixo e sua remoção da memória são feitas de maneira concorrente com a aplicação, o que introduz um problema: a aplicação pode modificar a memória enquanto a coleta está sendo executada.

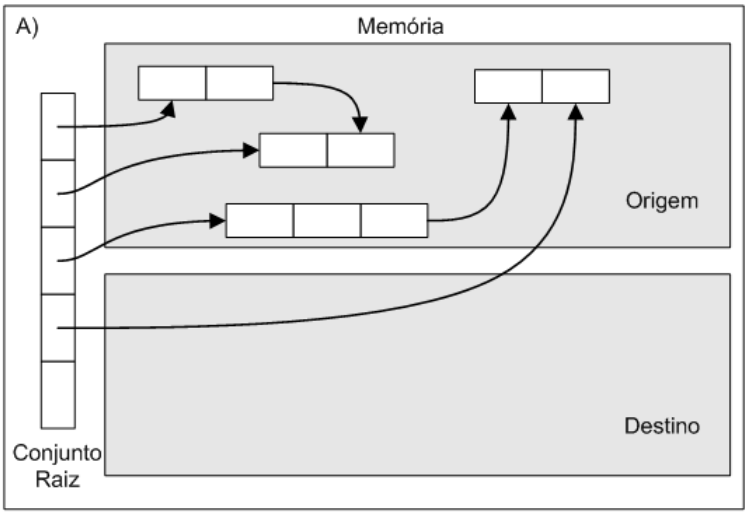

Figura 2.3 a) Coleta de lixo com cópia, configuração inicial

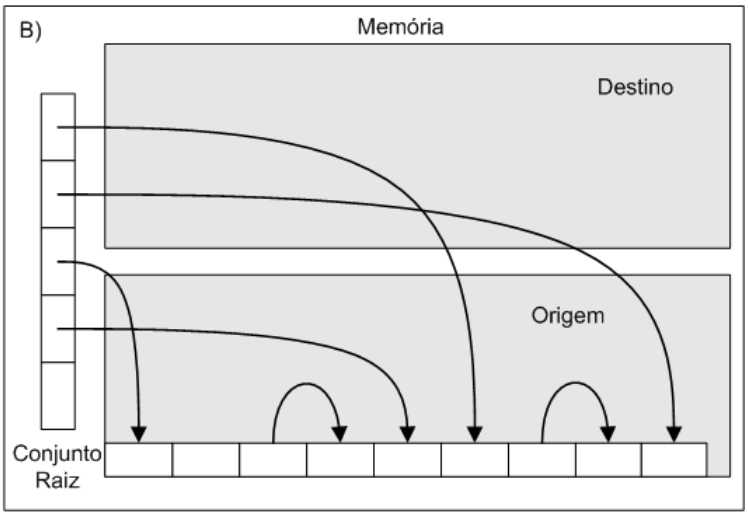

Figura 2.3 b) Coleta de lixo com cópia, configuração final

O algoritmo das três cores apresentado é considerado uma solução simples para esse tipo de necessidade. Assim como nos algoritmos de rastreamento, o coletor percorre a memória marcando os objetos pelos quais ele passar. A diferença seria a abstração do uso de cores para esse processo. Seriam utilizados três grupos para os objetos: os objetos do grupo da cor preta seriam considerados os que estão vivos e devem ser mantidos na memória; objetos da cor branca seriam todos os candidatos a lixo, os quais ainda não foram analisados. Um novo grupo, da cor cinza, seria criado e representaria os objetos em um estágio intermediário entre o grupo branco e preto. Esses membros do grupo cinza são os objetos que são alcançáveis, mas os objetos para os quais estes mantêm referências podem se tornar lixo antes de serem analisados. Isso acontece em mecanismos incrementais

porque a aplicação principal pode modificar o grafo de transições da memória durante a passagem de um objeto do grupo branco para o preto.

A execução do algoritmo é simples: inicialmente, os objetos do conjunto raiz são coloridos de cinza. Escolhe-se qualquer objeto cinza para análise. Todas as referências deste objeto para outros são analisadas e, quando ela apontar para algum objeto filho que pertencer ao grupo branco, devemos mover o objeto filho para o grupo cinza. Ao final da análise das referências, o objeto pai é movido para o grupo preto. Esse processo é repetido até que o grupo cinza se torne vazio. Nessa hora, o que sobrar no grupo branco é lixo e pode ser eliminado e o que tivermos no grupo preto são os objetos alcançáveis pela aplicação.

Dessa forma, é possível garantir que todos os objetos da memória pertencem a um único conjunto e que não existem referências diretas de objetos do conjunto preto para objetos do conjunto branco. Essa última propriedade garante que os objetos do grupo branco podem ser destruídos com segurança. Se a aplicação principal alterar o grafo de relações durante o rastreamento, o objeto modificado vai para o grupo cinza para ser novamente analisado, eliminando o problema com a mudança concorrente.

Uma proposta para combater o problema de desperdício de memória no coletor com cópia é utilizar um mecanismo de coleta incremental com listas. BAKER (1992) propôs um algoritmo para coleta de lixo chamado *Treadmil*. A idéia inicial é colocar todos os objetos da aplicação em uma única lista circular duplamente encadeada mantendo elementos de uma mesma cor em posições subseqüentes.

A lista é dividida em quatro segmentos como representa a figura 2.4: A lista de novos, onde os objetos de novas alocações são colocados; a lista de livre, onde existe espaço disponível para as novas alocações; a lista de origem, que contém os objetos que foram alocados antes do início da coleta e irão ser analisados e a lista de destino, que inicialmente é vazia e começa a ser preenchida com os objetos que são analisados e vão se tornando pretos ou cinzas.

A alocação é feita incrementando o ponteiro que separa a lista de novos da lista de livres e colorindo os novos objetos de preto. No início de uma coleta, o segmento de novos é sempre vazio. Ao final da execução, que é semelhante ao algoritmo das três cores apresentado anteriormente, todos os objetos alcançáveis da lista de origem foram movidos para a lista de destino, que deve conter somente objetos pretos (se existirem objetos cinza estes ainda devem ser analisados). Os objetos que sobraram na lista de origem são

considerados lixos. Nesta hora é suficiente mesclar a lista de origem com a lista de livres e o algoritmo está pronto para uma nova coleta.

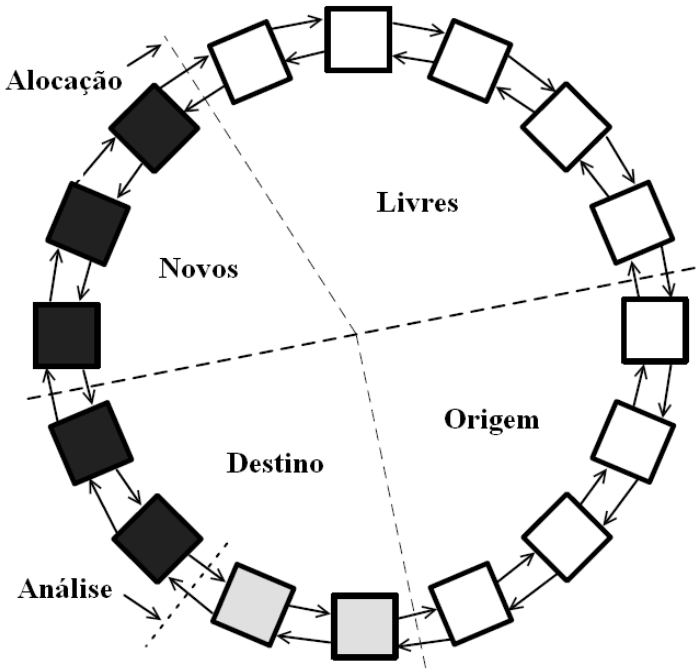

Figura 2.4 – Estrutura da memória para o coletor de lixo *Treadmil*

2.4. Coleta por gerações (*generational collection*)

O maior problema com eficiência nos algoritmos de coleta de lixo é que a memória da aplicação pode ficar grande, e percorrê-la toda de uma vez torna-se bastante custoso. Em quase todos os programas em linguagens variadas, a maioria dos objetos vive por um período muito pequeno, enquanto uma pequena porcentagem deles vive por muito tempo (UNGAR, 1984), (LIEBERMAN, 1983).

Em uma coleta tradicional, devido ao seu pequeno intervalo de varredura, um mesmo objeto pode ser considerado vivo em vários rastreamentos, e talvez seja movido de um lugar para outro várias vezes. A coleta por gerações pode reduzir esse excesso de cópias, dividindo a memória da aplicação de acordo com a idade do objeto.

Novos objetos são sempre alocados na área mais nova. Quando essa parte da memória fica cheia, um processo de limpeza é invocado somente para esta área. O lixo é eliminado e objetos acima de uma determinada idade (consideremos idade como número de sobrevivências às limpezas) são promovidos para áreas de memória de objetos mais

velhos. Para que esse algoritmo seja eficiente, é necessário que o número de gerações seja superior a dois (CAUDILL, 1986).

Nesse modelo, os objetos mais novos recebem a coleta de lixo com freqüência maior do que os objetos mais velhos, que somente são limpos quando chegam ao seu limite de sua porção de memória. A figura 2.5 representa a simulação do uso de memória para um sistema de coleta de lixo por gerações. Cada bloco representa um objeto na memória. Todos os novos objetos são alocados na geração mais nova, representada na parte inferior da figura. Quando não há mais espaço para novas alocações nesta área, uma limpeza é invocada somente para esta parte inferior. Alguns objetos são promovidos à geração superior e outros objetos são identificados como lixo e retirados da memória. Com o tempo, a segunda geração também irá saturar. Nesta hora, quando a coleta da geração mais nova ocorrer e objetos forem promovidos a gerações mais velhas, uma coleta na geração superior será executada. Se o mecanismo possuir uma terceira geração, alguns objetos serão também promovidos, em efeito cascata.

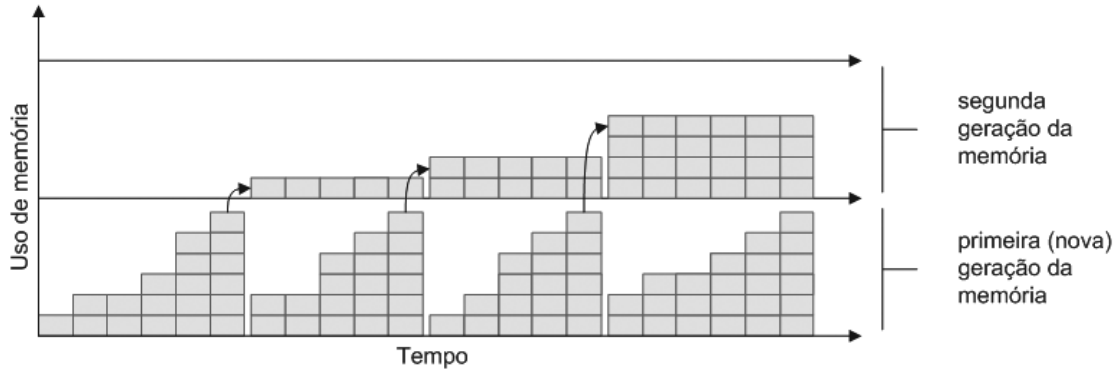

Figura 2.5 – Uso de memória em coleta por gerações

A coleta em cada geração pode ser feita utilizando técnicas distintas, como as já apresentadas aqui. A escolha pode ser associada à freqüência e necessidade de eficiência que determinada geração demanda. Para gerações mais novas, que são coletadas com grande freqüência, os algoritmos mais rápidos podem ser uma boa opção, enquanto para gerações mais velhas, algoritmos incrementais podem executar seu trabalho com boa eficiência.

É importante lembrar que não podemos considerar as gerações independentes umas das outras. Objetos podem manter referências para objetos de outras gerações, que devem ser analisadas quando a coleta ocorrer. Se o coletor utilizado movimentar os objetos na

memória, é importante saber se objetos de outras gerações possuem ponteiros para ele, para que os mesmos sejam atualizados. Esse tipo de necessidade introduz um custo ao coletor, já que o mesmo deve prover uma forma de armazenar essas referências entre as gerações em uma tabela ou estrutura similar para identificá-las no momento da coleta.

2.5. Coleta distribuída

Os coletores de lixo para sistemas distribuídos são bem mais complexos que os locais. Aspectos como troca de mensagens e disponibilidade da rede precisam ser considerados em um processo de limpeza

Um mecanismo de contagem de referências em um coletor distribuído pode ser construído a partir de uma extensão simples da contagem de referências local (PLAINFOSSÉ, 1995). Para este modelo, se um objeto em uma máquina A criar uma referência à um objeto em outra máquina B, é necessário que A envie para B uma mensagem pela rede para que B possa atualizar seu contador de referências. O mesmo procedimento é necessário no caso de remoção de referências.

Essa forma de abordagem necessita de alguns cuidados para que objetos ainda referenciados não sejam considerados lixo; já que é possível que as mensagens na rede cheguem ao destinatário fora de ordem. Por exemplo, se uma mensagem informando a remoção da última referência para um objeto chega antes de uma mensagem informando a criação de uma nova referência ao mesmo, o objeto será eliminado da memória erroneamente. LERMEN (1986) propôs um protocolo de comunicação para solucionar este tipo de problema. Em seu trabalho, é proposto que todas as mensagens sejam entregues na ordem que foram criadas e devem ser sempre confirmadas. Um objeto só pode ser coletado quando um número igual de mensagens de criação, remoção e confirmação for recebido pelo mesmo objeto. Esse protocolo garante a efetividade do mecanismo ao custo de um aumento no número de mensagens trafegando pela rede.

Outra maneira de realizar a coleta é através de um coletor de marcação e varredura distribuído. Para isso, a fase inicial de marcação é feita em sincronia com todas as máquinas, onde mensagens são enviadas e recebidas indicando que objetos devem ser marcados. O coletor local pára seu processo sempre que precisa marcar um objeto em outra máquina e retoma ao trabalho sempre que recebe a confirmação da marcação. Ao receber um pedido de marcação, ele interrompe o seu trabalho, faz a marcação e responde ao nó solicitante. Assim, todo o processo é feito em cooperação entre os nós do sistema

distribuído e quando todos tiverem terminado este processo, eles se comunicam indicando o fim da fase. A partir deste momento, os coletores locais iniciam individualmente a fase de desalocação de memória a fim de se desfazer dos objetos públicos e locais que não serão mais utilizados.

Existem outros algoritmos na literatura para coleta distribuída, como os apresentados em VAUGHAN (2000), BOEHM (1991) e BARABASH (2005), mas é importante lembrar que nenhum desses algoritmos consegue satisfazer ao mesmo tempo os requisitos de eficiência, escalabilidade, tolerância a falhas e coleta de todos os tipos de estruturas de dados (BARROS, 2006).

2.6. Comparativo

A escolha dentre as técnicas existentes depende do tipo de aplicação que está sendo executada. Na verdade, a miscigenação de diferentes abordagens pode ser uma boa saída, utilizando-se das vantagens que cada algoritmo apresenta e corrigindo as falhas por eles introduzidas.

Em seu artigo, BACON (2004) apresentou a teoria de que os grupos de coletores por contagens de referências e rastreamento são basicamente duais entre si, indicando que, se as melhores otimizações forem feitas em ambos, a tendência é de que os algoritmos se tornem similares. A tabela 2.1 apresenta um comparativo apresentado por BACON (2004) entre esses dois grupos básicos.

Os mecanismos de contagem de referência são considerados incrementais porque atualizam os contadores a cada modificação em ponteiros, enquanto a abordagem por rastreamento trabalha em blocos, examinando toda a memória de uma única vez. O rastreamento não gera custo devido à modificações nos ponteiros. Já na contagem de referências, cada mudança de ponteiros resulta numa modificação em um contador, gerando um custo computacional maior. O *throughput,* que é mensurado pela porcentagem de tempo que a aplicação não executa coleta de lixo, é considerado alto porque as coletas são muito distantes uma da outra. Assim, a aplicação tem um tempo maior de processamento. Do outro lado, os coletores com contadores de referência implicam em pequenas pausas, os que o tornam mais aptos para aplicações de tempo real, mas interrompem o processamento várias vezes. Entretanto, os algoritmos de rastreamento são capazes de identificar e coletar ciclos de referências na memória, que não podem ser executados nos outros modelos sem a ajuda de um mecanismo auxiliar.

Tabela 2.1 – Comparativo entre dois grupos básicos de coletores

	RASTREAMENTO	CONTAGEM DE REFERÊNCIAS
TIPO DE COLETA	Em blocos	Incremental
CUSTO POR EXECUÇÃO	Baixo	Alto
THROUGHPUT	Alto	Baixo
TEMPO DE PAUSA	Grande	Pequeno
ATENDE TEMPO REAL?	Não	Sim
COLETA CICLOS?	Sim	Não

FONTE: BACON (2004)

No capítulo seguinte, analisaremos algumas dessas implementações que estão disponíveis na JVM (*Java Virtual Machine*) e tentaremos identificar quais opções dentre as apresentadas foram utilizadas.

Capítulo 3

A coleta de lixo na JVM

Aos 23 de maio de 1995, a *Sun Microsystems* lançou Java, uma linguagem de programação orientada a objetos de propósito geral, com sintaxe similar ao C e C++, porém omitindo muitas das características que fazem destas complexas, confusas e inseguras. Além de ser projetada para permitir o seu uso em múltiplas arquiteturas, Java implementa um mecanismo de gerenciamento automático de memória, onde o programador precisa se preocupar apenas em criar os objetos necessários, enquanto a limpeza é de completa responsabilidade da máquina virtual. A especificação da Máquina Virtual Java (LINDHOLM, 1999) indica esta característica como segue:

> *Heap storage for objects is reclaimed by an automatic storage management system (typically a garbage collector); objects are never explicitly deallocated.*
>
> LINDHOLM (1999) - Seção 3.5.3

As plataformas Java são utilizadas por uma grande variedade de aplicações, desde pequenos *applets* até grandes *web services* em servidores multiprocessados. Na tentativa de oferecer suporte a estes distintos aplicativos de maneira eficiente, a *HotSpot* JVM implementa diferentes coletores de lixo, cada um com foco em necessidades específicas (SUN, 2006).

Ao longo deste capítulo serão apresentados a Máquina Virtual Java (seção 3.1) e sua organização de memória (seção3.2). Os coletores implementados nas versões 5.0 e 6.0 também são apresentados (seção 3.3), bem como informações de utilização e opções complementares (seção 3.4).

3.1. A Máquina Virtual Java

A Máquina Virtual Java é a base fundamental da plataforma Java. Ela é o componente com a tecnologia responsável pela independência entre o hardware, sistema operacional e a aplicação Java. A JVM é um computador abstrato e, assim como uma máquina real, ela possui um conjunto de instruções e manipula várias áreas de memória em tempo de execução.

Uma máquina virtual que apresente todas as características e funcionalidades contidas na especificação descrita por LINDHOLM (1999), é considerada uma Máquina Virtual Java. A *Sun Microsystems,* retentora dos direitos sobre linguagem Java, possui sua própria máquina virtual, conhecida por *Java HotSpot Virtual Machine*, e é a implementação padrão e mais utilizada para a execução de aplicativos na respectiva linguagem. O nome *HotSpot* é referente a atual tecnologia utilizada, que visa aperfeiçoar o desempenho da mesma. É com base nesta máquina virtual que apresentaremos as informações sobre os coletores de lixo existentes.

3.2. As gerações *HotSpot*

Para que seja possível analisar as implementações dos coletores na *HotSpot* JVM, é necessário que se tenha conhecimento da organização da memória na mesma. Na *HotSpot* JVM, todos os objetos estão alocados na *heap* e esta é dividida em três gerações: a geração jovem (*young*), a geração estável (*old*) e a geração permanente (*permanent*). A geração jovem contém os objetos que foram recentemente alocados na memória; a geração estável contém os objetos que sobreviveram a um determinado número de limpezas e objetos que ocupam muita memória (que neste caso são alocados diretamente nesta área a fim de diminuir o número de cópias e movimentações do mesmo) e a geração permanente, onde são alocados os metadados (objetos que a JVM considera conveniente que tenham um tratamento especial durante as coletas, como descritores de classes e métodos).

A geração jovem ainda é dividida em duas sub-gerações: o éden, onde os novos objetos são alocados; e o espaço de sobreviventes (*survivors spaces*), para onde os objetos que sobrevivem à primeira coleta no éden são transferidos. O espaço de sobrevivente é dividido em origem e destino, para a implementação de um coletor com cópia. Neste caso, sempre um desses espaços está vazio. A figura 3.1 representa a divisão completa da memória na *HotSpot* JVM.

Os mecanismos de coleta para a *heap* são diferentes em cada uma das áreas da memória. Como as novas alocações são feitas na geração jovem, exceto para os objetos que não caibam no éden, a tendência é que esta seja a primeira geração a ficar sem espaço e necessitar de uma coleta. As coletas nesta geração são conhecidas por coletas menores (*minor collections*) e se caracterizam por serem extremamente eficientes na identificação de lixo entre objetos novos. Quando as gerações estável e permanente ficam sem espaço, limpezas nas respectivas áreas são executadas. Estas coletas são conhecidas por coletas

maiores (*major collections*). Se algum desses mecanismos provê compactação, estas são feitas em separado para cada geração (PRINTEZIS, 2004).

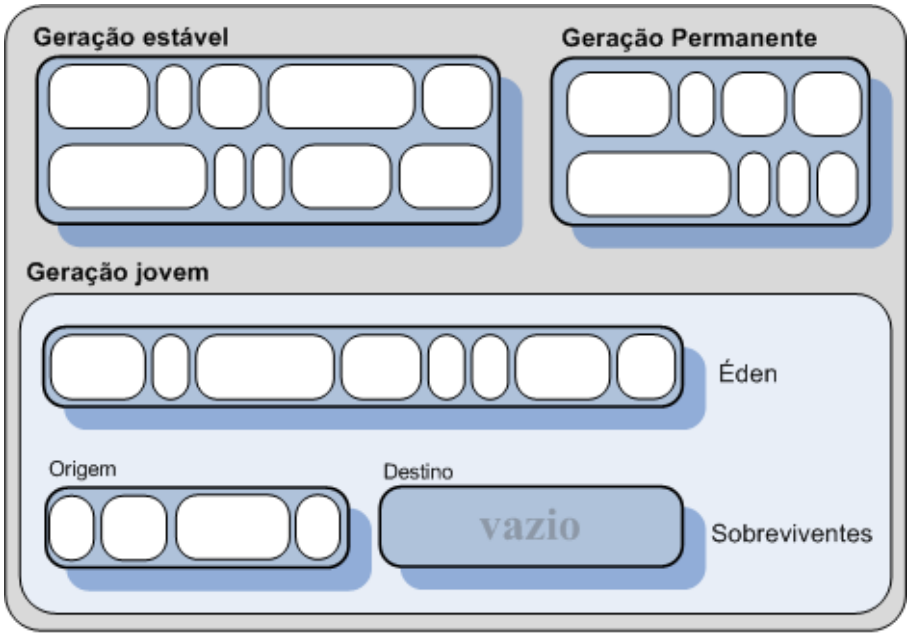

Figura 3.1 – Organização de memória na *HotSpot Java Virtual Machine*

Algumas vezes pode acontecer da geração estável não ter espaço suficiente para aceitar todos os objetos que seriam promovidos da geração jovem, caso uma coleta na geração jovem fosse executada primeiro. Para estes casos, a coleta na geração mais nova não é executada e a limpeza na geração estável é realizada primeiro.

3.3. Os coletores disponíveis

A *HotSpot* JVM nas versões 5.0 e 6.0 incluem três coletores de lixo distintos, cada um deles com diferentes características de desempenho: 1) o coletor serial, que utiliza uma única *thread* para fazer toda a limpeza; 2) o coletor paralelo, utilizado normalmente em máquinas com dois ou mais processadores, onde a coleta pode usufruir do multiprocessamento e 3) o coletor concorrente, cujo principal objetivo é diminuir o tempo de pausas para atender aplicações de tempo real. Esses coletores serão abordados em detalhes a seguir.

3.3.1. O coletor serial

No coletor serial, tanto a coleta de lixo da geração jovem como da geração estável são feitas utilizando uma única *thread*, numa abordagem de execução pare o mundo (*stop-the-*

world), onde a aplicação é interrompida e todos os recursos da máquina são disponibilizados ao coletor. Entretanto, as limpezas são distintas e independentes para cada geração.

Na geração jovem, a *HotSpot* JVM implementa um mecanismo de coleta com cópia. Para isso, ela faz a procura por todos os objetos vivos do éden, que são copiados para o espaço de sobrevivente denominado destino, inicialmente vazio. O espaço denominado origem também é analisado, onde os objetos vivos que ainda são relativamente jovens para serem promovidos à geração estável são copiados para o destino. Os objetos que já estão no espaço de sobreviventes a um determinado tempo e os objetos do éden que são muito grandes são copiados diretamente para a geração estável. Isso evita que grandes porções de memória sejam copiadas inúmeras vezes, já que as coletas nessa geração são bastante freqüentes (RAMAKRISHNA, 2007). Ao fim desta coleta, temos o éden e a origem do espaço de sobreviventes vazios; e somente o sobrevivente de destino contém objetos vivos. Nessa hora, os espaços origem e destino invertem suas entitulações. Com esse modelo, toda nova alocação de memória pode ser feita a partir de um incremento do ponteiro que delimita o espaço vazio do éden.

A limpeza na geração estável é feita com um coletor de marcação e compactação (*mark-sweep-compact*). Na fase de marcação, o coletor identifica dentre os objetos aqueles que ainda estão vivos. Na fase de varredura, os objetos que não estiverem marcados são removidos da *heap* e, na última fase de compactação, todos os objetos são movidos para o começo da geração, num processo de compactação, deixando todo o espaço disponível ao final e facilitando as próximas alocações.

O coletor serial é o coletor padrão para as JVM clientes[1], onde a aplicação não requer um tempo de pausa extremamente baixo. É possível forçar sua utilização através da opção de linha de comando `-XX:+UseSerialGC`.

3.3.2. O coletor paralelo

O coletor paralelo, também conhecido como coletor de *throughput,* é, assim como o serial, um coletor de gerações, porém fazendo uso da disponibilidade de vários processadores de uma máquina multiprocessada. Esse coletor é selecionado como padrão em JVM servidoras, mas é possível utilizá-lo através da opção `-XX:+UseParallelGC`.

[1] JVM clientes e servidores são classificações atribuídas com base no *hardware* existente e serão explicadas em detalhes na seção 3.4

Por padrão, somente as coletas da geração jovem são executadas em paralelo, ainda sim em uma abordagem pare o mundo. Em uma máquina com *n* processadores, o coletor irá criar por padrão *n threads.* A utilização de mais de uma *thread* faz com que o *overhead* do coletor seja diminuído e a aplicação tenha um aumento no *throughput*. O número de *threads* pode ser ajustado através da opção `-XX:+ParallelGCThreads=<N>`, para que não necessariamente todos os processadores tenham que participar da coleta. A figura 3.2 ilustra as diferenças entre o coletor serial e dois coletores paralelos (um com três e outros com seis *threads*) executando a limpeza de uma geração jovem da memória. As setas representam a execução das *threads*, com indicações de pausas para a coleta do lixo.

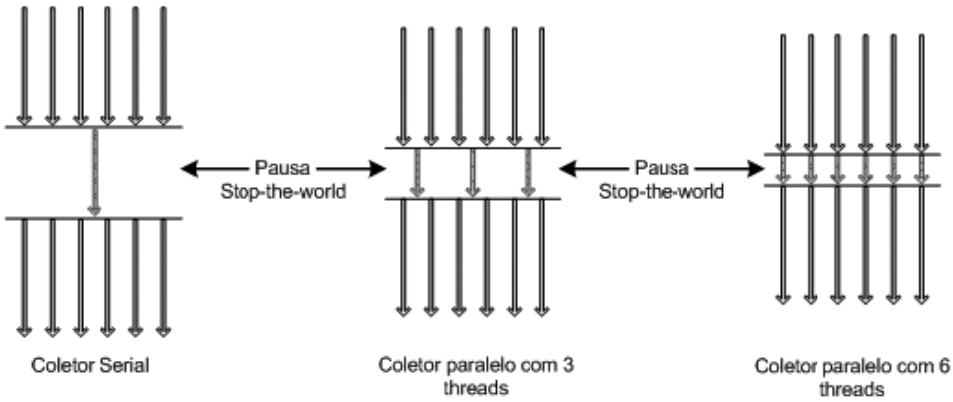

Figura 3.2 – Comparação entre um coletor serial e dois coletores em paralelo na JVM

Para que a coleta possa ser feita em paralelo na geração jovem, cada *thread* reserva uma parte da geração estável para promover os objetos por ela analisados durante a coleta menor. Como não necessariamente todo esse espaço será preenchido, ao final da limpeza pode existir uma fragmentação pequena da geração estável. Segundo SUN (2006), para reduzir esse efeito é necessário diminuir o número de *threads* ou aumentar o tamanho da geração estável.

A partir do J2SE 5.0 *update* 6, uma modificação do coletor paralelo foi introduzida, cuja principal diferença é a implementação de um algoritmo paralelo para todas as coletas, e não só mais na geração jovem. Assim como o coletor anterior, trata-se de uma abordagem pare o mundo, quase paralela e com compactação. Esse coletor pode ser utilizado através da adição da opção `-XX:+UseParallelOldGC` no J2SE 5.0 e não precisa ser especificado no J2SE 6.0, já que para esta última versão ele se tornou padrão da implementação do coletor paralelo normal.

Este algoritmo é dividido em três fases. Primeiramente, cada geração de memória é logicamente dividida em intervalos de tamanho fixo. Na fase de marcação, os objetos do conjunto raiz também são dividido entre as *threads* de coleta, e então todos os objetos vivos são marcados em paralelo. Quando um objeto é identificado como vivo, a região onde o mesmo se encontra é atualizada com as informações de tamanho e localização do objeto. A fase de limpeza e compactação é realizada nas regiões definidas anteriormente, ao invés de utilizarmos os objetos.

Devido à compactação das coletas anteriores, é típico que as porções mais a esquerda de cada geração estejam completamente cheia e contenha pouco lixo. É nesta região que se encontram os objetos sobreviventes a várias coletas. Nestes casos, a quantidade de espaço que pode ser recuperada pode não compensar o custo de realizar uma compactação na mesma (RAMAKRISHNA, 2007). Sendo assim, esse algoritmo examina a densidade da região, começando mais a esquerda e indo para a direita, à procura de um ponto onde a quantidade de espaço a ser recuperada salve o custo de uma compactação. Encontrado esse ponto, a região à esquerda do mesmo passa a ser considerada como prefixo denso e não será modificada; enquanto a região à direita do mesmo será compactada.

Na fase de compactação, as *threads* coletoras utilizam os dados produzidos na fase anterior para identificar as regiões que precisam ser compactadas e preenchidas. Cada *thread* copia os objetos que identificaram como vivos e foram marcados na primeira fase do algoritmo para as regiões especificadas. Esse processo produz ao final uma *heap* que é bastante densa em uma das extremidades e com espaço vazio na outra.

Aplicações que podem se beneficiar do coletor paralelo são aquelas que possuem poucas restrições em relação ao tempo das pausas. Isso acontece porque as coletas na geração estável, mesmo que pouco freqüentes, ainda consomem tempo considerável devido à grande quantidade de memória a ser analisada. É importante observar que a utilização do coletor paralelo em máquinas com apenas um processador irá diminuir o desempenho em relação ao coletor serial, já que existirá uma disputa por processamento entre as *threads* e mecanismos de sincronizações desnecessários.

3.3.3. O coletor concorrente

O coletor concorrente, ou coletor de baixa latência (*low-latency*), como também é conhecido, foi desenvolvido para aplicações que necessitam de pausas pequenas para a coleta de lixo e que podem compartilhar recursos de processamento com o coletor

enquanto a aplicação está em execução. Normalmente, as coletas na geração jovem não causam longas pausas. Entretanto, mesmo que infreqüentes, as limpezas na geração estável implicam em longas interrupções da aplicação, especialmente quando grandes *heaps* estão envolvidas.

Similar aos outros mecanismos apresentados, o coletor concorrente também utiliza coletas por gerações. Ele tenta reduzir o tempo de limpeza para a geração estável através da criação de *threads* para percorrer e identificar o lixo concorrentemente com a execução da aplicação.

Em PRINTEZIS (2004) é apresentado o funcionamento deste mecanismo. Durante essas limpezas, o coletor irá interromper a aplicação por um pequeno período no começo da coleta e novamente no meio da mesma. A primeira pausa é chamada de marcação inicial (*initial mark*), onde o coletor identifica o conjunto inicial de objetos vivos diretamente alcançáveis pela aplicação (conjunto raiz). Logo após esta pausa, na fase de marcação concorrente, o coletor identifica todos os objetos vivos que são transitivamente alcançáveis a partir do conjunto raiz. Como a aplicação continua executando, ela pode criar ou modificar referências durante a fase de marcação. Neste caso, não é garantido que todos os objetos vivos tenham sido identificados ao final da marcação concorrente. Para solucionar este problema, a aplicação é interrompida novamente em uma segunda pausa, chamada de remarcação (*remark*), que tende a ser mais longa do que a primeira. Durante esta etapa, todos os objetos que foram modificados na marcação concorrente são analisados novamente. Para evitar que isso seja muito demorado, esta etapa é realizada em paralelo, utilizando todos os processadores disponíveis. Ao final da remarcação, todos os objetos vivos foram identificados e o coletor vai para a etapa final, que é a varredura concorrente, onde o lixo identificado é liberado enquanto a aplicação executa. A figura 3.3 representa uma linha de tempo destacando as diferenças entre as coletas de gerações estáveis de um coletor serial e do coletor concorrente da *HotSpot* JVM.

Algumas tarefas, com revisitar objetos durante a fase de remarcação incrementam o trabalho a ser feito, e em conseqüência, produzem um *overhead*. Esse é um impasse que coletores que visam diminuir os tempos de pausa precisam lidar. Para compensar um pouco a perda desse tempo, o coletor concorrente não realiza compactação da memória após a limpeza. A figura 3.4 representa a memória na geração estável antes e após a limpeza.

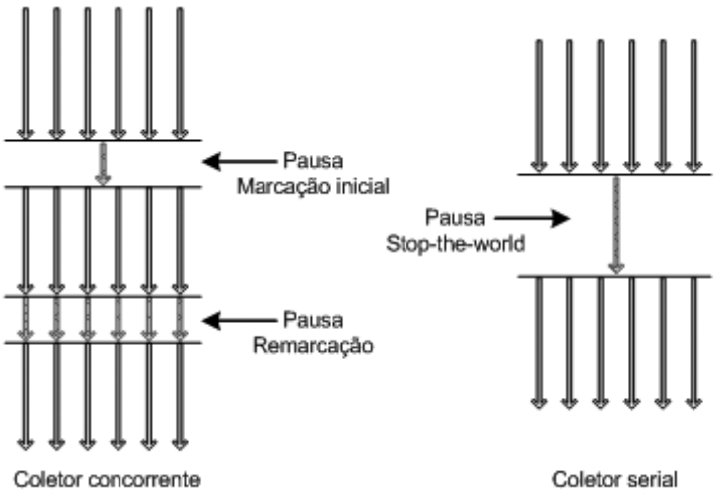

Figura 3.3 – Comparação entre um coletor serial e coletor concorrente na JVM

Em 3.4 a) temos a memória com todos os objetos logo após da fase de remarcação. Os objetos com um 'X' representam o lixo e devem ser eliminados. Em 3.4 b) vemos a memória após a limpeza, com fragmentação.

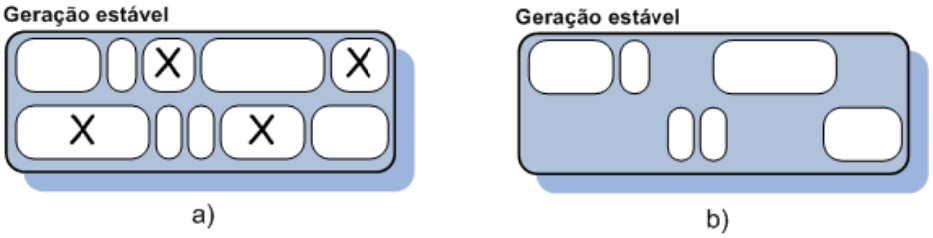

Figura 3.4 – Memória na geração estável antes a) e depois b) de uma coleta concorrente

É possível observar que nesta situação uma nova alocação de memória não pode ser realizada simplesmente com um incremento de ponteiro. Neste caso, é necessária a criação de listas de espaços livres, ligando as regiões disponíveis. Toda vez que um objeto precisar ser alocado, esta lista é percorrida até que um espaço apropriado seja encontrado, de acordo com a política de alocação. Essa dificuldade pode impactar nas limpezas da geração jovem, já que quando um objeto for promovido à geração estável, a sua alocação irá demorar mais, aumentando assim o tempo das coletas menores.

Outra desvantagem do coletor concorrente é a necessidade de *heaps* maiores do que para os outros mecanismos. Isso acontece porque, como a aplicação continua em execução durante o processo de limpeza da geração estável, novos objetos podem ser alocados na memória, resultantes de coletas na geração jovem ou mesmo através do código da aplicação. Além disso, embora o coletor sempre identifique todos os objetos vivos, alguns

desses podem se tornar lixo logo após sua análise e não serão liberados até a próxima coleta na geração. Esses objetos são conhecidos como lixo flutuante (*floating garbage*) (SUN, 2006).

Diferentemente dos outros coletores, o coletor paralelo não inicia a limpeza quando a geração fica sem espaço. Ao invés disso, as coletas precisam começar cedo o suficiente para que possam ser completadas antes que isso ocorra. Caso uma falha de alocação aconteça por falta de espaço durante a execução de uma limpeza, a aplicação é interrompida e a coleta é completada em paralelo com todas as *threads* disponíveis.

Em resumo, comparado com o coletor paralelo, o coletor concorrente diminui as pausas para limpezas em gerações estáveis, algumas vezes drasticamente; ao custo de um aumento de tempo nas coletas menores, uma redução do *throughput* e a necessidade de mais espaço para a *heap*.

RAMAKRISHNA (2006) alerta para o fato de que, como ao menos um processador é utilizado para a limpeza de lixo durante as fases concorrentes, esse coletor normalmente não provê nenhum benefício para uma máquina com apenas um processador. Entretanto, existe um modo disponível que permite pausas menores nesses sistemas, chamado de modo incremental.

A idéia para diminuir o impacto de longas pausas no modo incremental é interromper o coletor na sua fase concorrente e devolver os recursos de processamento à aplicação por um intervalo de tempo. O trabalho feito pelo coletor é dividido em pequenos blocos de tempo que são escalonados entre as coletas das gerações jovens. Assim, a cada coleta menor, uma parte da coleta na geração estável é realizada.

Tipicamente, aplicações que manipulam grande quantidade de dados, executam em máquinas com vários processadores e precisam de tempo de pausa pequenos são as mais indicadas para obterem benefícios deste coletor, que pode ser ativado através da opção `-XX:+UseConcMarkSweepGC`. Para ativar o modo incremental é necessário adicionar a opção `-XX:+CMSIncrementalMode`.

3.4. Seleções automáticas e otimizações

No J2SE versões 5.0 e 6.0 é possível definir uma série de parâmetros que irão interferir no desempenho dos coletores, como veremos adiante. Os valores padrões para as propriedades dos coletores são definidos automaticamente com base na plataforma e no sistema operacional onde a *HotSpot* JVM está em execução. Para que isso seja possível, a JVM

analisa o computador e define a máquina virtual como sendo uma máquina cliente ou servidora. Máquinas servidoras são aquelas que executam em computadores com dois ou mais processadores físicos e dois ou mais *gigabytes* de memória física principal. As configurações de hardware inferiores definem a JVM como cliente. A principal diferença entre ambos está no compilador dos *bytecodes*[2] utilizados pela JVM. O compilador cliente tem como objetivo minimizar o tempo de início do aplicativo efetuando menos otimizações, enquanto que o compilador servidor tem como objetivo maximizar a execução do aplicativo.

As seleções automáticas atendem à maioria das aplicações, porém a *HotSpot* JVM fornece opções para o usuário especificar alguns comportamentos desejados que podem ser específicos para uma aplicação.

Para que o desenvolvedor possa mensurar informações sobre a coleta de lixo em sua aplicação, a máquina virtual disponibiliza algumas opções que trazem características sobre o processo de limpeza. A opção `-verbose:gc` exibe algumas informações de cada coleta. Para mais dados, podemos utilizar a opções `-XX:+PrintGCDetails`, que informa detalhes do processo de limpeza e `-XX:+PrintGCTimeStamps`, que informa o tempo de aplicação quando do início da limpeza.

Um primeiro conceito importante está relacionado ao tamanho da *heap.* Para isso, dois valores são considerados na alocação de memória: o tamanho inicial e o tamanho máximo. Na inicialização da máquina virtual, a quantidade de memória especificada no tamanho máximo é reservada pelo SO para uso. Entretanto, se o tamanho inicial for menor do que o tamanho máximo, a *HotSpot* JVM não irá efetivar o bloqueio de toda a memória, mas somente do indicado. O restante será chamado de espaço virtual. Esse procedimento ocorre para que a memória não seja toda bloqueada, antes mesmo de ser necessária. É possível especificar os valores de tamanho inicial e tamanho máximo da *heap* através das opções `-Xms` e `-Xmx` respectivamente. Em máquinas não servidoras (clientes), o tamanho inicial padrão é 4MB e o tamanho máximo da *heap* é 64MB. Para máquinas servidoras, o valor inicial é dado por $1/64$ da memória física disponível e o tamanho máximo é $1/4$, ambos não excedendo um *gigabyte*.

Como as limpezas ocorrem quando as gerações ficam sem espaço, o *throughput* é inversamente proporcional à quantidade de memória disponível. Isso faz com que o total

[2] *Bytecode* é a representação de um código Java através de um arquivo .class.

de memória seja um dos mais importantes fatores que afetam o desempenho dos coletores de lixo.

A JVM permite o aumento ou diminuição da parte bloqueada da *heap* a cada coleta, na tentativa de manter uma proporção entre o espaço livre e o ocupado. Essa proporção pode ser especificada pelo usuário através das opções `-XX:Min*Heap*FreeRatio=<mínimo>` e `-XX:Max*Heap*FreeRatio=<máximo>`, onde os valores mínimo e máximo variam de 0 a 100 e representam a porcentagem de espaço utilizado em cada geração. Eles são aplicados independentemente em cada geração. Assim, se o mínimo tem valor 30 e, ao término de uma limpeza, o espaço livre em uma geração diminuir para menos que 30% do total bloqueado para esta geração, o tamanho da mesma é incrementado, consumindo-se o espaço virtual, para que se alcance 30% de espaço bloqueado livre. Similarmente, se o valor máximo definido for 70 e houver mais do que 70% de espaço livre na geração, o tamanho da mesma é reduzido para tentar atingir o objetivo. Os valores *default* giram em torno de 30 – 40 para o mínimo e 60 – 70 para o máximo, dependendo do SO e da plataforma.

Um dos problemas com estes valores é a partida lenta, já que inicialmente a *heap* é muito pequena e precisa ser redimensionada muitas vezes até se estabilizar. Uma tentativa de solucionar isso é manter disponível o máximo de memória possível para a máquina virtual, mas isso pode afetar o tempo de pausa durante as limpezas. (SUN, 2006)

A divisão da memória entre as gerações da *heap* é definida através de proporções, que também podem ser modificadas pelo usuário. A figura 3.5 exemplifica uma divisão de espaços da *heap* entre gerações.

A proporção da *heap* dedicada à geração jovem é outro fator de grande influência. Quanto maior a geração jovem, menos coletas menores irão ocorrer. Se esse valor crescer muito, a geração estável tende a ficar pequena, o que indica que mais coletas maiores irão acontecer. A melhor escolha depende da distribuição de tempo de vida dos objetos alocados pela aplicação.

Para poder modificar essa proporção, existe a opção `-XX:NewRatio=<valor>`, onde o valor representa a razão entre a geração estável e a geração jovem. Em outras palavras, um valor 3 indica que para cada parte de memória da geração jovem existem três partes na geração estável, ou seja, a geração jovem possui $1/4$ do total da *heap*, com exceção da área permanente.

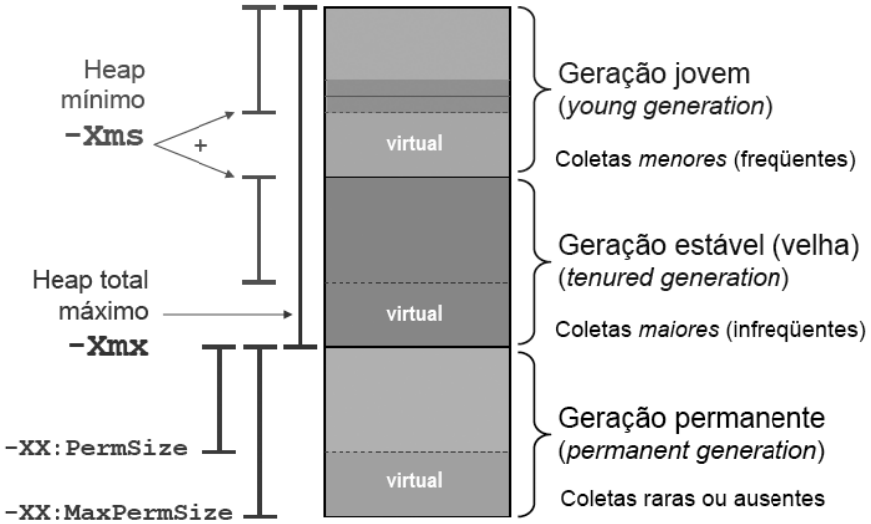

Figura 3.5 – Divisão de memória entre gerações na *heap*

Outras duas opções, `-XX:NewSize=<valor>` e `-XX:MaxNewSize=<valor>` também podem ser utilizadas para definir uma valor inicial e um valor máximo para a geração jovem. Se os dois valores forem iguais significa que o tamanho é fixo.

Uma característica a ser lembrada é que, quando ocorre uma coleta menor, vários objetos da geração jovem são promovidos à geração estável. Para garantir que esse procedimento possa ocorrer sem problemas, a JVM não permite que o tamanho da geração jovem seja maior do que metade de todo o tamanho da *heap* (MASAMITSU, 2005b). Com isso, se metade da *heap* está alocada para a geração jovem, é possível garantir que ao final de uma coleta menor, todos os objetos promovidos poderão ser alocados na geração estável, que ocuparia a outra metade; isso imaginando uma situação onde todos os objetos sofram a promoção.

Assim como é possível dimensionar a geração jovem, é possível dimensionar o espaço de sobreviventes em relação ao éden. Isso é feito através da opção `-XX:SurvivorRatio=<valor>`, que representa a razão entre o éden e um dos dois espaços de sobreviventes. Ou seja, para um valor oito, significa que cada espaço de sobrevivente terá o mesmo tamanho que do éden. Se o espaço de sobreviventes for muito pequeno, a maioria dos objetos será alocada diretamente na geração estável, causando mais coletas maiores. Se ele for muito grande, ele pode não ficar cheio e o espaço será desperdiçado.

A cada coleta, a máquina virtual seleciona um valor base que indica o número de coletas que cada objeto deve sobreviver antes que ele seja elevado à geração estável. Por padrão, esse valor é ajustado dinamicamente com o intuito de manter a metade do espaço

de sobreviventes sempre cheia. É possível definir manualmente essa porcentagem de ocupação, para que com base nesta nova porcentagem, o número de cópias possa ser definido. Isso é feito através da opção `-XX:TargetSurvivorRatio=<porcentagem>`. Também é possível definir um valor base máximo através da opção `-XX:MaxTenuringThreshold=<n>`, que indica que qualquer objeto não pode ser copiado de um espaço de sobreviventes para o outro mais do *n* vezes.

Uma última opção para gerenciamento do tamanho da *heap* é a definição de um tamanho inicial e um tamanho máximo para a geração permanente, através das opções `-XX:PermSize=<valor>` e `-XX:MaxPermSize=<valor>`, respectivamente.

SUN (2006) apresenta a tabela 3.1, que contém os valores padrões para algumas das opções apresentadas anteriormente em máquinas virtuais servidores e clientes. Esses valores estão com base no sistema operacional Solaris 32 bits em uma plataforma SPARC. Esses valores podem sofrer alterações para outras plataformas e/ou SO.

Tabela 3.1 – Valores padrão para a JVM rodando no Solaris em plataforma SPARC

PARÂMETRO	JVM CLIENTE	JVM SERVIDOR
NEWRATIO	8	2
NEWSIZE	2228Kb	2228Kb
MAXNEWSIZE	ilimitado	ilimitado
SURVIVORRATIO	32	32

FONTE: SUN (2006).

Após o conhecimento de algumas opções de modificação da *heap* na máquina virtual, vamos apresentar algumas opções específicas para cada coletor implementado na mesma.

Para o coletor paralelo, uma primeira mudança pode ser efetuada no número de *thread*s que irão realizar a limpeza. Em um sistema com *n* nós de processamento, pode ser interessante que nem todos participem de um processo de limpeza. Por padrão, o coletor paralelo cria *n threads* para coleta de lixo. Esse número de *threads* pode ser modificado através da opção `-XX:ParallelGCThreads=<numero>` na linha de comando. A máquina virtual não impede que o usuário defina mais *threads* do que o máximo de processadores disponíveis, mas essa atitude tende a ser prejudicial, já que mais *thread*s do que

processadores provoca uma disputa por processamento, resultando em freqüentes trocas de contexto por parte do SO.

Para este coletor, é possível definir alguns objetivos que o mesmo deve atingir. O primeiro deles é um valor máximo de pausa. Por padrão esse valor é ilimitado, mas através da opção `-XX:MaxGCPauseMillis=<valor>` é possível definir, em milissegundos, um tempo máximo que a aplicação ficará em pausa para processos de limpeza.

Se um tempo máximo de pausa é especificado, o coletor ajusta seus parâmetros numa tentativa de manter as pausas menores que o valor estipulado. Para isso, ao final de cada coleta, a média do tempo de pausa é atualizada pelo coletor. É então realizado o teste para verificar se o objetivo foi atingido e as modificações necessárias são feitas. Essas mudanças são realizadas dinamicamente, e podem ocasionar uma redução no *throughput* e, dependendo dos valores e da aplicação, este objetivo pode não ser atingido.

Em casos onde a definição do tempo máximo de pausa é inatingível, o desempenho da aplicação tende a reduzir, já que a JVM reduz o tempo de pausa diminuindo o tamanho da *heap*. Reduzir muito esse tamanho causa um aumento na freqüência das coletas e o *throughput* tende a cair. (MASAMITSU, 2005a)

Quando o tempo máximo de pausa não é atingido, somente uma geração tem o seu tamanho modificado por coleta. Se o tempo de pausa não for atingido em ambas as gerações, a geração que teve a maior pausa tem seu tamanho modificado primeiro.

Um segundo comportamento que pode ser indicado é um valor mínimo de *throughput*, que é o tempo gasto em limpezas *versus* o tempo da aplicação. O valor pode ser informado através da opção `-XX:GCTimeRatio=<N>`, que define a razão do tempo de coleta pelo tempo da aplicação para $\frac{1}{(1+n)}$. Em outras palavras, um $n = 19$ define uma meta de tempo total de coleta de lixo de $1/20$, ou 5% do tempo total de execução. O tempo utilizado para o cálculo é o tempo total para todas as gerações. Se o objetivo não é atingido o tamanho da geração é incrementado em uma tentativa de manter a aplicação rodando mais tempo entre duas limpezas consecutivas, já que uma geração maior demora mais tempo para esgotar o espaço. O valor padrão é 99, resultando em um objetivo de no máximo 1% de tempo em limpezas.

Quando o objetivo do *throughput* não é atingido, ambas as gerações têm seu tamanho incrementado. A quantidade é proporcional à contribuição de tempo da mesma no tempo total de limpeza. Ou seja, se o tempo de coleta da geração jovem representa 25% do

tempo total de coleta da aplicação, e o incremento definido para esta geração seja de 20%, então a geração jovem será incrementada em 5% (25% de 20%) de seu tamanho total.

Se o *throughput* e o tempo máximo de pausa forem atingidos, o coletor pode trabalhar para atingir um menor tamanho de *heap* possível, conhecido como *footprint*. Para isso, basta reduzir o tamanho da *heap* até o tamanho mínimo onde os dois objetivos anteriores sejam alcançados. É importante ressaltar que o coletor tenta primeiramente atingir o valor máximo de pausa e só então trabalha para atingir o *throughput*. O *footprint* só é considerado depois que os dois anteriores forem satisfeitos (RAMAKRISHNA, 2006).

A cada incremento ou decremento no tamanho da *heap*, o valor de mudança é calculado com base nas informações das coletas que já ocorreram. Para uma primeira limpeza, esse valor é padrão: incremento de 20% e decremento de 5% do tamanho total. Se o usuário desejar, é possível definir estes valores para incrementos das gerações jovem e estável através das opções `-XX:YoungGenerationSizeIncrement=<n>` e `-XX:TenuredGenerationSizeIncrement=<n>`, respectivamente. O valore *n* representa a porcentagem de acréscimo em função do tamanho atual da geração. Para o decréscimo, o valor definido é dado em função do valor de acréscimo escolhido para cada geração. A opção `-XX:AdaptiveSizeDecrementScaleFactor=<d>` modifica um fator de decréscimo *d* que define a porcentagem de redução em x/d, onde x é a porcentagem de acréscimo definida para a geração.

No coletor concorrente, uma das informações importantes é saber quando iniciar o processo de limpeza para que o mesmo possa terminar antes que a geração fique sem espaço e ocorra uma falha de alocação de memória. Para isso, a máquina virtual mantém estimativas de tempo restante para que a geração fique cheia e o tempo necessário para se realizar uma coleta. Assim é possível determinar o momento certo de iniciar o processo. Este momento é definido por um valor de ocupação da geração, que inicialmente é definido em 92% da geração. Quando este valor for ultrapassado, um processo de limpeza é automaticamente iniciado. Ao término da coleta, esse valor é atualizado pela JVM na tentativa de aperfeiçoar o desempenho. Também é possível definir um valor de ocupação inicial para a geração pela opção `-XX:CMSInitialOccupancyFraction=<N>`, onde N representa a porcentagem de ocupação da geração.

Quando o coletor concorrente está com o modo incremental ativado, o mesmo mantém um ciclo de trabalho (*duty cycle*) para controlar a quantidade de trabalho que o coletor deve executar antes de devolver o processamento à aplicação. Esse valor é um

percentual do tempo entre as coletas na geração jovem e que também pode ser definido pelo usuário pela opção `-XX:CMSIncrementalDuityCycle=<N>`. Por padrão, a *HotSpot* JVM define este valor como 50% , mas este valor pode ser ajustado com base no comportamento da aplicação ao final de cada processo de limpeza. Esse mecanismo é conhecido como *automatic pacing* e pode ser ativado ou desativado com a opção `-XX:+CMSIncrementalPacing`. Quando esse mecanismo está ativado, é possível definir um limite inicial inferior para este ciclo de trabalho através da opção `-XX:CMSIncrementalDutyCycleMin=<N>`.

Outro parâmetro é um fator de segurança, que serve como margem para o início de uma coleta incremental. Esse parâmetro é utilizado para casos onde as taxas de alocação de memória nas gerações estáveis não são constantes e a análises de coletas anteriores pode levar a falsas predições. Com esse fator, a máquina virtual pode iniciar uma coleta antes do momento por ela mesma calculado. Para modificar esse valor é necessário utilizar a opção `-XX:CMSIncrementalSafetyFactor=<N>`, onde N é a porcentagem de segurança. Um valor de N = 10 indica que a próxima coleta iniciará com 10% de antecedência do momento planejado e um valor de N = 100 representa o inicio da mesma assim que possível, mesmo que não seja necessário (MASAMITSU, 2006a)

A máquina virtual utiliza os dados das coletas anteriores para manter médias estatísticas dos processos que serão utilizadas para decisões futuras. Para casos onde as alocações de memória são muito inconstantes, esses dados podem não ser confiáveis e levar à falsas inferências. Nestes casos, pode ser interessante definir um valor de peso para a influência das informações da última coleta no cálculo dos mesmos. A JVM permite que isso seja feito através da opção `-XX:CMSExpAvgFactor=<N>`, onde N representa a porcentagem de peso dos dados da última coleta.

Com o conhecimento dos coletores existentes para a JVM e ciente das possibilidades de modificações nas opções dos mesmos, o próximo capítulo apresenta os resultados das execuções de várias aplicações utilizando diferentes configurações entre coletores e opções.

Capítulo 4

Avaliação de desempenho

Neste capitulo são apresentados os resultados de desempenho dos três algoritmos de coleta de lixo da JVM inicialmente descritos no capitulo 3, utilizando um conjunto de testes composto por oito aplicações representativas de uma grande variedade de padrões de alocação/uso de objetos presentes nas mais variadas e representativas classes de aplicações.

O restante deste capítulo está dividido como se segue. Inicialmente são apresentados os objetivos da avaliação (seção 4.1). O ambiente experimental (seção 4.2) e as aplicações que integram o conjunto de testes (seção 4.3) são descritas em seguida. Os resultados obtidos pelas execuções do conjunto de testes utilizando os três algoritmos de coleta de lixo e suas variações, bem como as análises dos mesmos são então apresentadas (seção 4.4), organizadas por aplicação. Para a finalização do capítulo estão indicados os resultados gerais das execuções vista pelos algoritmos de coleta de lixo e suas variações (seção 4.5) e algumas sugestões para modificações nas opções existentes na JVM (seção 4.6).

4.1. Objetivos

O objetivo inicial é obter uma visão completa sobre a execução de todos os coletores para cada aplicativo. Com essas informações é possível avaliar se os mesmos se comportaram como seria teoricamente previsto, levando em consideração as características da aplicação e do coletor utilizado.

Sendo assim, o conjunto inicial de execuções é composto por 15 configurações distintas entre os três coletores padrões e algumas opções disponíveis na JVM. A tabela 4.1 apresenta estas configurações. Os números de 1 a 15 na coluna da direita servirão como referência para as próximas tabelas e figuras deste trabalho.

A escolha deste conjunto de execuções foi realizada com o intuito de cobrir todos os mecanismos de coleta de lixo disponíveis na JVM e que foram abordados no capítulo 3. No modo cliente, as opções de coleta em paralelo foram avaliadas para coleta somente na geração jovem (configuração número 3, apresentada na tabela 4.1), somente na geração estável (4) e para ambas as gerações (2). O modo servidor já inclui o paralelismo na

geração jovem por padrão. Podemos, entretanto, utilizar paralelismo neste modo também na geração estável (configuração 11).

Os coletores CMS (*Concurrent Mark-Sweep*) são, por sua natureza, coletores paralelos e foram avaliados nos modos padrão (5 e 12), com remarcação em paralelo (6 e 13) e com paralelismo somente na geração jovem (8 e 15). O modo incremental, desenvolvido para máquinas com um único processador, também foi avaliado nas configurações 7 e 14.

Tabela 4.1 – Possíveis configurações entre coletores e opções na JVM. Os números identificam as configurações utilizadas.

MODO JVM	COLETOR	OPÇÃO	CONF
-client	*default* (-XX:+UseSerialGC)	-	1
	-XX:+UseParallelGC	-	2
	-XX:+UseParNewGC	-	3
	-XX:+UseParallelOldGC	-	4
	-XX:+UseConcMarkSweepGC	*default*	5
		-XX:+CMSParallelRemarkEnabled	6
		-XX:+CMSIncrementalMode	7
		-XX:+UseParNewGC	8
-server	*default* (-XX:+UseParallelGC)	-	9
	-XX:+UseSerialGC	-	10
	-XX:+UseParallelOldGC	-	11
	-XX:+UseConcMarkSweepGC	*default*	12
		-XX:+CMSParallelRemarkEnabled	13
		-XX:+CMSIncrementalMode	14
		-XX:+UseParNewGC	15

Com as informações obtidas através da execução de cada aplicação nas combinações listadas acima, deseja-se responder a quatro perguntas:

1) Os coletores se comportaram como seria teoricamente esperado, levando em consideração a característica da aplicação?
2) Para cada aplicação, qual coletor utilizou a memória da maneira mais eficiente?
3) Para cada aplicação, qual coletor interferiu menos no tempo de execução?
4) No geral, qual o melhor coletor cada classe de aplicações?

As avaliações levaram em consideração principalmente dois quesitos: o *footprint*, que representa o maior tamanho atingido pela *heap* e o *throughput*, que é a porcentagem do

tempo total que não é gasto com coleta de lixo. Foram considerados ainda outros indicadores, como o tempo total de aplicação, tempo de pausas acumulados, tempo médio de pausa e os tempos da menor e maior interrupção.

4.2. O ambiente experimental

Todos os experimentos foram executados em uma mesma máquina Dell com dois processadores Intel Xeon 5110 64 bits (*dual core)*, totalizando assim quatro núcleos de processamento, cada um com *clock* de 1.6GHz, 4096KB de memória *cache* L2 e 4GB de memória principal. O sistema operacional utilizado é o SuSE Linux versão 2.6.5-7.283-smp. A versão do Java utilizada é a 1.5.0 *update* 14 (1.5.0_14-b03).

Para que fosse possível mensurar as informações necessárias foi utilizado o sistema de *log* dos mecanismos de limpeza já existente na máquina virtual (SUN, 2003) em conjunto com o GCViewer (GCVIEWER, 2006), um aplicativo *open source* para visualização gráfica de dados obtidos pelas saídas geradas pelas opções de execução `-XX:+PrintGCDetails`, `-XX:+PrintGCTimeStamps` e `-Xloggc:<file>`, todas disponíveis na máquina virtual.

Para cada configuração, as aplicações foram executadas ao menos cinco vezes, no intuito de verificar a variância dos tempos de execução e, consequentemente, garantir a confiabilidade dos resultados. Os parâmetros e dados de entradas são os mesmos para todas as execuções e serão apresentados na seção seguinte. Alguns aplicativos demandaram modificação no tamanho padrão máximo da *heap* para que pudessem ser executados.

4.3. O conjunto de avaliação

Os aplicativos utilizados para as avaliações foram retirados de *benchmarks* distintos e procuram satisfazer aos diferentes comportamentos de alocação e utilização de memória pelas aplicações.

O primeiro aplicativo do *benchmark* utilizado é o GCBench. Ele é uma adaptação do *benchmark* originalmente escrito por John Ellis em parceria com Pete Kovac, e posteriormente modificado por Hans Boehm.

Em sua fase inicial, um processo de alongamento da *heap* é realizado, com a alocação e liberação imediata de uma árvore. Esse comportamento deve ser acompanhado de várias coletas, sendo que após algumas coletas menores, uma coleta maior deve ser executada. Isso acontece porque, inicialmente, tanto a *heap* como a gerações jovem e

estável são pequenas. Ao preencher completamente a geração jovem, os dados são movidos para a geração estável, que nesta fase inicial será logo saturada, e uma coleta maior será necessária. É durante essas coletas maiores que o tamanho da *heap* é modificado para permitir novas alocações de dados. Após esta fase inicial, espera-se que a *heap* e a geração estável sejam grandes o suficiente para que o aplicativo possa criar e destruir árvores binárias balanceadas de diferentes tamanhos sem que existam grandes modificações na *heap*, resultando em seqüencias de coletas menores e maiores decorrentes somente do lixo produzido. Durante esta segunda fase, as árvores pequenas resultam em objetos com tempo de vida curtos, e logo serão retiadas da memória pelo coletor. As árvores de tamanho maior fazem com que os objetos sobrevivam a algumas limpezas e sejam promovidos à geração estável, forçando algumas varreduras completas na *heap*. É importante observar que o tamanho das árvores cresce ao passar do tempo e as coletas tendem a serem maiores ao final da execução.

A aplicação mantém duas estruturas de dados ativas na memória: uma árvore binária contendo vários ponteiros e um grande vetor de elementos de ponto flutuante com dupla precisão (*double).* Isso caracteriza aplicações que armazenam alguma quantidade de dados durante todo o processo.

Este aplicativo aceita como parâmetros as profundidades das árvores de alongamento da *heap* e da que será mantida ativa na memória, o tamanho do vetor de elementos numéricos e os limites inferior e superior para as profundidades das várias árvores que são construídas e destruídas. Para as avaliações foram utilizadas as entradas padrão sugeridas pelo autor do aplicativo: uma árvore de 20 níveis de profundidade para o processo de alongamento da *heap*; uma árvore com 18 níveis de profundidade e um vetor com 5.000.000 posições que serão mantidos em memória; e testes com árvores de profundidade entre 4 e 24 níveis, considerando apenas os valores pares.

O GCOld (DETLEFS, 2003) é o segundo aplicativo do *benchmark* utilizado. Ele parte do pressuposto que, para algumas aplicações, a maioria dos objetos não vive por um período pequeno de tempo. No caso de servidores de aplicações *web*, que podem realizar interações com o usuário em escala de minutos, a alocação de dados é realizada no começo do processo e persiste durante o mesmo. Se essa duração é suficientemente similar, então, o comportamento da memória é como uma fila: o dado mais antigo é o que tem a maior chance de se tornar lixo.

A execução consiste em uma fase de inicialização e em uma fase estável. O programa mantém um vetor de ponteiros para raízes de árvores binárias. A fase de

inicialização consiste na alocação das árvores e na inicialização do vetor. A fase estável é composta por um *loop* de passos, com cada iteração realizando: a) a alocação de alguma quantidade de objetos de vida curta e que se tornam lixo ao fim do passo; b) a alocação de uma porção de árvore binária que, quando concluída, irá substituir uma parte de alguma árvore já existente (fazendo a anterior se tornar lixo); c) algumas mudanças de ponteiros das árvores (que podem se encontrar na geração estável e influenciar no mecanismo de limpeza com referências entre distintas gerações) e d) algumas modificações de valores nos dados, conhecida por mutações.

A aplicação aceita como parâmetro o tamanho, em *megabytes*, dos objetos que ficarão armazenados na memória; o número de mutações, que são as responsáveis pelas modificações na estrutura de dados (podem impactar na taxa de promoção dos objetos para a geração estável, já que mais mutações por passo implicam em menos passos por segundo, logo menos promoções por segundo), a proporção de objetos que permanecerão na memória em relação aos objetos que tornarão lixo logo após sua criação, o número de mudança em ponteiros por passo e, finalmente, o número de passos que serão executados. Foi utilizado como entrada para os testes 64MB de estrutura a ser mantida viva, 10 mutações por passo, 2 como proporção de objetos a serem promovidos para a geração estável (50%), 300 mudanças em ponteiros por passo em um total de 1000 passos.

O terceiro aplicativo do *benchmark* é o GeneId (GENOME, 2002), uma aplicação para prever genes em seqüências genômicas anônimas, concebidas com uma estrutura hierárquica. Escrito por Enrique Blanco, Tyler Alioto e Roderic Guigó, o programa recebe como entrada uma seqüência de DNA em um arquivo texto no formato FASTA[3] e um arquivo com opções na extensão `.params`. Cada filo[4] possui um arquivo `.params` diferente. Para esta aplicação foram utilizadas informações genéticas da espécie *drosophila*, tando no arquivo FASTA quanto no arquivo `.params`.

Este algoritmo armazena grande quantidade de caracteres em vetores, representando as seqüências de nucleotídeos. Esse procedimento é realizado em uma fase de inicialização, com muita alocação de memória. Durante o restante da execução, objetos são criados e destruídos com freqüências diferentes. Esse aplicativo foi inicialmente

[3] FASTA é um formato de texto específico para representar cadeias de DNA onde se utiliza os caracteres A, T, G e C para indicar as bases nitrogenadas

[4] Os filos são os agrupamentos mais elevados geralmente aceitos em cada um dos reinos em que os seres vivos foram divididos, tendo em conta, para esta aplicação, os seus traços genéticos (filogenéticos)

concebido para a linguagem C, mas foi utilizada uma versão paralelizada em Java, que será avaliada nas execuções.

Os próximos aplicativos pertencem ao Java Grande *Suite Benchmarkm* (EPCC, 2007), que possui a versão serial e a *multithread*. O aplicativo JGFCreateBench, da suíte serial, é responsável por testar alocação de diferentes tipos de objetos. Para isso, *loops* com várias alocações para um mesmo ponteiro são invocados, resultando somente em objetos com vida extremamente curtas que não devem sobreviver a uma primeira limpeza da memória. O código do aplicativo foi modificado para reduzir o tempo de execução, já que, como o *loop* de execução é o mesmo, não há necessidade de realizar testes muito longos. Este aplicativo não aceita parâmetros e aloca na casa de 10^8 objetos de diferentes tipos de dados para as modificações realizadas.

O aplicativo JGFSerialBench, ainda na suíte serial, é o responsável por avaliar a criação e leitura de arquivos de dados em disco. Para isso são construídas estruturas como árvores, *LinkedLists*, *Arrays* e *Vectors*[5] que são gravados em arquivos e depois carregados novamente em memória. Para cada estrutura diferente são realizadas loops de 4000 iterações gravando e lendo os arquivos. O consumo de memória deve ser constante para cada tipo de estrutura presente no mesmo. Este aplicativo também não aceita parâmetros.

O aplicativo JGFForkJoinBench é um aplicativo da suíte *multithread*, é realiza testes de criação e junção de *threads* em Java. A execução é bastante simples. São criadas na casa de oito milhões de *threads*, todas elas com o mesmo objetivo de calcular o valor do *seno* para um ângulo qualquer. Ao final da execução, é realizado o *join* entre essas *threads*. Esse processo é repetido várias vezes até que se atinja um tempo mínimo de execução. Isso acontece porque é dependente do número de processadores da máquina. Um computador com muitos processadores iria concluir o teste mais rápido que uma máquina com apenas dois. Este aplicativo não aceita parâmetros.

O aplicativo JGFSORBench está presente nas suítes serial e *multithread*. A versão serial deste programa realiza 100 iterações do algoritmo SOR (*Successive over relaxation)*[6] em uma matriz de dados. Esta aplicação consiste em um *loop* externo e dois *loops* internos. Com o objetivo de atualizar um elemento principal da matriz, os elementos vizinhos do

[5] *LinkedLists*, *Arrays* e *Vectors* são estruturas de dados da API Java que implementam as interfaces para gerenciamento de coleções de objetos.

[6] *Successive over relaxation* é um método numérico utilizado para aumentar a convergência do método de *Gauss-Seidel* para a resolução de sistemas de equações lineares.

mesmo são necessários, incluindo os elementos previamente atualizados. Para permitir o paralelismo, o *loop* externo sobre a matriz foi distribuído entre os processadores em blocos. O programa recebeu como parâmetro de entrada uma matriz de 2000x2000 elementos representando um sistema de equações lineares. Toda a alocação de memória é realizada no início da aplicação e após essa etapa somente cálculos matemáticos são executados.

O último aplicativo, JGFMonteCarloBench, é uma simulação financeira, utilizando a técnica de Monte Carlo para definição de preços de produtos com base em fatores distintos. O código gera *n* séries temporais com as mesmas variações dos dados de uma série histórica de informações. O *loop* principal sobre os números pode ser facilmente paralelizável dividindo o trabalho em blocos independentes. A entrada para as execuções dos testes é um vetor de dados inteiros com 6000 posições.

4.4. Análises dos resultados

Nesta seção são apresentados os resultados para as execuções do conjunto de testes com as diferentes configurações entre coletores e opções. Os dados apresentados sãos os valores médios das várias execuções do mesmo aplicativo para cada configuração. Os desvios padrão dos dados foram muito pequenos. No geral, eles foram menores que 0,001 para os indicadores de tempos médias e extremos de pausas, menores que 0,1 para o tempo total de aplicação e para o *throughput* e na maioria das vezes iguais a zero para o *footprint* de memória. Sendo assim, eles não foram incluídos nas tabelas desta seção para facilitar a leitura das mesmas.

Os números na coluna mais a esquerda das tabelas desta seção correspondem às configurações da tabela 4.1. As células com o símbolo "✓" possuem os melhores resultados obtidos para cada indicador da tabela, enquanto as células com um "✖" contém os piores valores encontrados para os mesmos indicadores. Essa formatação será utilziada para as próximas tabelas e figuras desta seção.

Os resultados e as análises estão divididos em sub-seções por aplicativo, como seguem.

4.4.1. GCOld

Para o aplicativo GCOld, as execuções realizadas tiveram o tamanho máximo da *heap* definido em 256Mb. Este valor pequeno, se comparado com aproximadamente dois

gigabytes de dados que são manipulados durante toda a execução, foi escolhido para forçar a JVM a executar mais vezes o processo de limpeza.

No geral, para todas as quinze configurações distintas, o comportamento foi semelhante. No início da execução, os 64MB de dados que vão permanecer alocados durante o processo são colocados na *heap,* levando a algumas coletas que resultaram na expansão do tamanho da mesma. Após esta etapa, o padrão de comportamento do coletor se repete até a conclusão do aplicativo. Já que, segundo os parâmetros de entrada, metade dos objetos alocados em cada passo deve permanecer na *heap*, o consumo de memória vai aumentando e o tamanho da geração estável também. Durante a fase estável são alocados cerca de dois *gigabytes* de memória para todas as execuções. Coletas maiores ocorrem com grande freqüência, resultando em tempos de pausa longos e pequeno *throughput*. Essas interrupções para limpeza são ainda maiores perto do fim da execução, já que este é o ponto onde a *heap* está próxima do seu tamanho máximo.

A tabela 4.2 apresenta os valores médios obtidos para as cinco execuções desta aplicação com cada configuração entre os coletores e suas opções.

Analisando as execuções separadamente, é possível identificar algumas peculiaridades de cada algoritmo de coleta. Com o coletor padrão do modo cliente (o coletor serial, representado por 1), o tempo médio de execução ficou em 10,8 segundos e o *footprint* de memória em 224MB. 74,75% do tempo são utilizados para os processos de limpeza, o que equivale a um baixo *throughput* de 25,25%. Neste coletor, os tempos de pausa para coleta também ficaram altos, na média, 0,08 segundos, podendo chegar até 0,34 segundos, para as limpezas maiores no final da execução.

O coletor paralelo no modo cliente (representado por 2) praticamente não conseguiu reduzir o tempo acumulado de coletas, trazendo um pequeno aumento no *throughput.* O tempo de execução, tempo médio de pausa e o *footprint* permaneceram praticamente os mesmos. A opção de coleta paralela na geração estável (representada por 4) resultou em grande perda de desempenho. Para esta configuração, o tempo médio de execução subiu em aproximadamente 25% se comparado com o coletor serial. O tempo médio das pausas diminui, mas os tempos para as coletas da geração estável aumentaram, onde a maior pausa passou de 0,34 para 0,54 segundos. Isso resultou em um tempo total de limpeza muito superior ao coletor paralelo padrão fazendo *throughput* cair. Esse resultado é decorrente da fragmentação da geração estável devido ao grande número de coletas na mesma, como discutido na seção 3.3.2.

Tabela 4.2 – Valores médios obtidos na execução do GCOld para os indicadores avaliados

Heap de até 256MB				Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	25,25	224,15	8,0967	0,00771	0,34135	0,07989	10,83100
2	Cliente, Paralelo	26,22	225,46	7,9867	0,00720	0,34474	0,07897	10,82567
3	Cliente, Paralelo geração jovem	28,70	176,83	6,5767	0,00662	0,33915	0,12650	✓ **9,22400**
4	Cliente, Paralelo geração estável	21,17	227,56	✗ **10,7333**	0,00545	0,53899	0,05532	13,61500
5	Cliente, CMS	55,24	✗ **255,94**	4,5500	0,00015	0,04374	0,02935	10,16600
6	Cliente, CMS, Remarcação paralela	55,19	✗ **255,94**	4,5800	✓ **0,00013**	0,04460	0,02955	10,22033
7	Cliente, CMS, Incremental	58,40	✗ **255,94**	5,0267	0,00019	0,04316	0,03202	12,08300
8	Cliente, CMS, Paralelo geração jovem	55,30	✗ **255,94**	✓ **4,5400**	0,00014	0,04317	0,02929	10,15600
9	Servidor	26,02	226,94	8,2135	0,00065	0,35436	0,08116	11,10300
10	Servidor, Serial	✗ **21,04**	176,35	9,0633	✗ **0,01231**	0,39741	✗ **0,17426**	11,47867
11	Servidor, Paralelo geração estável	21,11	227,56	10,5868	0,00806	✗ **0,62992**	0,10079	13,43300
12	Servidor, CMS	67,66	✓ **127,94**	4,7667	✓ **0,00013**	0,03006	0,01782	14,73800
13	Servidor, CMS, Remarcação paralela	67,66	✓ **127,94**	4,8233	✓ **0,00013**	0,11799	0,01799	14,91600
14	Servidor, CMS, Incremental	✓ **70,10**	✓ **127,94**	4,8667	0,00019	0,03065	0,01815	✗ **16,27767**
15	Servidor, CMS, Paralelo geração jovem	67,57	✓ **127,94**	4,7567	✓ **0,00013**	✓ **0,02971**	✓ **0,01776**	14,66767

Dentre os coletores paralelos do modo cliente, o que se destacou com um melhor desempenho foi o coletor paralelo para a geração jovem (representado por 3). Com esta configuração, o *throughput* da aplicação subiu para 28,7%, reduzindo o tempo total de execução e o número de processos de limpeza. Como resultado, houve o aumento no tempo médio de pausa, chegando em 0,13 segundos por coleta (um aumento de 58% em relação ao coletor serial). Um destaque importante para esta configuração foi o menor consumo de memória: média de 177MB.

As avaliações dos coletores CMS no modo cliente apontaram grande eficiência na redução dos tempos de pausa e, conseqüentemente, no tempo total acumulado das mesmas, fazendo o throughput aumentar. Entretanto, todas as configurações tiveram o mesmo footprint de 255,938MB, o maior dentre os testes realizados. O coletor concorrente padrão (representado por 5), e as configurações do mesmo com a remarcação em paralelo (6) e a coleta paralela na geração jovem (8) apresentaram resultados muito semelhantes. O tempo de execução ficou em torno de 10,17 segundos, com throughput próximo de 55,25%. No geral, o coletor CMS diminuiu o tempo gasto com os mecanismos de limpeza, fazendo o tempo médio de pausa por coleta diminuir para 0,029 segundos, uma redução de quase 47% em relação ao coletor paralelo para a geração jovem, que havia apresentado o menor tempo médio de pausa até então. O principal objetivo deste coletor foi alcançado, já que nenhum processo de limpeza ultrapassou 0,045 segundos de pausa. O modo incremental (representado por 7) foi o que demonstrou os piores resultados dente os coletores CMS no modo cliente. O consumo de memória permaneceu o mesmo mas o throughput e o tempo de execução aumentaram. Esse comportamento é esperado para máquinas mutiprocessadas, já que este mecanismo foi desenvolvido para computadores com um único núcleo de processamento e realizam preempções do processo em execução.

Para o modo servidor, os coletores paralelos se comportaram de maneira distinta uns dos outros. O coletor padrão do modo servidor, o coletor paralelo (9), não obteve bons resultados. O consumo de memória (aproximadamente 227MB) e o tempo de execução ficaram altos. Ele apresentou o pior *throughput* entre todas as configurações. O mesmo aconteceu para o coletor servidor paralelo na geração estável (11), que apresentou um *throughput* bastante baixo e um tempo de execução ainda maior. Esse coletor produziu o maior tempo de interrupção para um processo de limpeza (0,62 segundos), resultando em um grande tempo acumulado de pausas para coletas. O coletor serial (10) obteve um *footprint* menor (176,35MB) e um pequeno aumento no tempo de execução. Conseqüentemente, houve uma perda no *throughput*.

Os coletores CMS (12, 13, 14 e 15) se destacaram por indicarem os maiores *throughputs* (em média, 68,25%) e os menores *footprints* (127,9MB) para este aplicativo, mas foram as execuções que mais demoraram em serem concluídas (média de 15,1 segundos). Novamente, o modo incremental teve o melhor *throughput* e o pior tempo de execução entre eles.

As figuras 4.1, 4.2 e 4.3 representam respectivamente os gráficos comparativos de *footprint, throughput* e tempo de execução para os testes realizados nesta aplicação. Os indicadores do eixo horizontal remetem o número correspondente à configuração apresentada nas tabelas 4.1 e 4.2.

Após o exame dos três gráficos e das informações apresentadas pode-se concluir que ambos os coletores CMS do modo cliente e do modo servidor foram os que apresentaram os melhores *throughputs*, representando a menor interferência no tempo da aplicação. No modo servidor, obteve-se o melhor consumo de memória ao custo dos maiores tempos de execução devido ao pequeno tamanho da *heap,* implicando em mais chamadas ao processo de coleta e gerando *overhead* resultante de constantes modificações de contexto.

O melhor coletor para esta aplicação pode ser o CMS modo servidor com coleta paralela na geração nova (15) que obteve quatro melhores indicadores dentre os sete avaliados. Se o fator tempo for determinante, a melhor opção passa a ser o coletor cliente paralelo na geração jovem (3), que obteve o menor tempo total de execução.

4.4.2. GCBench

Os testes com o aplicativo GCBench foram realizados para tamanhos máximos de *heap* de 256MB e 512MB. Durante a execução as alocações giraram em torno de dois *gigabytes* de dados. Os tamanhos diferentes foram utilizados para analisar o comportamento do coletor quando há maior e menor disponibilidade de memória.

Os padrões de comportamento dos coletores nos modos cliente e servidor foram bastante semelhantes, considerando as diferenças existentes entre cada algoritmo de coleta. No geral, para todos os coletores, o tamanho da *heap* cresceu o suficiente no início da execução. A partir daí o consumo de memória foi aumentando aos poucos juntamente com o número de coletas. Esse comportamento era esperado segundo as características já apresentadas. A tabela 4.3 contém os valores médios dos indicadores analisados nas execuções deste aplicativo.

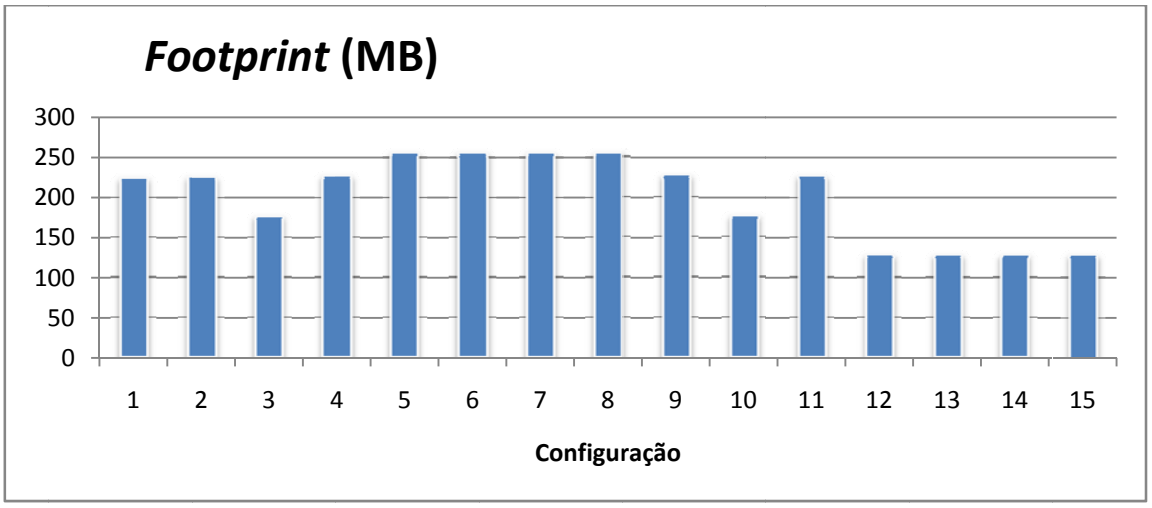

Figura 4.1 – Comparativo entre os *footprints* das execuções do aplicativo GCOld

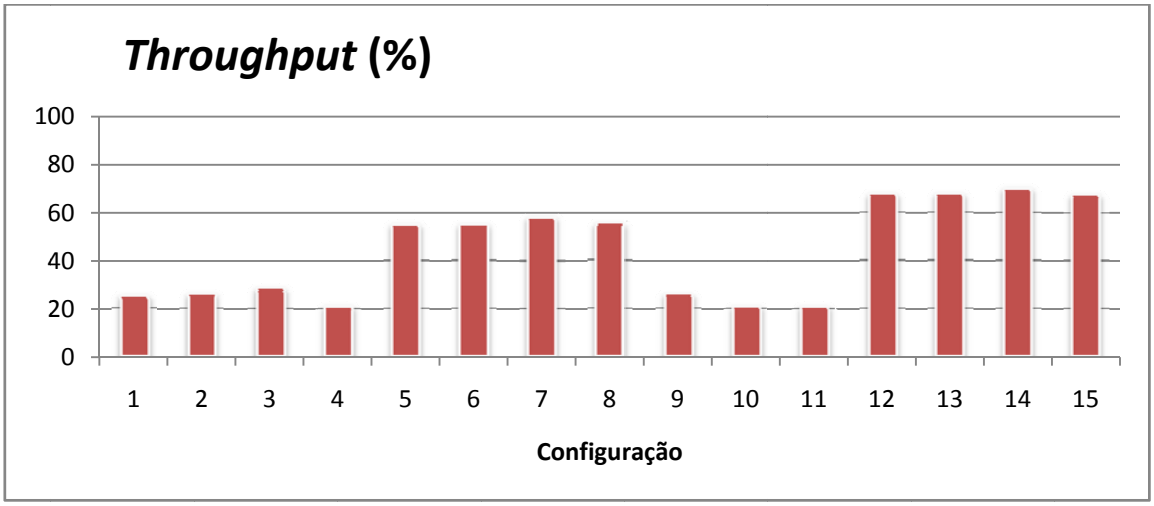

Figura 4.2 – Comparativo entre os *throughputs* das execuções do aplicativo GCOld

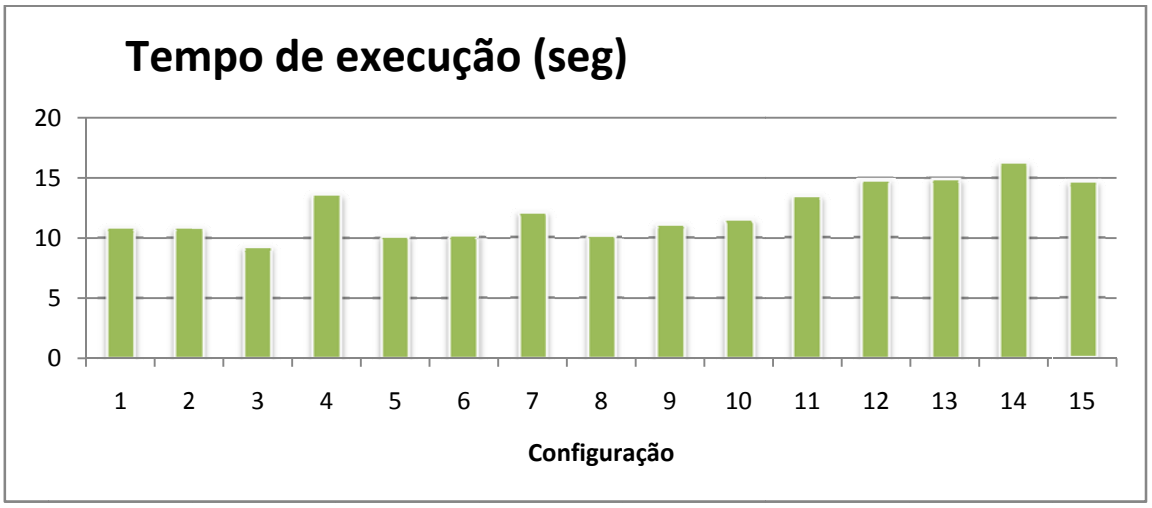

Figura 4.3 – Comparativo entre os tempos de execução do aplicativo GCOld

O coletor serial (1), padrão do modo cliente, apresentou comportamento muito semelhante para os dois tamanhos de *heap*. Com a *heap* maior, ele não fez uso da disponibilidade de memória mantendo baixo o *footprint,* já que em cada processo de coleta foi possível liberar memória suficiente para as alocações seguintes. Neste coletor, após o alongamento inicial da *heap*, a disponibilidade de espaço livre e um éden relativamente grande (cerca de 50MB) fizeram com que a maioria das alocações das árvores provocasse apenas coletas menores, com pequenos tempos de pausa. Como as árvores de testes foram ficando maiores, elas passaram a não caber mais no éden e provocaram coletas completas no final da execução, que chegaram aos 0,34 segundos para a *heap* de 256MB.

As opções no modo cliente de coleta em paralelo (2) e coleta em paralelo na geração jovem (3) também apresentaram comportamento semelhante ao coletor serial, mantendo o consumo de memória baixo para a *heap* de 512MB.

A opção com coleta estável paralela (4) teve uma queda no *throughput* e um aumento no tempo total. Para a *heap* de 256MB, este coletor apresentou tempos altos em algumas das execuções, decorrentes de uma limpeza que consumiu, em média, pouco mais de três segundos. O desvio padrão dos indicadores de tempo ficaram altos (superiores a 1,25) e, para validar estes resultados, a execução deste aplicativo com esta configuração de coletores foi executada 25 vezes, reduzindo o desvio padrão para menos que 0,5. A figura 4.4 foi extraída do *software* GCViewer e mostra a utilização da memória (para a linha) e as pausas para limpeza (para os retângulos) para uma execução do GCBench com este coletor em um tamanho máximo da *heap* de 256MB. O eixo horizontal representa o tempo de execução, enquanto o eixo vertical representa o consumo de memória (para a linha) e o tempo de pausa (para os retângulos). As linhas verticais no meio do gráfico indicam o início de um processo de limpeza completa da *heap*. É possível identificar uma grande pausa logo após a fase de inicialização, precisamente na primeira invocação do coletor após esta etapa.

A explicação desta pausa foi abordada por MASAMITSU (2006b) e está relacionada à taxa de promoção de objetos vivos da geração jovem para a geração estável. Este coletor mantém uma média de objetos que serão promovidos com base nas coletas anteriores. Se essa média exceder o espaço livre na geração estável, a próxima chamada ao coletor da geração jovem não será atendida e uma limpeza completa na *heap* será executada. A média de promoção encontrava-se alta devido à fase de inicialização. Como um processo de limpeza completo havia acabado de ocorrer e toda a geração estável foi liberada, o coletor diminuiu o tamanho da mesma para economizar memória na *heap*.

Tabela 4.3 – Valores médios obtidos na execução do GCBench para os indicadores avaliados

a) *Heap* de até 256MB

Conf.	Coletor e opções	*Throughput (%)*	*Footprint (MB)*	Tempos de pausa (seg)				Tempo total
				acumulado	menor	maior	média	
1	Cliente	37,00	227,56	3,3816	0,00082	0,37107	0,03338	5,36800
2	Cliente, Paralelo	41,80	227,56	2,7538	0,00083	0,38510	0,02720	4,73200
3	Cliente, Paralelo geração jovem	46,45	227,56	✓ 1,9213	0,00059	0,36027	0,01959	✓ 3,58800
4	Cliente, Paralelo geração estável	✗ 17,28	227,56	✗ 6,0854	0,00083	✗ 3,25685	✗ 0,05067	7,35687
5	Cliente, CMS	52,52	✗ 255,94	3,9981	0,00011	0,08242	0,01075	8,42000
6	Cliente, CMS, Remarcação paralela	51,84	✗ 255,94	3,5358	0,00010	0,07640	0,00926	7,34200
7	Cliente, CMS, Incremental	55,11	✗ 255,94	4,4131	0,00013	0,07709	0,01153	9,83000
8	Cliente, CMS, Paralelo geração jovem	53,13	✗ 255,94	3,6138	0,00012	0,07656	0,00958	7,71000
9	Servidor	40,44	228,64	2,3562	✗ 0,00106	0,37406	0,01986	3,95600
10	Servidor, Serial	34,37	246,86	4,3225	0,00092	0,49250	0,03815	6,58600
11	Servidor, Paralelo geração estável	25,13	✓ 227,32	5,2306	0,00084	2,91210	0,03995	6,98650
12	Servidor, CMS	✓ 56,07	255,68	3,6529	0,00012	0,08256	0,00956	8,31450
13	Servidor, CMS, Remarcação paralela	54,56	255,68	3,6134	✓ 0,00009	✓ 0,05625	✓ 0,00924	7,95120
14	Servidor, CMS, Incremental	52,47	255,68	4,8118	0,00010	0,09556	0,01157	✗ 10,12450
15	Servidor, CMS, Paralelo geração jovem	54,08	255,68	3,4515	0,00011	0,07659	0,01002	7,51620

b) *Heap* de até 512MB

Conf.	Coletor e opções	*Throughput (%)*	*Footprint (MB)*	acumulado	menor	maior	média	Tempo total
1	Cliente	38,61	✓ 221,38	3,1951	0,00080	0,29604	0,02280	5,20500
2	Cliente, Paralelo	✗ 31,36	292,13	3,4321	0,00080	0,44200	0,03038	5,00000
3	Cliente, Paralelo geração jovem	45,07	314,43	2,2731	0,00049	0,56790	0,02410	4,13800
4	Cliente, Paralelo geração estável	39,24	315,65	2,9485	0,00086	0,51647	0,02747	4,85300
5	Cliente, CMS	52,71	✗ 511,94	3,5623	✓ 0,00011	0,11727	0,01139	7,53240
6	Cliente, CMS, Remarcação paralela	62,46	✗ 511,94	2,9865	0,00015	0,11867	0,01170	7,95620
7	Cliente, CMS, Incremental	55,76	✗ 511,94	3,9653	✓ 0,00011	0,12022	0,01328	8,96320
8	Cliente, CMS, Paralelo geração jovem	51,81	✗ 511,94	3,5256	0,00013	0,11964	0,01144	7,31570
9	Servidor	✓ 75,19	346,38	✓ 0,6414	0,00077	0,11926	0,02276	✓ 2,58500
10	Servidor, Serial	40,54	486,38	✗ 4,3653	0,00068	0,30125	0,02294	7,34200
11	Servidor, Paralelo geração estável	50,30	458,88	1,4338	0,00081	✗ 0,60822	✗ 0,04343	2,88500
12	Servidor, CMS	42,94	508,64	3,7583	0,00016	0,09523	✓ 0,00956	6,58600
13	Servidor, CMS, Remarcação paralela	48,85	508,64	3,9568	0,00012	0,10030	0,01106	7,73500
14	Servidor, CMS, Incremental	59,66	508,64	4,0165	✗ 0,00109	0,16526	0,01132	✗ 9,95600
15	Servidor, CMS, Paralelo geração jovem	45,72	508,64	3,8568	✓ 0,00011	✓ 0,09232	0,01135	7,10600

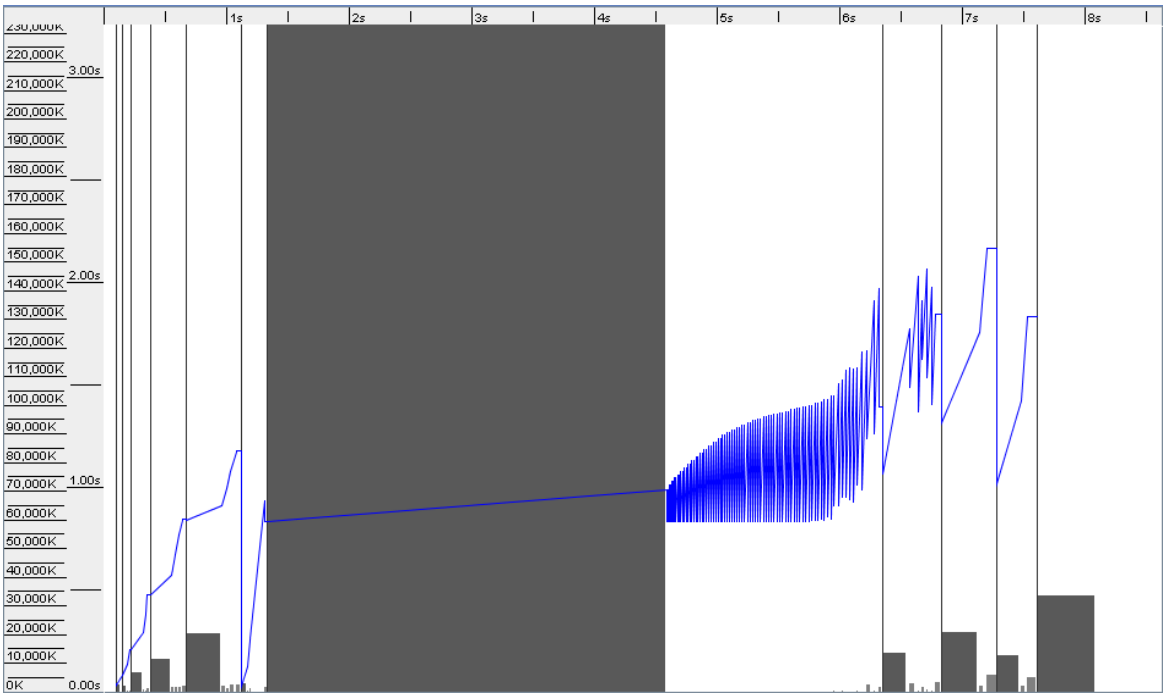

Figura 4.4 – Utilização de memória e tempos de pausa no aplicativo GCBench utilizando o coletor de lixo cliente paralelo na geração estável

Assim, o espaço livre na geração estável tornou-se menor do que a média de promoção, provocando uma limpeza completa. O excesso no tempo para o processo é resultado da nova modificação no tamanho da geração estável e das atualizações nos valores médios de promoção.

Esse comportamento diferente não é encontrado para a execução com a *heap* de 512MB, já que com mais memória, o espaço disponível na geração estável é suficiente para atender à demanda de alocação prevista.

Para o coletor CMS no modo cliente (5, 6, 7 e 8), a característica marcante é que, na execução com a *heap* de 512MB, o tamanho da geração jovem chega próximo de 400MB, permitindo que as alocações das árvores de testes sejam efetuadas todas nesta geração. Como conseqüência, o número de coletas maiores e o tempo de pausa são reduzidos. Sendo assim, somente as estruturas de dados que permanecem durante toda a execução foram promovidas à geração estável. Para a *heap* de 256MB, como as árvores de testes não cabem na geração jovem e não há espaço para aumentar o tamanho da mesma, essas árvores são promovidas à geração estável. Isso fez com que a geração jovem passasse a ser muito pequena (cerca de 15MB) porque a geração estável precisou crescer. Contudo, o tempo de pausa continuou baixo porque as coletas maiores tiveram que analisar uma *heap* da metade do tamanho, se comparadas à execução anterior. Esses coletores tiveram

os maiores *footprints* entre os coletores do modo cliente, utilizando praticamente toda a memória disponível.

O modo incremental (7) apresentou um aumento no tempo total de execução e manteve o *throughput*, o que indica que mais segundos foram gastos com processos de limpeza. Esse comportamento é esperado para este tipo de coletor.

No modo servidor, o comportamento dos coletores foi muito parecido com os do modo cliente, salvo pelo coletor paralelo padrão (9). Para este último, obtivemos os melhores resultados com a *heap* de 512MB atingindo o menor tempo total de execução e o melhor *throughput* entre todas as execuções com a mesma *heap*. Para a *heap* de 256MB os valores foram bons, mas são superados pelo coletor CMS modo servidor e o coletor cliente paralelo na geração jovem.

As figuras 4.5, 4.6 e 4.7 representam os gráficos comparativos de *footprint, throughput* e tempo de execução para os testes realizados nesta aplicação com os dois tamanhos de *heap* mencionados.

Através dos gráficos é fácil perceber que quando não há restrição no consumo de memória o melhor coletor para este aplicativo é o coletor padrão do modo servidor (paralelo representado por 9). Ele atingiu os melhores *throughput* e tempo de execução, além de deixar o *footprint* em um valor baixo.

Os coletores CMS também apresentaram bons *throughputs*, mas novamente deixaram seus tempos de execução e consumo de memória em valores elevados. O menor consumo de memória ficou por conta do coletor serial (1), que manteve baixo o consumo independente do tamanho da *heap*. Outro coletor com tempo de execução e *footprint* pequenos foi o cliente paralelo na geração estável. Por outro lado, esse coletor mostrou um dos menores *throughputs*.

4.4.3. JGFCreateBench

A execução do aplicativo JGFCreateBench, que faz intenso uso da geração jovem, apresentou resultados diferentes para cada grupo de coletores, mas manteve sempre alto os valores de *throughput*.

Durante o decorrer da aplicação são alocados aproximadamente 72,5GB de dados, que imediatamente são liberados para limpeza. As avaliações foram realizadas para uma *heap* de tamanho máximo de 256MB, mas a maior parte das execuções não fez uso desta grande disponibilidade de memória. A tabela 4.4 apresenta os valores médios para os indicadores obtidos nas execuções deste aplicativo.

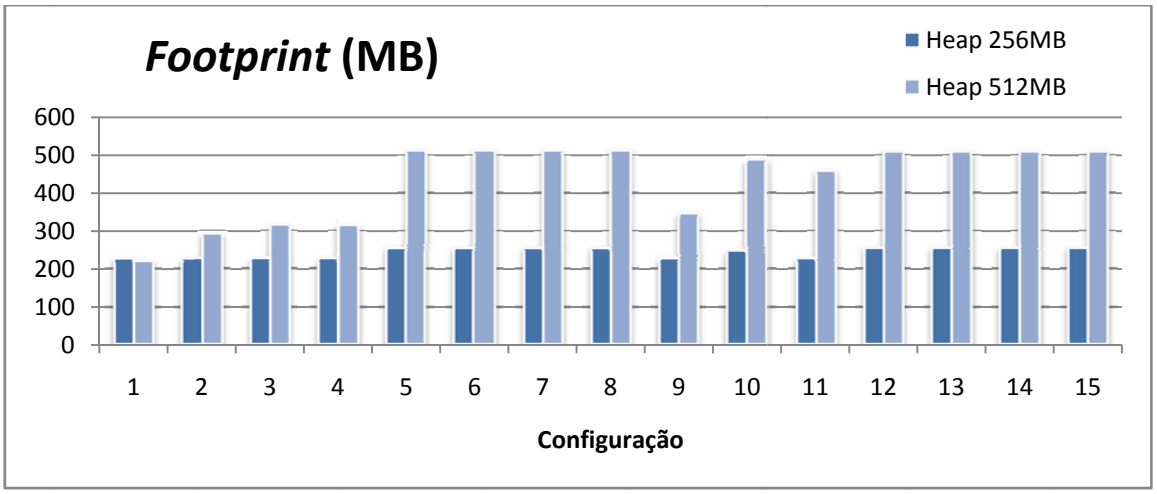

Figura 4.5 – Comparativo entre os *footprints* das execuções do aplicativo GCBench

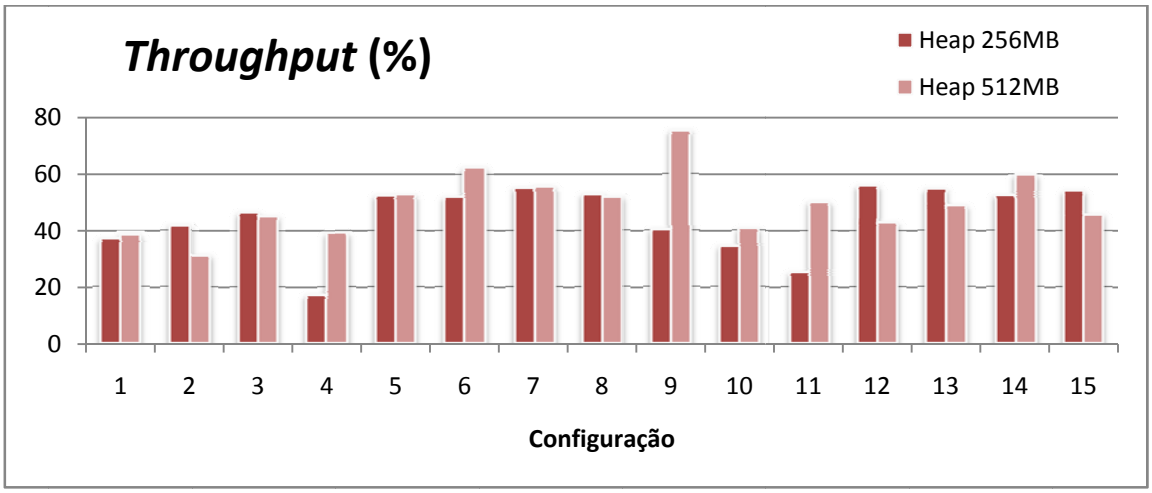

Figura 4.6 – Comparativo entre os *throughputs* das execuções do aplicativo GCBench

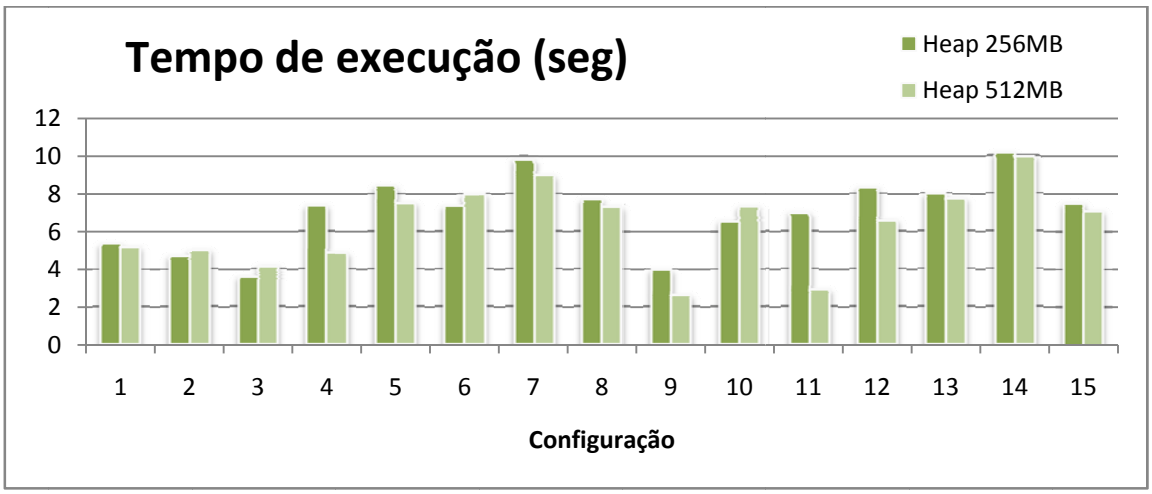

Figura 4.7 – Comparativo entre os tempos de execução do aplicativo GCBench

O coletor cliente serial (1) e os coletores cliente paralelo (2) e cliente paralelo na geração estável (4) apresentaram resultados bastante semelhantes. As três configurações mostraram tempos de execução muito próximos, com média de 47,6 segundos, e throughput alto, perto de 98%. A única diferença foi em relação ao consumo máximo de memória. Após as primeiras alocações de objetos, o algoritmo identifica que a aplicação preencheu a geração jovem completamente, aumenta o tamanho da mesma e executa um processo de limpeza. Como todos os objetos foram eliminados, a geração nova fica vazia. Novamente ao ser preenchida, outra coleta é iniciada e o tamanho incrementado. Esse comportamento faz com que logo no início da execução exista uma geração jovem muito grande enquanto a geração estável permanece constante num valor pequeno. Após algumas coletas menores, uns poucos objetos que sobreviveram às mesmas (como contadores e variáveis auxiliares de execução) devem ser promovidos para a geração estável. Uma coleta maior é então executada e, durante este processo, o coletor identifica qual o comportamento da aplicação. Para este caso, onde as alocações preenchem a geração jovem e ao final de uma coleta menor a mesma fica vazia, o coletor reduz novamente o tamanho desta geração a fim de economizar memória e diminuir o tempo das pausas.

Todo esse processo caracteriza um *footprint* alto no começo da execução e um consumo de memória baixo (média de 150MB com estas três configurações) para o restante da aplicação. O coletor paralelo (2) foi o que obteve o maior *footprint* entre os três já apresentados (251,4MB).

O coletor cliente paralelo na geração jovem (3) foi o algoritmo que menos consumiu memória entre as quinze configurações: somente 4,16MB. Este baixo consumo foi resultado de muitos processos de limpeza na geração jovem ao invés de aumentar o tamanho da *heap*. Como conseqüência, o tempo acumulado de pausas cresceu, tornando-se o maior entre todos. Isso fez o *throughput* cair e se tornar também o pior resultado. O tempo total de execução, entretanto, manteve-se mediano em 47,5 segundos.

Os coletores incrementais (5, 6, 7 e 8) tiveram comportamentos extremamente semelhantes. Todos eles mantiveram a *heap* em 20,7MB e um éden pequeno (de 5MB). Isso foi feito para que a *heap* não aumentasse de tamanho mantendo as coletas rápidas. Os tempos de execução também foram próximos, na casa de 51 segundos, com exceção do modo incremental, que como já foi abordado, tem a execução mais lenta para a máquina utilizada. Os valores de *throughput* também ficaram parecidos, com média de 87,7%.

Tabela 4.4 – Valores médios obtidos na execução do JGFCreateBench para os indicadores avaliados

Heap de até 256MB				Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	✓ 98,46	164,38	✓ 0,7365	0,00011	0,00746	0,00030	47,75300
2	Cliente, Paralelo	98,19	✗ 251,38	0,8650	0,00015	0,00764	0,00032	47,70800
3	Cliente, Paralelo geração jovem	✗ 77,60	✓ 4,16	✗ 11,0818	0,00012	0,00901	0,00020	49,48130
4	Cliente, Paralelo geração estável	97,60	164,38	1,1381	0,00011	✗ 0,02416	✗ 0,00047	47,44200
5	Cliente, CMS	87,65	20,69	6,2832	0,00020	0,01215	0,00037	50,88700
6	Cliente, CMS, Remarcação paralela	88,03	20,69	6,1119	✗ 0,00021	0,01226	0,00036	51,04700
7	Cliente, CMS, Incremental	87,13	20,69	6,7468	0,00011	✓ 0,00542	0,00038	52,40800
8	Cliente, CMS, Paralelo geração jovem	88,15	20,69	5,9881	✗ 0,00021	0,01249	0,00035	50,51700
9	Servidor	98,41	✗ 251,38	0,7567	✗ 0,00021	0,00754	0,00032	47,48100
10	Servidor, Serial	86,86	✓ 4,16	6,0775	✓ 0,00006	0,00857	✓ 0,00010	✓ 46,26375
11	Servidor, Paralelo geração estável	97,67	164,38	1,1110	0,00012	0,02090	✗ 0,00047	47,69900
12	Servidor, CMS	88,38	20,69	5,9095	0,00020	0,01236	0,00035	50,86700
13	Servidor, CMS, Remarcação paralela	87,52	20,69	6,3113	0,00020	0,01241	0,00038	50,58500
14	Servidor, CMS, Incremental	89,20	20,69	5,8564	0,00020	0,01226	0,00035	✗ 54,22200
15	Servidor, CMS, Paralelo geração jovem	86,32	20,69	6,9012	0,00011	0,00877	0,00038	50,43400

Os coletores paralelos do modo servidor (9 e 11) se comportaram como os coletores paralelos do modo cliente: houve um aumento no uso de memória durante início e uma redução posterior, estabilizando em um valor médio de 151MB. Novamente os tempos de execução ficaram baixos e o *throughput* alto, em 98%.

O coletor serial do modo servidor (10) apresentou um dos melhores resultados para consumo de memória (4,16MB) e o menor tempo total de execução (46,26 segundos). Para isso, o coletor se comportou como os coletores CMS, mantendo o tamanho da *heap* fixo em um valor pequeno, evitando o aumento da mesma e, conseqüentemente, o aumento do tempo de limpeza. A diferença é que, para este coletor, metade da *heap* foi disponibilizada para a geração jovem, o que fez com que o número de coletas caísse em aproximadamente 40%.

Os coletores CMS do modo servidor (12, 13, 14 e 15) novamente se comportaram como os coletores CMS do modo cliente. A única diferença foi em relação ao tempo total de execução do modo incremental (14), que aumentou em quase dois segundos.

As figuras 4.8, 4.9 e 4.10 representam os gráficos comparativos de *footprint, throughput* e tempo de execução para os testes realizados nesta aplicação.

Através da análise dos gráficos pode-se concluir que os coletores com menos interferência no tempo da aplicação foram os coletores paralelos, tanto no modo cliente como no modo servidor. Entretanto, foram eles os que mais consumiram memória, com exceção do coletor cliente paralelo na geração jovem. Este último e o coletor servidor serial foram os que melhor gerenciaram a memória disponível.

O coletor servidor serial (10) pode ser considerado o melhor coletor para esta aplicação. Ele atingiu quatro melhores valores dentre os sete indicadores analisados na tabela 4.4 e ainda manteve o *throughput* próximo do melhor valor alcançado.

4.4.4. JGFSerialBench

As análises realizadas com o aplicativo JGFSerialBench tiveram os tamanhos da *heap* definidos em 256MB e 512MB. Este aplicativo faz alocação de menos memória do que os outros vistos até aqui, em média 1GB por execução. No geral, para todas as quinze configurações distintas, o comportamento foi o mesmo: o consumo de memória cresce à uma taxa constante nos primeiros 75% da execução da aplicação. Próximo do fim há um processo de limpeza que produz uma queda na utilização da *heap* e novamente as alocações aumentam até a conclusão.

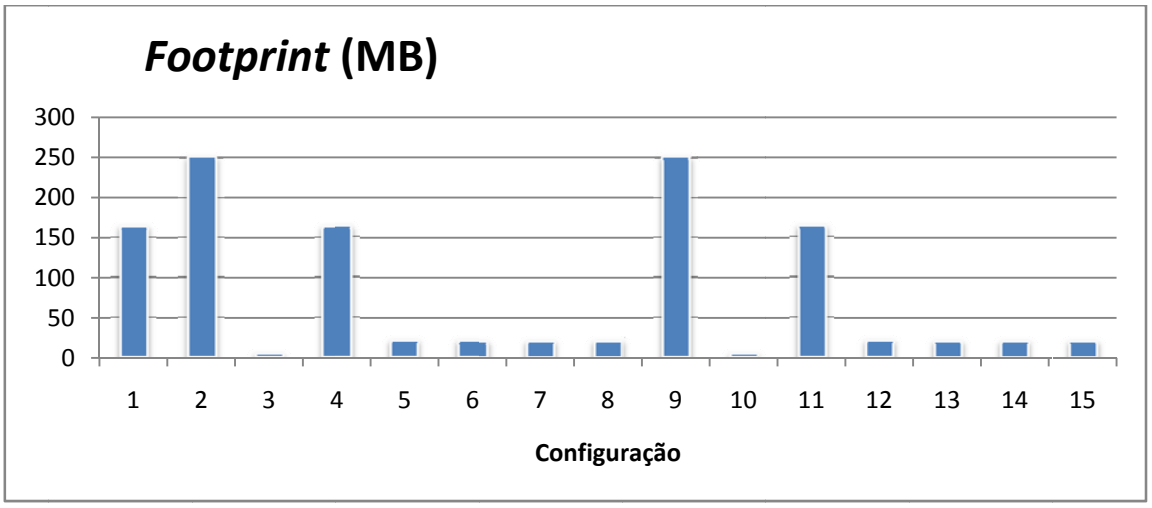

Figura 4.8 – Comparativo entre os *footprints* das execuções do aplicativo JGFCreateBench

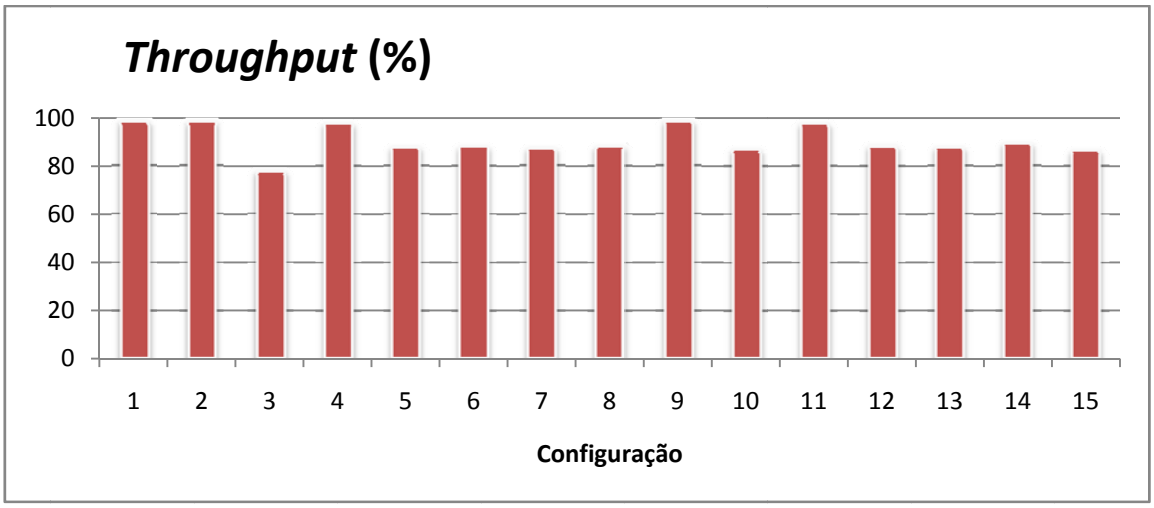

Figura 4.9 – Comparativo entre os *throughputs* das execuções do aplicativo JGFCreateBench

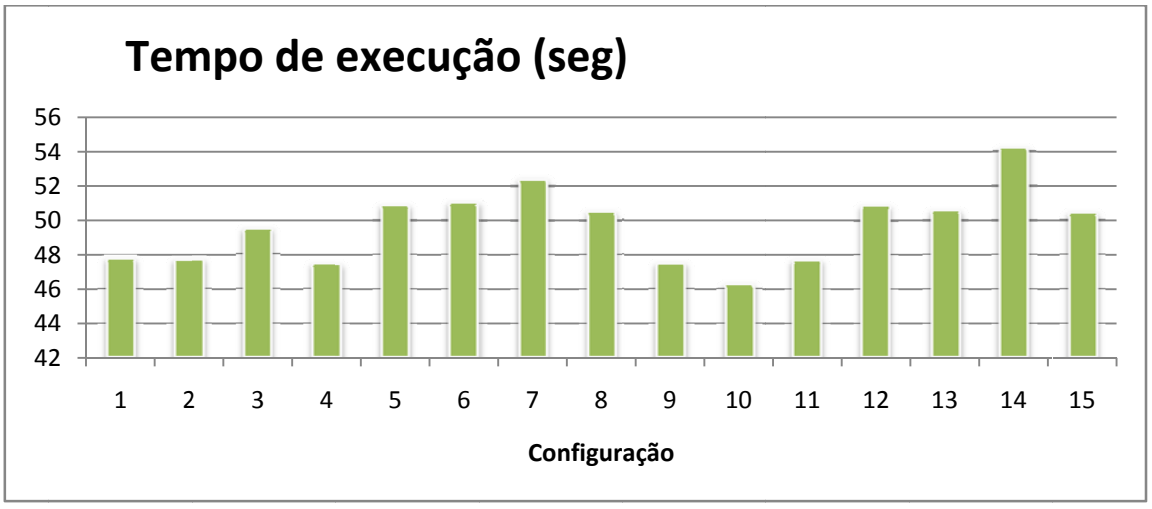

Figura 4.10 – Comparativo entre os tempos de execução do aplicativo JGFCreateBench

A tabela 4.5 contém os valores médios colhidos durante a execução dos algoritmos de limpeza para os indicadores avaliados e serão analisados em seguida.

O coletor cliente serial (1) apresentou bons resultados de *throughput* (94,9%) e consumo de memória (251MB) para ambos os tamanhos de *heap*. Uma diferença entre as duas execuções foi a porcentagem da *heap* destinada ao éden. Para a *heap* de 256MB a geração jovem ocupou em média 20% da memória alocada, enquanto para a *heap* de 512MB esse valor subiu para 40%. Essa diferença é resultado da menor freqüência de coletas completas para a *heap* maior. Mesmo a diferença de tamanho sendo pequena (47,2MB), esse valor já é suficiente para que durante um processo de limpeza mais objetos possam ser eliminados, fazendo a quantidade de promovidos diminuir e a necessidade de espaço na geração estável também.

Os coletores paralelos no modo cliente (2, 3 e 4) tiveram *footprint* de memória semelhantes. O coletor paralelo para a geração estável (4) apresentou tempos de pausa elevados para os dois tamanhos de *heap*, resultando em uma grande queda de *throughput* e pequeno aumento no tempo médio de execução.

Os coletores CMS no modo cliente (5, 6, 7 e 8) e no modo servidor (12, 13, 14 e 15) apresentaram os melhores *throughputs* entre as quinze configurações e tiveram *footprints* de memória muito parecidos (média de 252MB). Os tempos de execução foram diferentes entre eles. Todos mantiveram o tempo acumulado de pausas na casa dos dois segundos e, como esperado, os coletores incrementais acumularam os maiores valores (média de 2,44 segundos). Isso fez com que eles apresentassem *throughputs* menores. Os menores tempos totais de execução ficaram com coletores CMS: para a *heap* de 256MB o coletor servidor CMS paralelo na geração jovem teve o melhor resultado (64,17 segundos) enquanto para a *heap* de 512MB esse mérito foi do coletor CMS cliente (65,634 segundos).

O coletor servidor paralelo na geração estável (11) apresentou os piores resultados em relação ao tempo de execução para os dois tamanhos de *heap*, 6% maior do que o tempo médio de execução para todos os algoritmos. Para a *heap* de 512MB, este coletor também obteve os piores resultados de *throughput,* tempo acumulado de pausas e tempo da maior pausa. O coletor servidor paralelo (9) teve o pior resultado para o indicador de menor tempo de pausa, nos dois tamanhos de *heap.*

As figuras 4.11, 4.12 e 4.13 apresentam os gráficos comparativos de *footprint, throughput* e tempo de execução para os testes realizados nesta aplicação.

Tabela 4.5 – Valores médios obtidos na execução do JGFSerialBench para os indicadores avaliados

a) *Heap* de até 256MB			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	94,23	✓ **227,44**	3,9113	0,00101	0,40748	0,05436	67,83600
2	Cliente, Paralelo	94,81	246,25	3,6142	0,00242	0,41121	0,05816	69,58200
3	Cliente, Paralelo geração jovem	96,48	248,50	2,3837	0,00052	0,47384	0,04499	67,66500
4	Cliente, Paralelo geração estável	✗ **89,09**	227,56	✗ **7,7434**	0,00088	✗ **1,27080**	✗ **0,10456**	70,95600
5	Cliente, CMS	96,91	248,34	2,0834	✓ **0,00016**	0,15773	0,01246	67,52700
6	Cliente, CMS, Remarcação paralela	96,90	247,08	✓ **2,0634**	0,00018	0,15688	0,01235	66,47300
7	Cliente, CMS, Incremental	96,40	✗ **255,94**	2,4618	0,00020	0,08843	0,01090	68,38200
8	Cliente, CMS, Paralelo geração jovem	96,82	248,09	2,0776	✓ **0,00016**	0,15690	0,01238	65,25000
9	Servidor	94,44	227,56	3,7832	✗ **0,00248**	0,42051	0,05812	68,03900
10	Servidor, Serial	94,69	247,50	3,5784	0,00058	0,47100	0,06994	67,37300
11	Servidor, Paralelo geração estável	91,07	227,56	6,3654	0,00085	1,26439	0,08624	✗ **71,26300**
12	Servidor, CMS	96,91	246,25	2,0732	✓ **0,00016**	0,15556	0,01240	67,03500
13	Servidor, CMS, Remarcação paralela	✓ **96,97**	248,20	2,0864	0,00018	0,15609	0,01248	68,78200
14	Servidor, CMS, Incremental	96,48	✗ **255,94**	2,4685	0,00029	✓ **0,05112**	✓ **0,01032**	70,07500
15	Servidor, CMS, Paralelo geração jovem	96,47	✗ **255,94**	2,2635	0,00017	0,26572	0,01381	✓ **64,17100**

b) *Heap* de até 512MB								
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	95,60	274,63	2,9165	0,00088	0,43128	0,05297	66,34000
2	Cliente, Paralelo	95,28	282,63	3,2132	0,00093	0,41991	0,05540	68,06800
3	Cliente, Paralelo geração jovem	96,27	290,40	2,4965	0,00044	0,47077	0,04784	67,00300
4	Cliente, Paralelo geração estável	90,81	270,75	6,2532	0,00093	1,21307	✗ **0,09060**	68,06800
5	Cliente, CMS	96,86	247,24	2,0632	✓ **0,00016**	0,15797	0,01236	✓ **65,63400**
6	Cliente, CMS, Remarcação paralela	96,94	✓ **244,61**	✓ **2,0232**	0,00019	0,15805	0,01212	66,18300
7	Cliente, CMS, Incremental	96,44	256,41	2,4132	0,00019	✓ **0,08553**	0,01070	67,83600
8	Cliente, CMS, Paralelo geração jovem	96,94	247,06	2,0723	✓ **0,00016**	0,15636	0,01238	67,72900
9	Servidor	95,66	277,00	2,9232	✗ **0,00257**	0,43613	0,05119	67,33900
10	Servidor, Serial	94,80	✗ **314,10**	3,5685	0,00072	0,56774	0,07001	68,61200
11	Servidor, Paralelo geração estável	✗ **89,84**	248,25	✗ **7,3384**	0,00100	✗ **1,44178**	0,07557	✗ **72,26200**
12	Servidor, CMS	96,86	247,27	2,0835	✓ **0,00016**	0,15681	0,01244	66,44300
13	Servidor, CMS, Remarcação paralela	96,92	247,24	2,0698	✓ **0,00016**	0,15636	0,01234	67,25900
14	Servidor, CMS, Incremental	96,41	267,47	2,4432	0,00021	0,09616	✓ **0,01047**	68,07200
15	Servidor, CMS, Paralelo geração jovem	✓ **96,96**	247,24	2,0632	✓ **0,00016**	0,15870	0,01234	67,79700

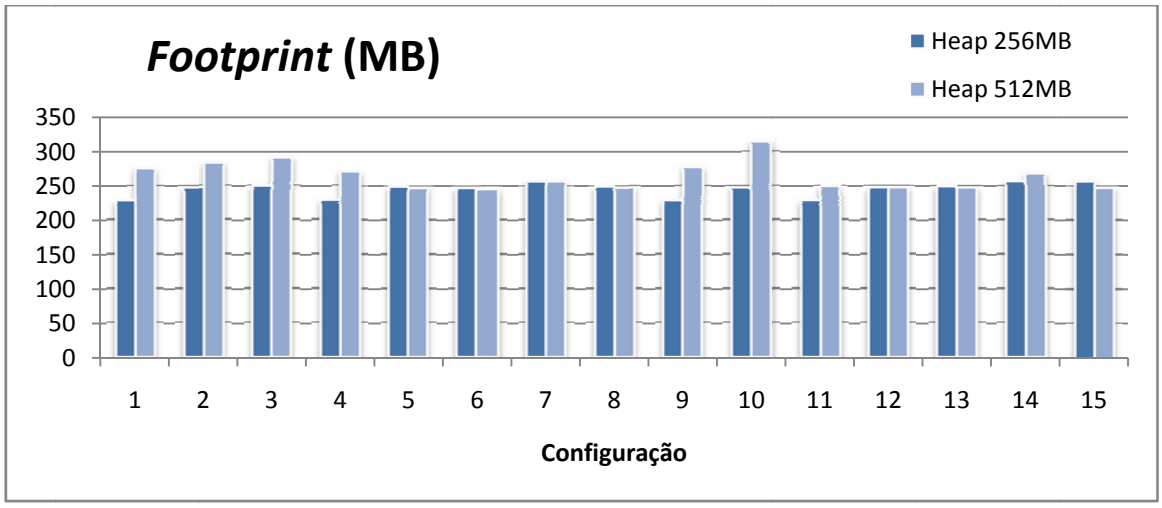

Figura 4.11 – Comparativo entre os *footprints* das execuções do aplicativo JGFSerialBench

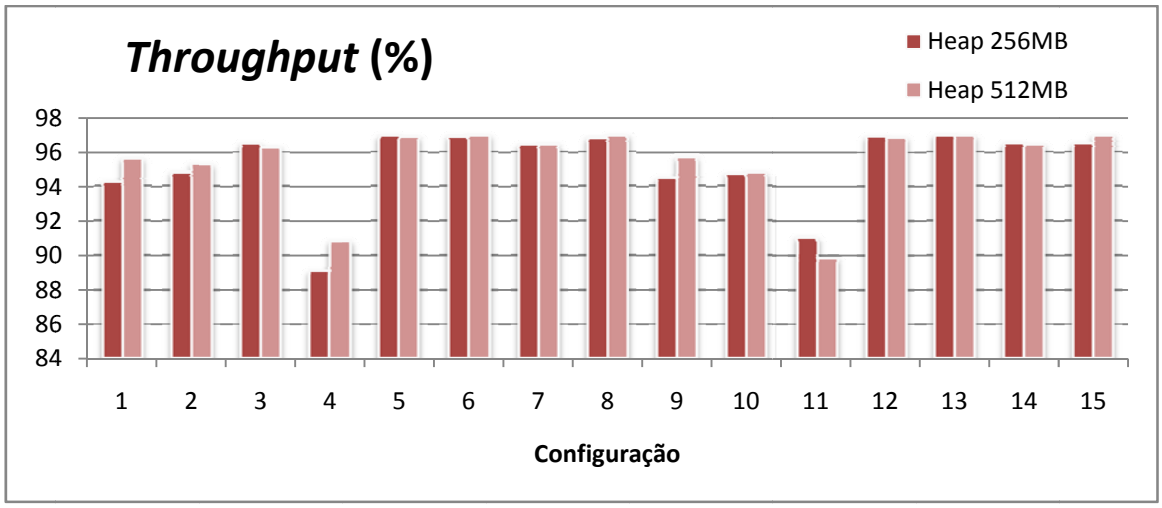

Figura 4.12 – Comparativo entre os *throughputs* das execuções do aplicativo JGFSerialBench

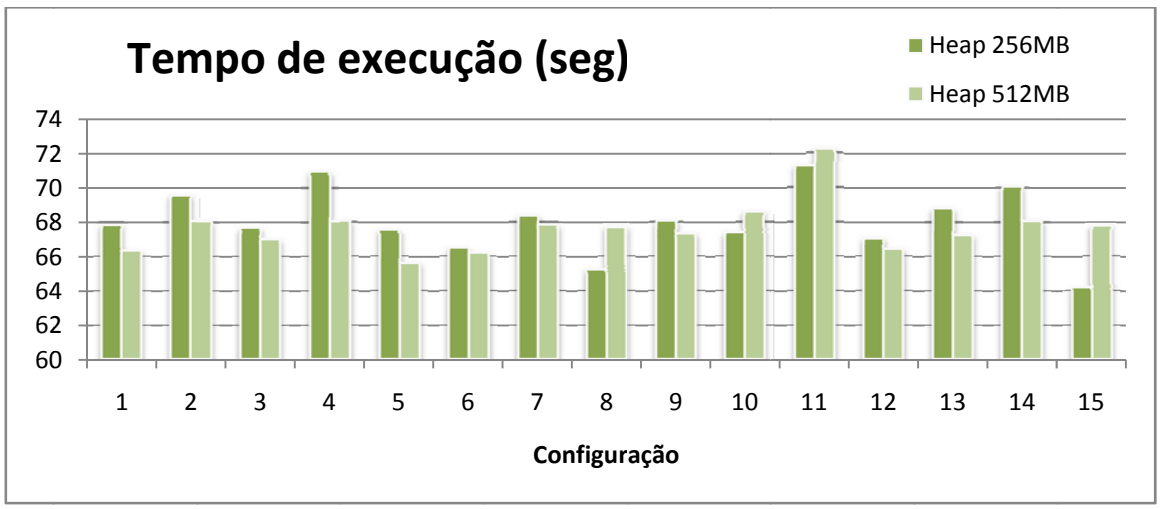

Figura 4.13 – Comparativo entre os tempos de execução do aplicativo JGFSerialBench

No geral, algumas características podem ser observadas. A disponibilidade de uma *heap* maior não foi aproveitada pelos coletores. O melhor e o pior tempo para a *heap* maior aumentaram, mas a média do tempo de execução diminuiu em apenas 0,35 segundos, o que indica que mesmo com mais memória o tempo de execução se manteve estável.

Definir o melhor coletor para esta aplicação tornou-se uma tarefa difícil. Os melhores resultados em geral ficaram com os coletores CMS, exceto para o modo incremental. Se comparado com tempo, os coletores CMS cliente (5) e o CMS servidor com coleta paralela para a geração nova (15) obtiveram os melhores resultados.

4.4.5. JGFForkJoinBench

O JGFForkJoinBench é um aplicativo *multithread* que foi avaliado com tamanho máximo de *heap* de 128MB para dois, três e quatro processos paralelos. Esse aplicativo analisa o desempenho da JVM na criação e junção de *threads*, o que torna sem sentido a realização do mesmo com apenas uma *thread* porque nesta situação nada seria realizado.

O tamanho da heap escolhido foi grande se comparado com aproximadamente 210MB de dados que são alocados em média por execução. Essa quantidade de alocação pode variar de acordo com o número de threads. Esse tamanho total foi utilizado para analisar o comportamento da JVM em aplicações paralelas quando há disponibilidade de memória.

O padrão de comportamento para cada coletor foi o mesmo com os três números de threads distintos. As diferenças ficaram restritas aos indicadores de tempo, o que valida a invariância do algoritmo de coleta. Sendo assim, os comportamentos descritos abaixo são aplicados à todas as quantidades diferentes de threads utilizadas nos testes. Com relação ao tempo, o que pode ser observado no geral é que eles aumentam pouco em relação ao aumento do número de threads. Como o número é no máximo igual à quantidade de núcleos de processamento da máquina utilizada, cada processador se encarrega de executar o trabalho de uma thread, que é igual para todas elas. Isso faz com que o tempo de execução da cada uma seja semelhante, fazendo com que o tempo total da aplicação permaneça estável. O pequeno aumento é resultado de overhead nos procedimentos de fork e join que serão executados mais vezes, além das estruturas maiores que a aplicação deve gerenciar.

A tabela 4.6 apresenta a média dos valores dos indicadores avaliados durante as execuções dos testes neste aplicativo.

Tabela 4.6 – Valores médios obtidos na execução do JGFForkJoinBench para os indicadores avaliados

a) 2 *Threads - Heap de até 128MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	✓ 99,95	5,38	0,0123	0,00017	0,00139	✓ 0,00025	25,32184
2	Cliente, Paralelo	99,91	5,38	0,0235	0,00024	0,00132	0,00039	26,13218
3	Cliente, Paralelo geração jovem	99,85	✓ 4,16	0,0368	0,00023	0,00193	0,00040	25,13518
4	Cliente, Paralelo geração estável	99,83	5,38	0,0432	0,00018	✓ 0,00126	0,00031	26,13517
5	Cliente, CMS	99,90	20,69	0,0268	0,00033	✗ 0,00586	0,00063	26,01385
6	Cliente, CMS, Remarcação paralela	99,87	20,69	0,0332	0,00032	0,00250	0,00051	26,06218
7	Cliente, CMS, Incremental	✗ 98,95	20,69	✗ 0,2818	0,00032	0,00478	0,00170	26,76513
8	Cliente, CMS, Paralelo geração jovem	✓ 99,95	20,69	✓ 0,0118	0,00033	0,00339	0,00054	25,63651
9	Servidor	99,89	✗ 59,06	0,0273	0,00094	0,00174	0,00112	24,13190
10	Servidor, Serial	99,92	✓ 4,16	0,0286	✓ 0,00013	0,00256	✓ 0,00025	✗ 36,16510
11	Servidor, Paralelo geração estável	99,92	✗ 59,06	0,0194	✗ 0,00109	0,00296	0,00152	24,19542
12	Servidor, CMS	99,88	20,69	0,0273	0,00037	0,00265	0,00060	✓ 23,65423
13	Servidor, CMS, Remarcação paralela	99,85	20,69	0,0387	0,00031	0,00255	0,00053	25,91215
14	Servidor, CMS, Incremental	99,40	20,69	0,1587	0,00034	0,00509	✗ 0,00176	26,65813
15	Servidor, CMS, Paralelo geração jovem	99,87	20,69	0,0346	0,00036	0,00310	0,00050	26,31214

b) 3 *Threads - Heap de até 128MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	99,88	5,38	0,0310	0,00020	✓ 0,00112	0,00030	26,75301
2	Cliente, Paralelo	99,86	5,38	0,0384	0,00018	0,00138	0,00031	27,12301
3	Cliente, Paralelo geração jovem	99,82	✓ 4,16	0,0473	0,00020	0,00161	0,00031	26,66801
4	Cliente, Paralelo geração estável	99,89	5,38	0,0312	0,00018	✗ 0,01370	0,00029	27,29001
5	Cliente, CMS	99,91	20,69	0,0256	0,00031	0,00285	0,00044	27,14401
6	Cliente, CMS, Remarcação paralela	99,86	20,69	0,0387	0,00030	0,00326	0,00051	27,11101
7	Cliente, CMS, Incremental	✗ 99,04	20,69	0,2613	0,00030	0,00440	0,00165	27,33701
8	Cliente, CMS, Paralelo geração jovem	99,91	20,69	0,0231	0,00032	0,00221	0,00044	26,67701
9	Servidor	99,94	✗ 59,06	0,0168	✗ 0,00058	0,00159	0,00082	25,92301
10	Servidor, Serial	99,86	✓ 4,16	0,0384	✓ 0,00015	0,00254	✓ 0,00021	26,62801
11	Servidor, Paralelo geração estável	✓ 99,96	✗ 59,06	✓ 0,0114	0,00041	0,00158	0,00079	✓ 25,72801
12	Servidor, CMS	99,88	20,69	0,0316	0,00031	0,00627	0,00050	27,02801
13	Servidor, CMS, Remarcação paralela	99,91	20,69	0,0231	0,00032	0,00288	0,00048	27,04701
14	Servidor, CMS, Incremental	99,05	20,69	✗ 0,2635	0,00030	0,00528	✗ 0,00166	✗ 27,60101
15	Servidor, CMS, Paralelo geração jovem	99,87	20,69	0,0361	0,00030	0,00359	0,00057	27,14601

c) 4 *Threads - Heap de até 128MB*				Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	99,86	5,38	0,0398	✓ 0,00017	0,00127	0,00032	27,86699
2	Cliente, Paralelo	99,92	5,38	0,0221	0,00020	✓ 0,00109	0,00030	27,97299
3	Cliente, Paralelo geração jovem	99,87	✓ 4,16	0,0368	0,00022	0,00142	0,00033	27,54599
4	Cliente, Paralelo geração estável	99,92	5,38	0,0232	0,00018	✓ 0,00109	0,00027	27,90499
5	Cliente, CMS	99,90	20,69	0,0287	✗ 0,00035	0,00263	0,00053	27,42099
6	Cliente, CMS, Remarcação paralela	99,90	20,69	0,0287	0,00030	0,00461	0,00049	27,64499
7	Cliente, CMS, Incremental	✗ 99,10	20,69	✗ 0,2560	0,00031	0,00453	✗ 0,00168	✗ 28,30599
8	Cliente, CMS, Paralelo geração jovem	99,92	20,69	0,0213	0,00032	0,00266	0,00046	27,64099
9	Servidor	99,93	✗ 59,06	0,0176	0,00029	0,00157	0,00077	✓ 26,65999
10	Servidor, Serial	99,90	✓ 4,16	0,0286	0,00018	0,00261	✓ 0,00022	27,83299
11	Servidor, Paralelo geração estável	✓ 99,95	✗ 59,06	✓ 0,0132	0,00029	0,00157	0,00068	26,76599
12	Servidor, CMS	99,92	20,69	0,0217	0,00030	0,00284	0,00047	27,61099
13	Servidor, CMS, Remarcação paralela	99,90	20,69	0,0268	0,00032	0,00282	0,00048	27,36699
14	Servidor, CMS, Incremental	99,30	20,69	0,1970	0,00033	✗ 0,00472	0,00161	28,01799
15	Servidor, CMS, Paralelo geração jovem	99,90	20,69	0,0268	0,00033	0,00260	0,00056	27,76999

Para os coletores cliente serial (1), cliente paralelo (2) e cliente paralelo na geração estável (4) o consumo de memória foi o mesmo e, em ambos coletores, o tamanho total da heap diminui de 5,38MB, que era o valor inicial da mesma (era também o footprint) para aproximadamente 4MB até os 10 segundos de execução. Essa redução de tamanho foi realizada na geração jovem, já que nesta aplicação poucos objetos são promovidos à geração estável e, para estes coletores, somente uma limpeza completa foi executada, ao final da aplicação. As médias de tempo entre eles ficaram em 25,9 segundos para duas threads, 27,05 segundos para três e 27,9 segundos para quatro threads, caracterizando o pequeno aumento de tempo já apresentado.

O coletor cliente paralelo na geração jovem (3) e o coletor servidor serial (9) foram os que menos consumiram memória entre todas as configurações. O footprint ficou em 4,16MB e esses dois coletores mantiveram o tamanho total da heap inalterado, já que diminuir ainda mais o tamanho da mesma provocaria excesso de limpezas. O coletor servidor serial (9) conseguiu manter os tempos de pausa extremamente baixos e, como conseqüência, teve os melhores valores de tempo médio de pausas para todas as quantidades de threads.

Os coletores CMS cliente (5, 6 e 8) e servidor (12, 13 e 15), com exceção dos coletores incrementais, também tiveram comportamento parecidos. Estes coletores criaram uma heap de 20,69MB com apenas 5MB de geração jovem. Esse valor pequeno na geração jovem já era esperado, com o objetivo de manter baixos os tempos de pausa. Somente uma coleta completa ocorreu neste coletor para esta aplicação, sempre no final da execução para liberar toda a memória alocada. Como resultado, é possível ver um desperdício grande de memória. Se não há coletas na geração estável devido ao pequeno tamanho necessário na mesma, não há porque mantê-la com mais de 15MB. O coletor CMS cliente incremental (7) e o CMS servidor incremental (14) realizaram várias coletas completas na heap durante a execução, o que justifica as maiores pausas e, como conseqüência, os maiores valores para tempo acumulado em coletas. Isso acarretou nos piores throughputs e tempos de execução longos.

Os coletores servidor paralelo (9) e servidor paralelo na geração estável (11) foram os algoritmos que tiveram o maior *footprint* (59,06MB). Eles mantiveram cerca de 20MB de espaço disponível para a geração jovem, fazendo com que as coletas menores ocorressem com menos freqüência. Eles realizaram apenas duas coletas completas na *heap*, uma logo após o início e outra ao fim da execução. Com uma geração jovem maior e

poucas coletas completas, esses coletores conseguiram diminuir o tempo total da aplicação, sendo deles os melhores tempos de execução do aplicativo para três e quatro *threads*.

As figuras 4.14, 4.15 e 4.16 apresentam gráficos comparativos entre *footprint*, *throughput* e tempo de execução para este aplicativo, considerando as execuções com duas, três e quatro *threads*.

Os coletores cliente e servidor serial e os coletores clientes paralelos foram os que melhor gerenciaram a memória, mantendo o consumo baixo. Todas as configurações, com exceção dos CMS incrementais, interferiram pouco no tempo de aplicação, trazendo como resultado *throughputs* altos.

Através da análise dos gráficos é possível definir o melhor coletor para esta aplicação como sendo o coletor cliente paralelo na geração jovem (3), que obteve o menor *footprint*, tempos de execução dentro da média e valores altos para o *throughput.*

4.4.6. GeneId

O GeneId é uma aplicação originalmente escrita em C e que depois foi traduzida para Java e paralelizada. As modificações não foram suficientes para reduzir o tempo total de computação, principalmente porque o tempo total de computação é relativamente baixo, poucos passos computacionais foram paralelizados, e pals operações de sincronização serem muitas e caras. Essa combinação de fatores fez com que o tempo total de execução aumentasse com o número de threads. Quando passamos de uma para duas threads o tempo aumenta, em média, 250%. Este valor é ainda maior quando passamos de uma para três threads (aumento de 300%). Não foi possível executar o aplicativo para quatro threads devido a um erro de paralelização do código, fazendo com que, para esta situação, uma das threads quebrasse e a aplicação não pudesse ser concluída. Como a correção do código foge ao escopo deste trabalho, a execução deste aplicativo com quatro threads foi simplesmente descartada das análises.

O programa faz grande uso da memória, armazenando estruturas de dados com aproximadamente 200MB durante toda a execução e realizando alocações de aproximadamente 1,5GB em objetos distintos. Para validar os experimentos, foram utilizados tamanhos máximos de heap de 512MB e 1024MB, em execuções com uma, duas e três threads.

No geral, o padrão de comportamento dos coletores foi o mesmo independente do número de threads, mas o consumo de memória e os indicadores de tempo alteraram em função do espaço disponível e do número de processos paralelos.

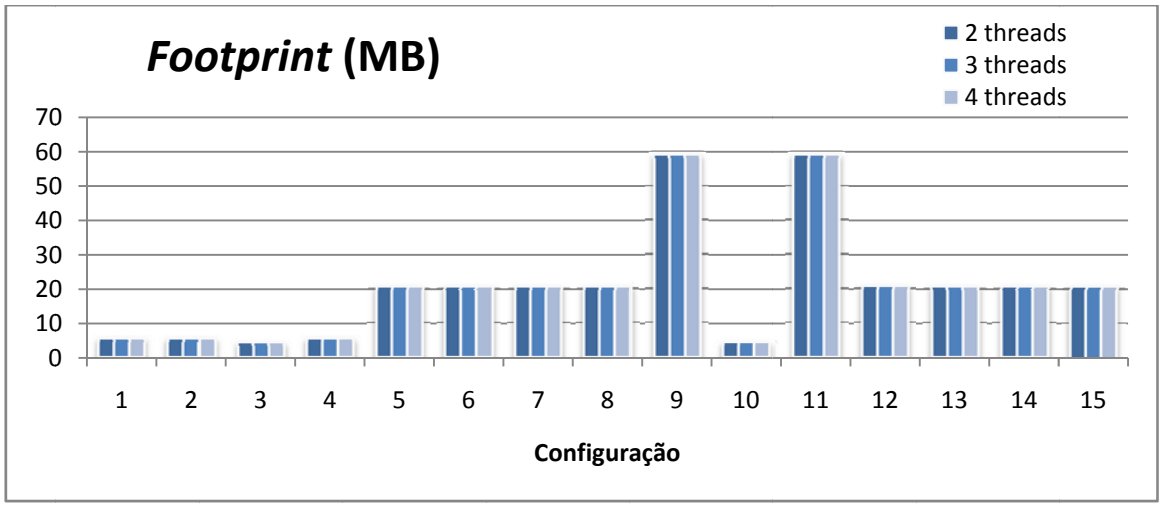

Figura 4.14 – Comparativo entre os *footprints* das execuções do aplicativo JGFForkJoinBench

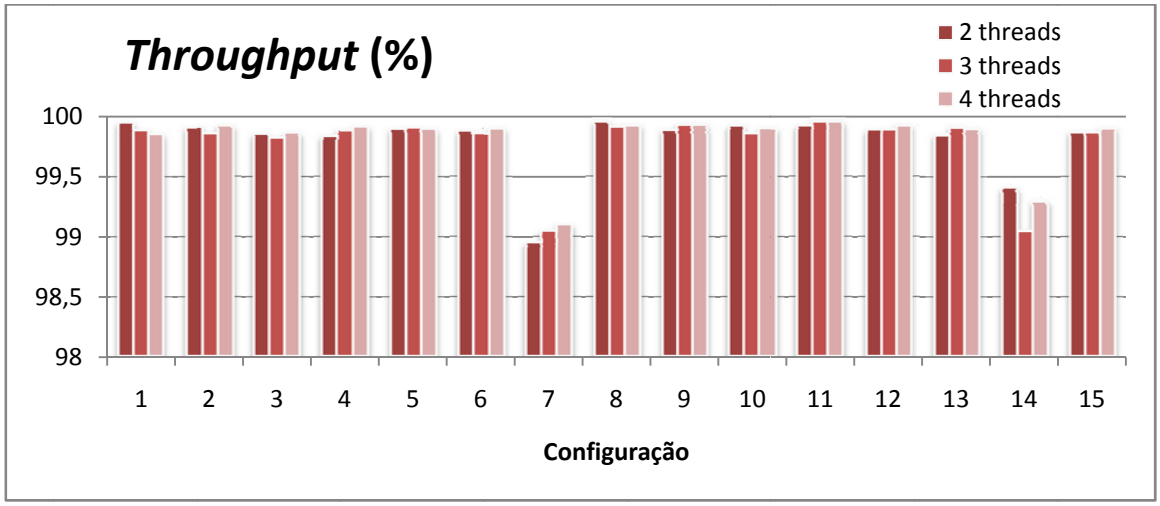

Figura 4.15 – Comparativo entre os *throughputs* das execuções do aplicativo JGFForkJoinBench

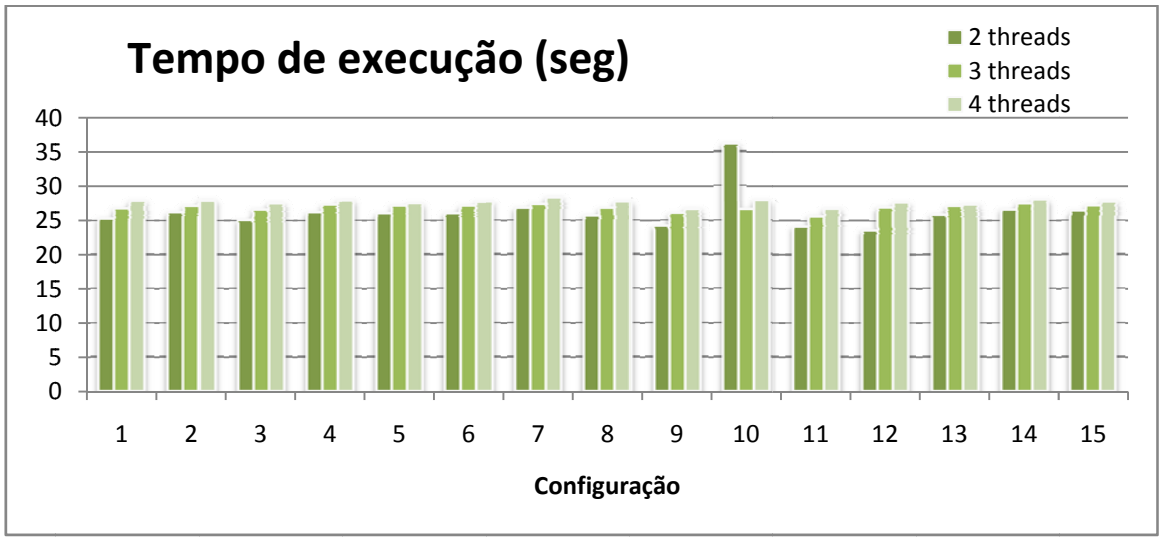

Figura 4.16 – Comparativo entre os tempos de execução do aplicativo JGFForkJoinBench

A tabela 4.7 apresenta os valores médios obtidos com as execuções do aplicativo no grupo de coletores de testes com tamanho máximo da heap de 512MB. Os mesmos indicadores são apresentados na tabela 4.8 para a heap de 1024MB

Como a grande parte dos objetos utilizados é colocada em memória no começo da execução, as primeiras coletas fazem a heap atingir o seu tamanho máximo. A partir deste ponto, as alocações no éden são realizadas e as coletas menores vão eliminando os objetos que logo se tornam lixo. O coletor cliente serial (1) fez com que o tamanho das gerações e da heap permanecessem inalterados até o fim da execução, não fazendo uso da maior disponibilidade de memória. Assim, o comportamento para os tamanhos de heap de 512MB e 1024MB foram semelhantes, com indicadores bem próximos para o mesmo número de threads. O footprint teve um aumento médio de 17% por thread a mais em execução para este coletor. Já o tempo total da aplicação aumentou obedecendo às proporções apresentados anteriormente. O destaque foi para o menor tempo total de aplicação (165,1 segundos) deste coletor entre os quinze analisados, obtido para a heap de 1024MB com duas threads em execução.

O coletor cliente paralelo (2) apresentou resultados semelhantes ao coletor cliente serial. O coletor cliente paralelo na geração jovem (3) foi o que, no geral, mais consumiu memória, com footprint médio de 565MB entre todas as execuções e o maior valor de 819,46MB para três threads na heap de 1024MB. Este algoritmo apresentou os melhores resultados para execuções com apenas uma thread, mas o ganho é pequeno em relação aos outros coletores. Uma das características observadas foi o aumento no número de coletas completas à medida que o número de threads aumenta (principalmente para a heap de 512MB), fazendo com que os resultados para as execuções com duas e três threads não sejam os melhores.

O coletor cliente paralelo na geração estável (4) apresentou os piores valores em suas execuções. Os baixos indicadores são resultados de tempos de pausa altos, com coletas completas demoradas, chegando em seis segundos para uma única limpeza. Essa característica é conseqüência de uma geração estável cheia e com poucos objetos candidatos a lixo. Assim, as threads de coleta gastam muito tempo varrendo a geração para eliminarem praticamente nenhum lixo (lembrando que ainda existe o cálculo do prefixo denso, visto em 3.3.2, que possui um custo computacional representativo no processo). Os piores valores ficaram com a heap de 512MB e três threads em execução, configuração esta que demanda forte uso do coletor, já que a necessidade de memória é maior e a disponibilidade menor.

Tabela 4.7 – Valores médios obtidos na execução do GeneId para os indicadores avaliados em uma *heap* de até 512MB

a) 1 *Thread - Heap de até 512MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	96,07	316,06	2,1121	0,00895	0,19327	0,02932	53,74500
2	Cliente, Paralelo	95,97	315,19	2,1764	0,00267	0,24550	0,01938	54,01800
3	Cliente, Paralelo geração jovem	✓ 98,42	✗ 468,70	✓ 0,8131	0,00680	0,14106	0,03532	✓ 51,35000
4	Cliente, Paralelo geração estável	✗ 84,85	281,25	✗ 9,1765	✗ 0,01330	3,87770	0,05894	✗ 60,56200
5	Cliente, CMS	98,04	334,58	1,0332	0,00017	0,02841	✓ 0,00091	52,80800
6	Cliente, CMS, Remarcação paralela	97,97	334,58	1,0721	0,00018	0,02826	0,00936	52,82500
7	Cliente, CMS, Incremental	97,67	365,13	1,2431	0,00018	✓ 0,02639	0,00792	53,27700
8	Cliente, CMS, Paralelo geração jovem	97,97	320,50	1,0765	0,00016	0,02898	0,00907	53,06700
9	Servidor	95,29	292,19	2,5550	0,00719	0,25548	0,02260	54,25500
10	Servidor, Serial	98,04	454,88	1,0321	0,00801	0,15980	0,04284	52,58300
11	Servidor, Paralelo geração estável	85,91	✓ 279,81	8,4340	0,01131	✗ 4,36514	✗ 0,07267	59,84600
12	Servidor, CMS	97,99	327,60	1,0684	0,00015	0,02878	0,00940	53,18800
13	Servidor, CMS, Remarcação paralela	98,04	334,58	1,0337	0,00018	0,02840	0,00908	52,70100
14	Servidor, CMS, Incremental	97,71	365,13	1,2199	✓ 0,00014	0,02815	0,00778	53,18700
15	Servidor, CMS, Paralelo geração jovem	97,94	320,50	1,0936	0,00017	0,03055	0,00932	53,07900

b) 2 *Threads - Heap de até 512MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	97,68	366,94	4,4435	0,00606	0,37556	0,04439	191,17400
2	Cliente, Paralelo	97,52	372,50	4,8900	✗ 0,00898	0,33167	0,03824	197,11300
3	Cliente, Paralelo geração jovem	97,99	494,94	4,2165	0,00815	0,48344	0,18312	210,28900
4	Cliente, Paralelo geração estável	98,20	356,88	4,4900	0,00331	1,41492	0,03210	249,71500
5	Cliente, CMS	✓ 99,34	481,65	✓ 1,3199	0,00018	0,03650	0,01109	200,69400
6	Cliente, CMS, Remarcação paralela	98,84	480,66	1,3265	✓ 0,00017	0,03298	0,01119	114,23600
7	Cliente, CMS, Incremental	98,94	489,61	1,4897	0,00019	✓ 0,03029	0,00951	140,58700
8	Cliente, CMS, Paralelo geração jovem	98,73	480,66	1,3832	✓ 0,00017	0,03946	0,01167	✓ 109,11900
9	Servidor	97,60	379,56	3,8432	0,00644	0,03469	0,03255	159,83700
10	Servidor, Serial	98,62	494,94	4,4598	0,00801	0,43280	✗ 0,19357	✗ 323,31000
11	Servidor, Paralelo geração estável	✗ 93,54	✓ 351,13	✗ 10,8465	0,00429	✗ 4,75784	0,07796	167,81900
12	Servidor, CMS	99,13	486,88	1,3832	0,00019	0,03154	0,01169	158,95600
13	Servidor, CMS, Remarcação paralela	99,25	475,95	1,3588	✓ 0,00017	0,03121	0,01144	180,02200
14	Servidor, CMS, Incremental	99,27	✗ 501,66	1,5212	0,00021	0,03624	✓ 0,00922	209,50200
15	Servidor, CMS, Paralelo geração jovem	99,31	484,19	1,3464	✓ 0,00017	0,03646	0,01133	196,35700

c) 3 *Threads - Heap de até 512MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	85,60	412,38	33,3213	0,00990	0,52062	0,02162	231,47400
2	Cliente, Paralelo	88,51	420,00	25,1932	✗ **0,01022**	0,53981	0,21908	219,26300
3	Cliente, Paralelo geração jovem	97,34	494,94	6,0132	0,00693	0,61061	0,25028	225,65200
4	Cliente, Paralelo geração estável	✗ **73,28**	411,50	✗ **72,5265**	0,00978	✗ **6,16802**	✗ **0,34370**	✗ **271,42300**
5	Cliente, CMS	99,28	✗ **511,94**	1,5338	0,00019	0,02856	0,01251	211,97700
6	Cliente, CMS, Remarcação paralela	99,22	✗ **511,94**	✓ **1,4988**	✓ **0,00017**	0,03375	0,01224	✓ **191,72300**
7	Cliente, CMS, Incremental	99,25	✗ **511,94**	1,7665	0,00021	0,05184	0,01069	234,37200
8	Cliente, CMS, Paralelo geração jovem	99,21	✗ **511,94**	1,5297	0,00019	✓ **0,02768**	0,01243	194,04800
9	Servidor	89,68	419,81	25,5765	0,00920	0,52651	0,22043	247,93300
10	Servidor, Serial	97,19	494,94	6,0244	0,00819	0,54293	0,25094	214,12700
11	Servidor, Paralelo geração estável	91,87	✓ **368,64**	17,2568	0,00466	6,13240	0,13548	212,13219
12	Servidor, CMS	✓ **99,30**	✗ **511,94**	1,5724	✓ **0,00017**	0,03329	0,01285	224,17700
13	Servidor, CMS, Remarcação paralela	99,27	✗ **511,94**	1,5513	✓ **0,00017**	0,03396	0,01273	213,31400
14	Servidor, CMS, Incremental	99,21	✗ **511,94**	1,6854	0,00023	0,03501	✓ **0,01014**	214,57400
15	Servidor, CMS, Paralelo geração jovem	99,27	✗ **511,94**	1,5548	0,00019	0,03696	0,01270	212,02800

Tabela 4.8 – Valores médios obtidos na execução do GeneId para os indicadores avaliados em uma *heap* de até 1024MB

a) 1 *Thread - Heap de até 1024MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	95,65	316,63	2,3534	✗ **0,00818**	0,17304	0,02977	54,09200
2	Cliente, Paralelo	95,41	293,94	2,4814	0,00800	0,25820	0,02457	54,07300
3	Cliente, Paralelo geração jovem	✓ **98,45**	✗ **455,23**	✓ **0,8165**	0,00656	0,14036	0,03371	52,65000
4	Cliente, Paralelo geração estável	✗ **82,46**	312,94	✗ **10,9513**	0,00348	3,41475	✗ **0,06972**	✗ **62,45200**
5	Cliente, CMS	97,96	365,21	1,0765	0,00017	✓ **0,02220**	0,00884	52,76500
6	Cliente, CMS, Remarcação paralela	97,94	320,50	1,0970	0,00016	0,03120	0,00932	53,14100
7	Cliente, CMS, Incremental	97,60	367,48	1,2865	0,00017	0,02623	✓ **0,00773**	53,52900
8	Cliente, CMS, Paralelo geração jovem	97,96	320,50	1,0831	0,00019	0,02855	0,00917	53,11900
9	Servidor	95,69	304,25	2,3316	0,00664	0,18070	0,02380	54,08200
10	Servidor, Serial	98,03	454,88	1,0300	0,00806	0,16064	0,04276	✓ **52,41400**
11	Servidor, Paralelo geração estável	86,91	✓ **289,38**	7,7831	0,00452	✗ **3,84310**	0,04774	59,45100
12	Servidor, CMS	98,01	360,98	1,0446	✓ **0,00015**	0,02644	0,00866	52,58500
13	Servidor, CMS, Remarcação paralela	97,89	320,50	1,1210	0,00017	0,03114	0,00943	53,08900
14	Servidor, CMS, Incremental	97,63	364,43	1,2647	✓ **0,00015**	0,02854	0,00785	53,27500
15	Servidor, CMS, Paralelo geração jovem	97,99	334,58	1,0598	0,00024	0,02749	0,00919	52,71100

b) 2 *Threads - Heap de até 1024MB*

Conf.	Coletor e opções	*Throughput (%)*	*Footprint (MB)*	Tempos de pausa (seg) acumulado	menor	maior	média	Tempo total
1	Cliente	97,99	377,75	3,3113	0,00966	0,03685	0,03938	✓ **165,13210**
2	Cliente, Paralelo	97,49	407,75	4,3316	✗ **0,00970**	0,24587	0,04557	172,51000
3	Cliente, Paralelo geração jovem	99,40	✗ **647,94**	1,2814	0,00718	0,27350	0,06420	213,39400
4	Cliente, Paralelo geração estável	✗ **95,21**	348,25	✗ **10,4648**	0,00642	✗ **4,13914**	✗ **0,07270**	218,43500
5	Cliente, CMS	99,25	434,09	1,3167	✓ **0,00016**	0,02795	0,01109	176,50300
6	Cliente, CMS, Remarcação paralela	99,40	434,09	✓ **1,2584**	0,00018	0,02887	0,01076	210,11600
7	Cliente, CMS, Incremental	99,11	488,73	1,5335	0,00017	0,03701	✓ **0,00961**	171,39300
8	Cliente, CMS, Paralelo geração jovem	99,34	479,11	1,3165	0,00019	0,02733	0,01111	198,66100
9	Servidor	97,95	351,94	3,8631	0,00319	0,41297	0,02699	188,62100
10	Servidor, Serial	99,33	511,65	1,6164	0,00800	0,28666	0,06987	✗ **239,47400**
11	Servidor, Paralelo geração estável	97,31	332,81	4,9984	0,00253	1,25499	0,02758	186,13900
12	Servidor, CMS	99,24	471,32	1,4186	0,00017	0,03394	0,01183	185,66400
13	Servidor, CMS, Remarcação paralela	99,33	477,04	1,3252	0,00018	0,02818	0,01119	196,89200
14	Servidor, CMS, Incremental	99,15	✓ **190,05**	1,5862	0,00019	✓ **0,02658**	0,00964	185,76400
15	Servidor, CMS, Paralelo geração jovem	✓ **99,43**	477,03	1,3251	0,00019	0,03187	0,01118	231,07300

c) 3 *Threads - Heap de até 1024MB*

Conf.	Coletor e opções	*Throughput (%)*	*Footprint (MB)*	Tempos de pausa (seg) acumulado	menor	maior	média	Tempo total
1	Cliente	97,29	✗ **454,69**	6,0815	✗ **0,08510**	0,46882	0,05532	224,53500
2	Cliente, Paralelo	97,29	481,00	5,1531	0,00829	0,48551	0,04855	✓ **190,20900**
3	Cliente, Paralelo geração jovem	99,25	✓ **819,46**	1,6231	0,00650	0,37621	0,08500	217,42200
4	Cliente, Paralelo geração estável	96,42	477,81	7,0569	0,00932	1,06211	0,06650	197,14000
5	Cliente, CMS	99,33	589,74	1,5654	0,00018	0,03242	0,01280	234,41400
6	Cliente, CMS, Remarcação paralela	99,24	594,25	1,5865	0,00018	0,04634	0,01292	209,37500
7	Cliente, CMS, Incremental	99,12	603,14	1,7132	0,00020	0,03188	0,01078	195,04300
8	Cliente, CMS, Paralelo geração jovem	99,27	590,02	1,5387	0,00018	0,03234	0,01253	210,39400
9	Servidor	96,93	456,94	6,3651	0,00771	0,47462	0,05324	207,45300
10	Servidor, Serial	99,07	520,70	2,1132	0,00940	0,43189	✗ **0,08760**	227,16100
11	Servidor, Paralelo geração estável	✓ **95,62**	455,81	✗ **9,3521**	0,00697	✗ **1,99296**	0,06539	213,53000
12	Servidor, CMS	✗ **99,36**	591,51	1,5158	✓ **0,00017**	0,03499	0,01236	✗ **235,06500**
13	Servidor, CMS, Remarcação paralela	99,30	533,66	✓ **1,4832**	0,00026	✓ **0,03154**	0,01237	212,35400
14	Servidor, CMS, Incremental	99,11	602,31	1,7132	0,00020	0,03221	✓ **0,01030**	191,43100
15	Servidor, CMS, Paralelo geração jovem	99,23	589,45	1,5468	✓ **0,00017**	0,03487	0,01259	201,32300

Os coletores CMS (5 e 12), CMS com remarcação em paralelo (6 e 13) e CMS paralelo na geração jovem (8 e 15) não se comportaram como os outros coletores, mantendo praticamente todos os objetos no éden. Isso resultou na ausência de coletas maiores, fazendo com o que os tempos de pausa se mantivessem baixos. Os coletores com paralelismo na geração jovem obtiveram resultados ainda melhores. Este comportamento não foi observado no coletor CMS incremental (7 e 14), que manteve a geração estável grande e ocupada. O número de coletas completas é superior ao outros coletores, mas, diferentemente do que tem sido observado nas outras aplicações, o modo incremental não obteve os piores resultados, ficando com valores próximos da média dos coletores CMS.

O coletor servidor paralelo (9) comportou-se como o coletor paralelo cliente. As diferenças vieram para o coletor servidor paralelo na geração estável (11). Com esta configuração, assim como no respectivo coletor cliente (4), os resultados foram inferiores. O coletor servidor serial (10) teve a execução mais demorada para duas threads, mas se comportou dentro dos padrões em relação às outras configurações.

As figuras 4.17, 4.18 e 4.19 sumarizam os desempenhos de footprint, throughput e tempo de execução para o GeneId, considerando uma heap de 512MB. Os mesmos indicadores são apresentados nas figuras 4.20, 4.21 e 4.22 para o novo tamanho da heap de 1024MB.

Após a análise dos gráficos apresentados é possível perceber que os coletores CMS mantiveram os maiores *throughputs,* indicando a menor interferência na execução. O melhor gerenciamento de memória ficou com os coletores paralelos na geração estável, mas estes apresentaram os piores *throughputs*. O coletor serial foi o mais constante para todos os números de *threads* com os dois tamanhos de *heap,* mantendo um *throughput* alto e consumo mediano de memória.

4.4.7. JGFSORBench

A demanda por memória do aplicativo JGFSORBench é pequena, aproximadamente 30MB. Sendo assim, o tamanho máximo da *heap* foi limitado em 128MB na tentativa de eliminar as limpezas durante a execução para avaliar somente o processo inicial de expansão da *heap*, já que não estamos definindo um limite inferior para a mesma. Esse objetivo foi praticamente alcançado em todas as configurações entre coletores e opções. A tabela 4.9 apresenta os valores médios obtidos durante as execuções deste aplicativo, utilizando uma, duas e quatro *threads.*

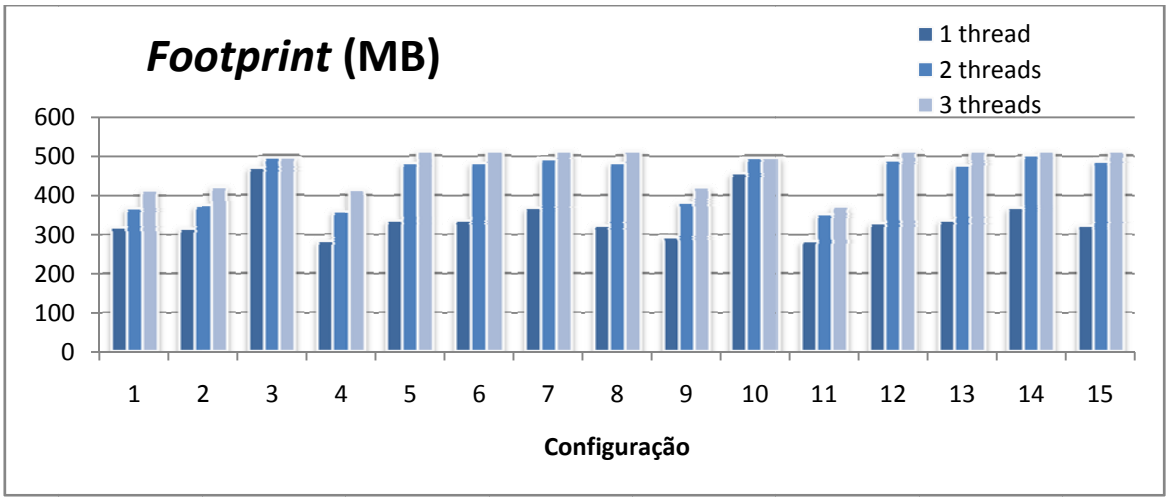

Figura 4.17 – Comparativo entre os *footprints* do aplicativo GeneId para *heap* de 512MB

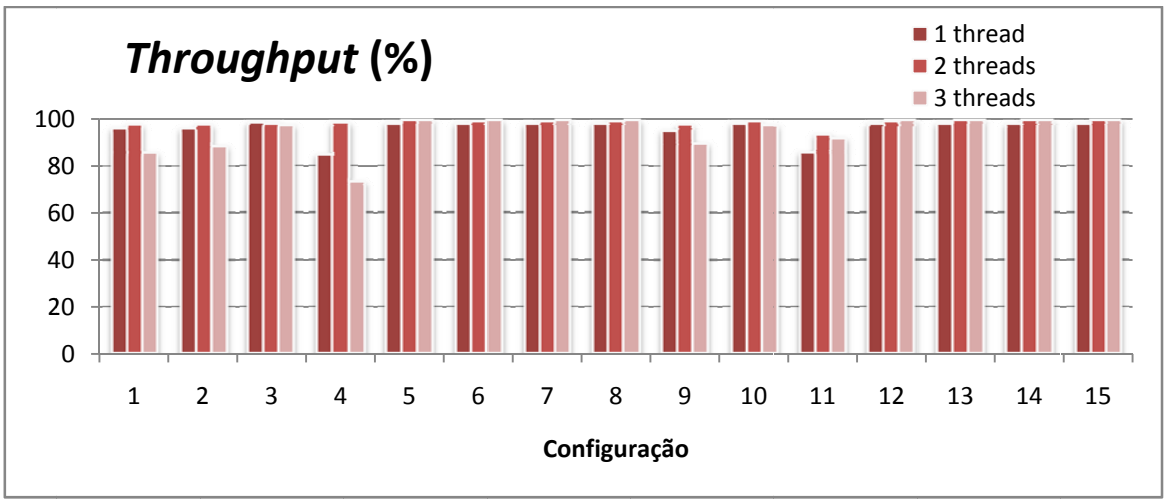

Figura 4.18 – Comparativo entre os *throughputs* do aplicativo GeneId para *heap* de 512MB

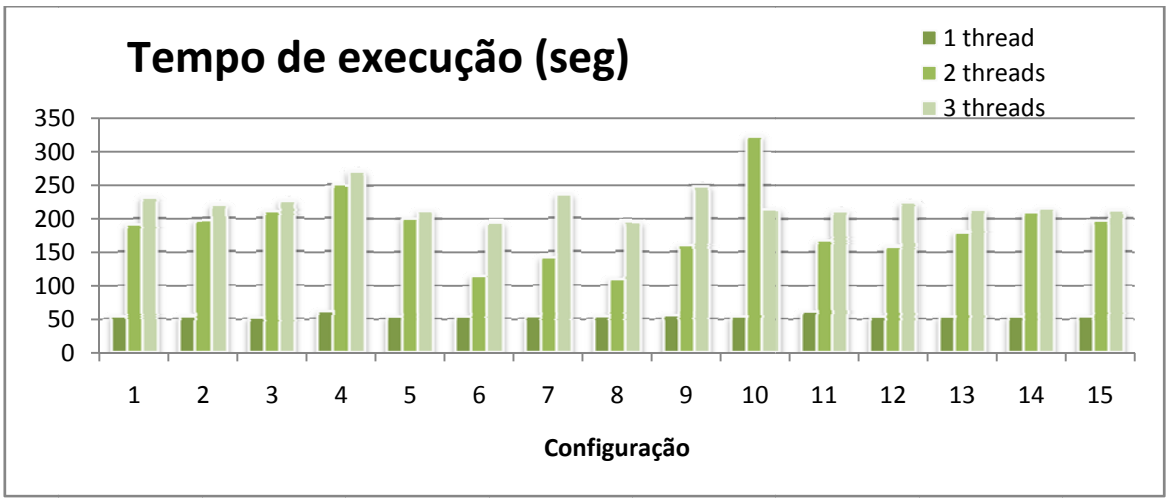

Figura 4.19 – Comparativo dos tempos de execução do aplicativo GeneId para *heap* de 512MB

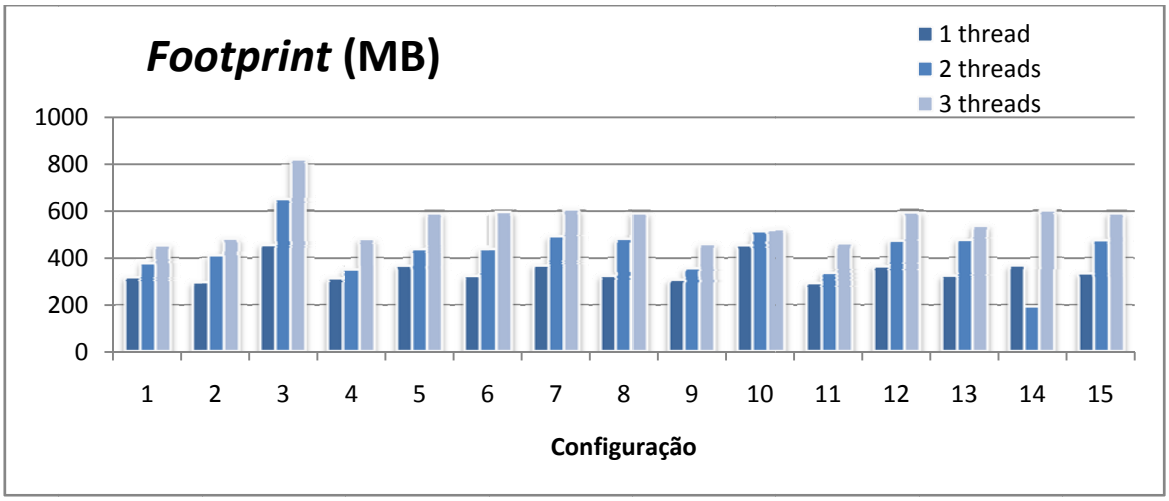

Figura 4.20 – Comparativo entre os *footprints* do aplicativo GeneId para *heap* de 1024MB

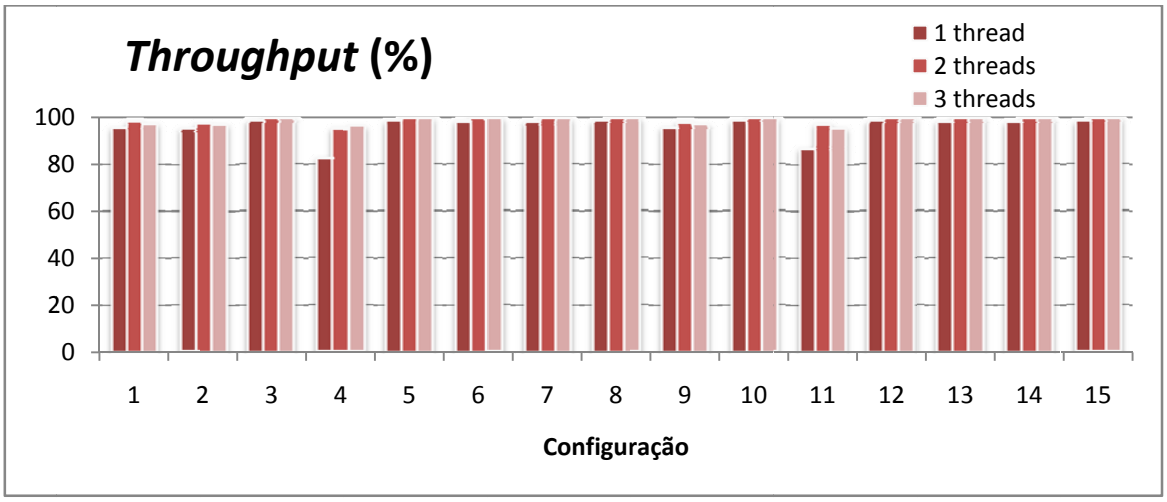

Figura 4.21 – Comparativo entre os *throughputs* do aplicativo GeneId para *heap* de 1024MB

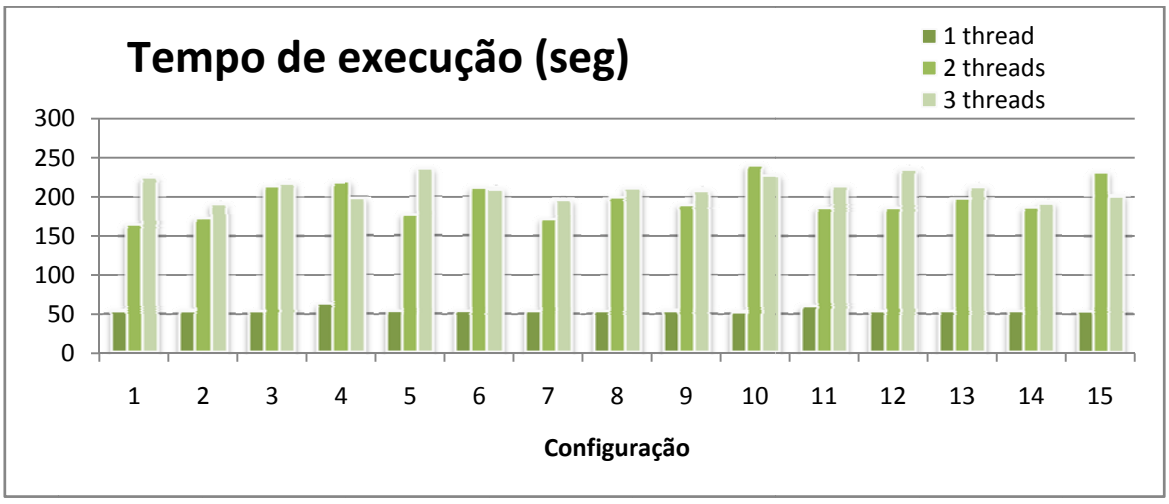

Figura 4.22 – Comparativo dos tempos de execução do aplicativo GeneId para *heap* de 1024MB

O coletor cliente serial (1) e o coletor cliente paralelo (2) realizaram o processo de expansão da heap de maneira semelhante. Foram quatro ou cinco coletas menores onde cada uma delas foi imediatamente seguida por uma coleta completa. Esse processo inicial fez com que a heap atingisse, em média, 64,5MB de tamanho, mas somente 45% desse total foram utilizados. O footprint alcançado por estes coletores foram os maiores entre as quinze configurações diferentes, demonstrando um grande desperdício de memória. Nenhum outro processo de limpeza é executado até o final, confirmando que não é necessário mais memória e atingindo o objetivo proposto para esta aplicação.

O tempo de execução deste aplicativo no coletor serial (4,04 segundos) foi o maior de todos para duas threads, ficando quase um segundo acima da média (que é 3,14 segundos). Todos os throughputs ficaram abaixo da média, como conseqüência dos maiores tempos de execução.

O coletor cliente paralelo na geração jovem (3) atingiu o tamanho necessário de heap mais rápido que os outros coletores, com apenas três coletas menores e três coletas completas. Como conseqüência, este coletor obteve os menores tempos de execução, salvo para a aplicação com quatro threads, onde o valor atingido ficou 10% maior do que o melhor resultado. O footprint deste coletor foi 47,28MB para todas as execuções, um valor baixo se comparado com os apresentados anteriormente.

O coletor cliente paralelo na geração estável (4) demorou um pouco mais para levar a heap ao tamanho mínimo para a execução. Como conseqüência teve um número de coletas um pouco maior (em média seis coletas menores e três coletas completas) e um tempo acumulado de limpeza superior, fazendo o throughput cair. O footprint ficou em 46,7MB, um valor menos abusivo em relação à memória necessária.

Os coletores CMS no modo cliente e servidor (5 e 12), CMS com remarcação em paralelo (6 e 13) e com coleta paralela na geração jovem (8 e 15) definiram o tamanho necessário para a heap com apenas quatro coletas menores e duas completas, com os menores tempos de pausa para os dados analisados. A rápida convergência para o valor ideal é resultado das limpezas mais rápidas, fazendo com que as modificações nos tamanhos fossem realizadas mais cedo. O footprint atingido foi o mesmo para todos estes coletores em todas as execuções: 55,3MB, um valor 80% maior do que o necessário.

O tempo total de execução destes coletores não foram os melhores porque todos eles iniciaram um processo de limpeza concorrente próximo do fim, processos estes que não foram concluídos a tempo. Durante esses processos, o tamanho da geração estável e da heap foram incrementados, e a geração jovem passou a ser menor.

Tabela 4.9 – Valores médios obtidos na execução do JGFSORBench para os indicadores avaliados em uma *heap* de até 128MB

a) Serial - *Heap de até 128MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	98,11	✗ 67,13	0,1132	0,0048	0,0226	0,0091	5,9770
2	Cliente, Paralelo	98,11	67,00	0,1132	0,0049	✗ 0,0236	0,0089	5,9770
3	Cliente, Paralelo geração jovem	98,97	47,30	0,0613	0,0035	0,0170	0,0089	✓ 5,9420
4	Cliente, Paralelo geração estável	✗ 97,65	44,19	✗ 0,1413	0,0047	0,0220	✗ 0,0124	6,0080
5	Cliente, CMS	98,97	55,30	0,0613	✓ 0,0002	0,0135	0,0050	5,9730
6	Cliente, CMS, Remarcação paralela	98,94	55,30	0,0633	✓ 0,0002	0,0135	✓ 0,0049	5,9620
7	Cliente, CMS, Incremental	✓ 99,78	✓ 35,88	✓ 0,0132	✓ 0,0002	0,0087	0,0072	5,9620
8	Cliente, CMS, Paralelo geração jovem	98,79	55,30	0,0721	✗ 0,0067	0,0143	0,0051	5,9610
9	Servidor	98,40	64,63	0,1014	0,0050	0,0231	0,0090	✗ 6,3470
10	Servidor, Serial	98,77	47,30	0,0732	0,0051	0,0168	0,0095	5,9480
11	Servidor, Paralelo geração estável	97,81	44,38	0,1313	0,0047	0,0205	0,0119	6,0020
12	Servidor, CMS	98,91	55,35	0,0654	✓ 0,0002	0,0133	0,0050	5,9750
13	Servidor, CMS, Remarcação paralela	98,94	55,35	0,0632	✓ 0,0002	0,0135	✓ 0,0049	5,9680
14	Servidor, CMS, Incremental	99,15	✓ 35,88	0,0510	✓ 0,0002	✓ 0,0085	0,0070	5,9680
15	Servidor, CMS, Paralelo geração jovem	98,75	55,35	0,0750	✓ 0,0002	0,0135	0,0051	5,9770

b) 2 *Threads - Heap de até 128MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	97,01	63,38	0,1210	0,0049	0,0177	0,0081	✗ 4,0460
2	Cliente, Paralelo	95,36	✗ 66,63	0,1321	✗ 0,0053	0,0222	0,0091	2,8480
3	Cliente, Paralelo geração jovem	97,71	47,28	0,0646	0,0034	0,0172	0,0089	✓ 2,8160
4	Cliente, Paralelo geração estável	95,42	44,38	0,1320	0,0049	0,0205	0,0122	2,8810
5	Cliente, CMS	97,91	55,30	0,0665	✓ 0,0002	0,0135	0,0059	3,1760
6	Cliente, CMS, Remarcação paralela	97,73	55,30	0,0715	✓ 0,0002	0,0135	0,0060	3,1560
7	Cliente, CMS, Incremental	✓ 98,23	✓ 35,88	✓ 0,0514	✓ 0,0002	0,0100	0,0076	2,9030
8	Cliente, CMS, Paralelo geração jovem	97,75	55,30	0,0765	✓ 0,0002	0,0134	0,0060	3,3970
9	Servidor	✗ 94,81	66,06	✗ 0,1654	0,0046	0,0223	0,0088	3,1840
10	Servidor, Serial	98,03	47,30	0,0787	0,0050	0,0169	0,0096	4,0040
11	Servidor, Paralelo geração estável	95,03	44,13	0,1436	0,0049	✗ 0,0237	✗ 0,0127	2,8900
12	Servidor, CMS	97,90	55,30	0,0620	✓ 0,0002	0,0134	✓ 0,0058	2,9500
13	Servidor, CMS, Remarcação paralela	97,56	55,30	0,0756	✓ 0,0002	0,0135	0,0060	3,1060
14	Servidor, CMS, Incremental	98,15	✓ 35,88	0,0535	✓ 0,0002	✓ 0,0085	0,0071	2,8850
15	Servidor, CMS, Paralelo geração jovem	97,47	55,30	0,0735	0,0003	0,0133	0,0059	2,9100

c) 4 *Threads - Heap de até128MB*			Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	95,42	62,94	0,1232	0,0048	0,0167	0,0080	2,6920
2	Cliente, Paralelo	95,10	✗ **66,25**	0,1321	0,0049	✗ **0,0225**	0,0089	2,6980
3	Cliente, Paralelo geração jovem	97,63	47,29	0,0632	0,0039	0,0179	0,0091	2,6630
4	Cliente, Paralelo geração estável	95,18	45,50	0,1332	✗ **0,0054**	0,0181	0,0116	✗ **2,7610**
5	Cliente, CMS	97,56	55,30	0,0656	✓ **0,0002**	0,0134	0,0059	2,6900
6	Cliente, CMS, Remarcação paralela	97,28	55,30	0,0732	0,0003	0,0135	0,0059	2,6900
7	Cliente, CMS, Incremental	✓ **98,07**	✓ **35,88**	✓ **0,0515**	✓ **0,0002**	✓ **0,0087**	0,0072	2,6700
8	Cliente, CMS, Paralelo geração jovem	97,35	55,30	0,0713	✓ **0,0002**	0,0135	0,0060	2,6950
9	Servidor	✗ **94,58**	63,44	✗ **0,1456**	0,0047	0,0159	0,0080	2,6870
10	Servidor, Serial	97,31	47,30	0,0715	0,0050	0,0169	0,0096	2,6590
11	Servidor, Paralelo geração estável	94,60	44,46	0,1310	✗ **0,0054**	0,0198	✗ **0,0119**	✓ **2,4240**
12	Servidor, CMS	97,68	55,30	0,0631	✓ **0,0002**	0,0134	0,0059	2,7150
13	Servidor, CMS, Remarcação paralela	97,28	55,30	0,0731	0,0003	0,0135	0,0059	2,6850
14	Servidor, CMS, Incremental	97,93	35,88	0,0556	✓ **0,0002**	✓ **0,0087**	0,0073	2,6840
15	Servidor, CMS, Paralelo geração jovem	97,66	55,30	0,0631	✓ **0,0002**	0,0136	✓ **0,0058**	2,6960

O destaque neste aplicativo ficou para o coletor CMS incremental cliente e servidor (7 e 14, respectivamente). Este coletor não iniciou um processo concorrente próximo ao final da execução, e fez com que o tempo acumulado de pausas atingisse seus menores valores. Sem este último processo, o tamanho da heap não foi incrementado desnecessariamente, mantendo seu valor em 35,88MB, próximo do ideal. Todas essas diferenças resultaram nos melhores throughputs, fazendo deste coletor a melhor opção para este aplicativo.

Os coletores servidor paralelo (9) e servidor paralelo na geração estável (11) não apresentaram bons resultados com as execuções de duas e quatro threads. O coletor servidor paralelo (9) obteve os piores throughputs e tempos de pausas acumulados para estes cenários, além de apresentar um valor alto para o footprint. O comportamento é resultado de algumas coletas completas demoradas logo no início da execução, ainda na fase de alongamento da heap. O coletor servidor serial (10) teve comportamento normal para a aplicação, com tempos totais muito próximos das médias.

As figuras 4.23, 4.24 e 4.25 comparam respectivamente os dados de footprint, throughput e tempo de execução; avaliados para os três diferentes números de threads em execução.

Os gráficos comprovam a superioridade dos coletores incrementais, que obtiveram os melhores resultados para esta aplicação.

Uma característica marcante nas execuções foi a grande semelhança nos valores de footprint para todos os coletores, que praticamente não foram modificados em função do número de threads. No geral, os melhores throughputs foram para as execuções seriais, já que elas tiveram os maiores tempos de execução. A distribuição dos tempos foi bem equiparada para o mesmo número de threads, com exceção das configurações 1 e 10 para duas threads em execução, onde os tempos foram superiores ao esperado.

4.4.8. JGFMonteCarloBench

O consumo de memória para a aplicação JGFMonteCarloBench cresce linearmente em função do tempo e independe do número de *threads* em execução. As alocações vão sendo realizadas e os processos de limpeza são chamados, eliminando alguns objetos e mantendo os outros em memória. Isso faz com que o consumo máximo seja próximo do fim, onde a *heap* deve atingir o seu *footprint*.

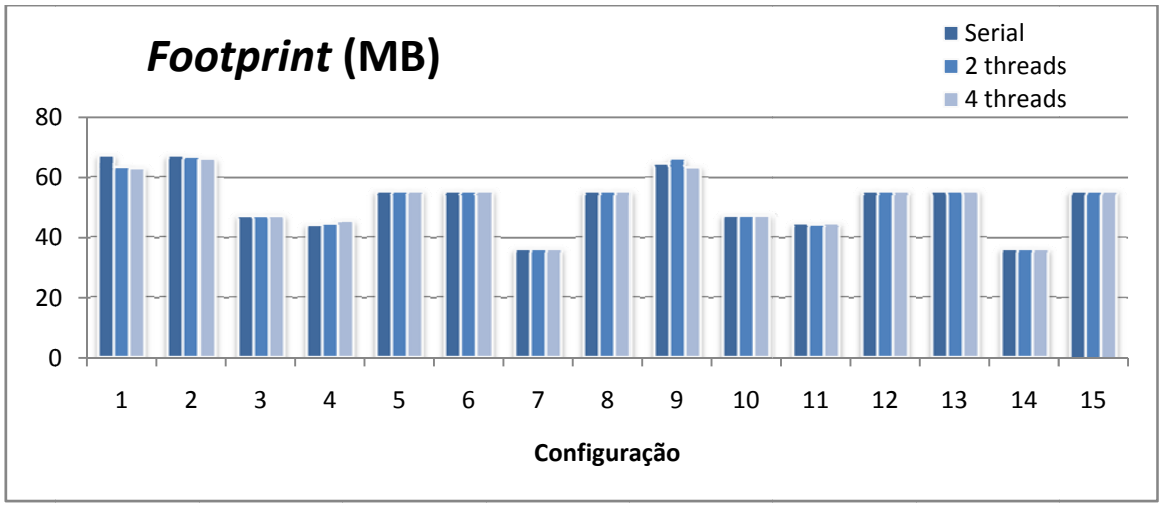

Figura 4.23 – Comparativo entre os *footprints* do aplicativo JGFSORBench

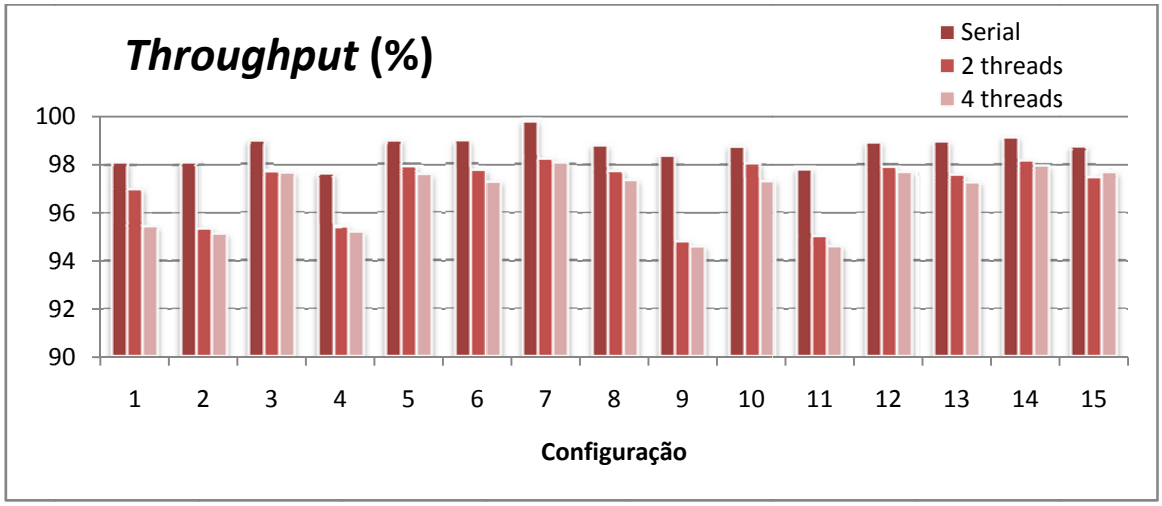

Figura 4.24 – Comparativo entre os *throughputs* do aplicativo JGFSORBench

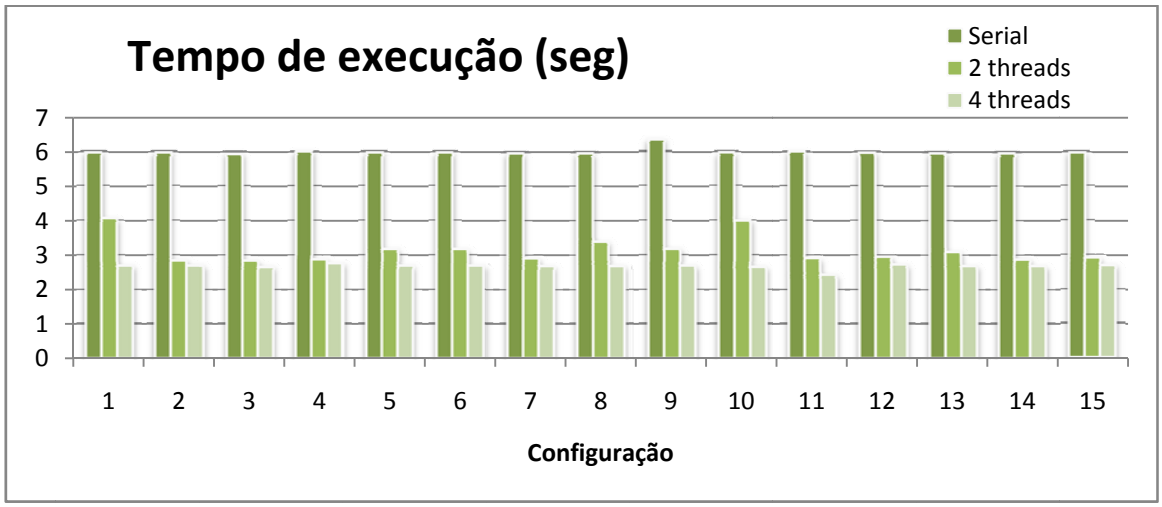

Figura 4.25 – Comparativo entre os tempos de execução do aplicativo JGFSORBench

Para a entrada utilizada, o valor máximo necessário é de aproximadamente 83MB, mas durante a execução são alocados quase 300MB de objetos. Sendo assim, o coletor deve ser capaz de liberar pouco mais de 200MB neste intervalo de tempo. O tamanho total da heap foi definido para 256MB. A tabela 4.10 contém os valores médios dos indicadores desta aplicação, analisados para execuções com uma, duas e quatro threads.

Durante a execução, o coletor cliente serial (1) manteve o tamanho do éden em aproximadamente 40% do total da heap. A distribuição das coletas foi constante, com uma limpeza completa a cada três ou quatro coletas menores, sempre resultando num aumento do tamanho total da heap e redistribuição desse espaço entre as gerações. O footprint médio entre as execuções com todos os números de threads ficou em 152MB, valor 83% maior do que o necessário.

O coletor cliente paralelo (2) teve o comportamento semelhante ao coletor serial. A principal diferença ficou por conta do footprint, que para duas e quatro threads foi o maior entre todas as configurações analisadas. Para uma única thread, esse valor máximo foi inferior ao coletor serial, chegando a apenas 122MB.

Para o coletor cliente paralelo na geração jovem (3), o tamanho da geração ficou, em média, com 60% do tamanho total da heap. Assim, as coletas em paralelo nesta geração foram mais vantajosas, já que esse comportamento reduziu o número de limpezas completas e fez com que o tamanho da heap fosse modificado nos momentos mais propícios. O footprint médio para as três execuções com diferentes números de threads ficou em 140MB.

Esse mesmo comportamento não foi observado para os mecanismos cliente e servidor paralelos na geração estável (4 e 11). Para estas configurações, o tamanho da geração estável foi mais representativo na heap e fez com que mais coletas menores ocorressem entre as limpezas completas. Essa modificação acarretou em aumento nos tempos de pausa, já que a geração estável se tornou maior e é mais demorado para analisá-la por completo. As pausas chegaram a quase 0,2 segundos, representando os maiores valores entre os coletores. Com isso, o tempo total de execução, o tempo médio de pausas e o tempo acumulado cresceram, resultando nos piores resultados deste aplicativo.

Os coletores servidor paralelo e serial (9 e 10) comportaram-se como os seus correspondentes no modo cliente (2 e 1).

Tabela 4.10 – Valores médios obtidos na execução do JGFMonteCarloBench para os indicadores avaliados em uma *heap*de até 256MB

a) Serial - *Heap de até256MB*			Tempos de pausa (seg)				
Conf. Coletor e opções	*Throughput (%)*	*Footprint (MB)*	acumulado	menor	maior	média	Tempo total
1 Cliente	94,00	163,13	0,4213	0,00242	0,06200	0,01434	7,02000
2 Cliente, Paralelo	88,66	121,75	0,7923	0,00175	0,07860	0,01436	6,98800
3 Cliente, Paralelo geração jovem	93,76	163,13	0,4321	0,00242	0,06200	0,01434	6,92500
4 Cliente, Paralelo geração estável	94,59	✗ 167,81	0,3954	0,00235	✓ 0,06190	0,01352	✗ 7,30200
5 Cliente, CMS	96,29	105,18	0,2532	0,00013	0,08141	0,00358	6,82200
6 Cliente, CMS, Remarcação paralela	96,26	105,40	0,2564	✓ 0,00010	0,08001	0,00357	6,84900
7 Cliente, CMS, Incremental	95,84	140,02	0,2832	0,00020	0,08184	✓ 0,00226	6,80800
8 Cliente, CMS, Paralelo geração jovem	✓ 96,39	105,24	✓ 0,2468	0,00012	0,07978	0,00353	6,83000
9 Servidor	93,81	158,63	0,4321	✗ 0,00250	0,06691	0,01371	6,98200
10 Servidor, Serial	94,59	119,89	0,3768	0,00101	0,06930	0,01236	6,96100
11 Servidor, Paralelo geração estável	✗ 88,58	129,69	✗ 0,8232	0,00175	0,08035	✗ 0,01543	7,20700
12 Servidor, CMS	96,14	105,28	0,2632	✓ 0,00010	✗ 0,08444	0,00375	6,82500
13 Servidor, CMS, Remarcação paralela	96,23	105,40	0,2564	✓ 0,00010	0,08001	0,00357	✓ 6,79500
14 Servidor, CMS, Incremental	95,95	143,43	0,2868	0,00024	0,08140	0,00227	7,07600
15 Servidor, CMS, Paralelo geração jovem	92,23	✓ 102,75	0,5310	0,00013	0,08101	0,00360	6,83800

b) 2 *Threads - Heap de até 256MB*			Tempos de pausa (seg)				
Conf. Coletor e opções	*Throughput (%)*	*Footprint (MB)*	acumulado	menor	maior	média	Tempo total
1 Cliente	91,67	135,94	0,3284	0,00172	0,03556	0,01193	3,94300
2 Cliente, Paralelo	90,07	✗ 166,81	0,3932	0,00197	0,07076	0,01388	3,95800
3 Cliente, Paralelo geração jovem	90,68	127,22	0,3618	0,00073	0,08062	0,01074	3,88400
4 Cliente, Paralelo geração estável	81,59	114,13	0,7812	0,00194	0,15442	0,01662	4,24400
5 Cliente, CMS	93,39	100,25	0,2521	0,00014	0,08058	0,00354	✓ 3,81400
6 Cliente, CMS, Remarcação paralela	93,28	105,40	0,2568	✓ 0,00010	0,08001	0,00357	3,82100
7 Cliente, CMS, Incremental	✓ 95,12	✓ 87,88	✓ 0,1868	✗ 0,00265	0,00953	0,00308	3,82500
8 Cliente, CMS, Paralelo geração jovem	93,43	100,23	0,2512	0,00012	0,08146	0,00366	3,82200
9 Servidor	88,35	159,06	0,4640	0,00232	0,07490	0,01424	3,98200
10 Servidor, Serial	90,25	120,70	0,3832	0,00096	0,07343	0,01276	3,93000
11 Servidor, Paralelo geração estável	✗ 78,96	128,25	✗ 0,9235	0,00197	✗ 0,19605	✗ 0,01847	✗ 4,39000
12 Servidor, CMS	93,30	100,22	0,2565	0,00013	0,08061	0,00364	3,82800
13 Servidor, CMS, Remarcação paralela	93,52	105,40	0,2532	✓ 0,00010	0,08001	0,00357	3,90500
14 Servidor, CMS, Incremental	95,11	✓ 87,88	0,1869	0,00036	✓ 0,00944	✓ 0,00302	3,81800
15 Servidor, CMS, Paralelo geração jovem	93,21	100,24	0,2616	0,00014	0,08078	0,00379	3,85200

c) 4 *Threads - Heap de até 256MB*				Tempos de pausa (seg)				
Conf.	**Coletor e opções**	***Throughput (%)***	***Footprint (MB)***	**acumulado**	**menor**	**maior**	**média**	**Tempo total**
1	Cliente	83,96	156,81	0,39310	0,00197	0,07793	0,01449	2,45100
2	Cliente, Paralelo	83,66	✗ **168,33**	0,39123	0,00182	0,07151	✗ **0,14270**	2,39400
3	Cliente, Paralelo geração jovem	86,21	131,57	0,32651	0,00076	0,07770	0,01093	2,36700
4	Cliente, Paralelo geração estável	73,36	119,63	0,72510	0,00198	0,14703	0,01894	✗ **2,72200**
5	Cliente, CMS	88,95	97,93	0,26640	0,00016	0,08258	0,00364	2,41000
6	Cliente, CMS, Remarcação paralela	88,72	140,69	0,27322	0,00016	0,09184	0,00428	2,42300
7	Cliente, CMS, Incremental	92,29	✓ **87,69**	0,18351	0,00256	0,01020	0,00314	2,38000
8	Cliente, CMS, Paralelo geração jovem	88,60	142,56	0,27351	0,00015	0,08907	0,00389	2,39900
9	Servidor	83,73	167,63	0,39120	0,00232	0,07178	0,01450	2,40500
10	Servidor, Serial	83,72	125,18	0,38854	0,00102	0,08039	0,01310	2,38700
11	Servidor, Paralelo geração estável	✗ **73,29**	117,25	✗ **0,72684**	✗ **0,01860**	✗ **0,15774**	0,01767	2,72100
12	Servidor, CMS	92,15	✓ **87,69**	0,18310	✓ **0,00014**	0,01091	✓ **0,00254**	✓ **2,33100**
13	Servidor, CMS, Remarcação paralela	88,62	138,64	0,27560	0,00016	0,09184	0,00428	2,42120
14	Servidor, CMS, Incremental	✓ **92,34**	✓ **87,69**	✓ **0,18150**	0,00026	✓ **0,00980**	0,00310	2,37000
15	Servidor, CMS, Paralelo geração jovem	90,17	97,92	0,25330	0,00016	0,08177	0,00357	2,57700

Os coletores CMS tiveram comportamentos semelhantes para os modos cliente e servidor. O CMS *default* (5 e 12) e o CMS com remarcação em paralelo (6 e 13) apresentaram várias limpezas na geração jovem e algumas coletas completas, com constante modificação no tamanho da *heap*. O CMS com remarcação em paralelo foi caracterizado por obter os menores tempos de pausa. O CMS paralelo para a geração jovem (8 e 15), teve como única diferença o tamanho da geração jovem que passou a ser mais representativa no tamanho total da *heap* em relação aos CMS já apresentados. O CMS obteve dois entre os três melhores tempos de execução, sendo o outro atribuído ao coletor CMS com remarcação paralela.

Novamente, o destaque para esta aplicação é o coletor CMS incremental (7 e 14). Este coletor caracterizou-se por somente realizar coletas maiores no começo da execução. A partir daí todas as alocações são feitas na geração estável, que é a única que cresce com o decorrer do tempo. Isso resultou em tempo médio de pausa baixo e, conseqüentemente, num tempo acumulado pequeno. Uma das características mais importantes deste comportamento foi a precisão no tamanho da *heap*, que só foi modificado quando necessário. Este coletor teve *footprint* médio de 105MB, mas obteve os melhores valores para duas e quatro *threads,* com apenas 88MB alocados na *heap*.

As figuras 4.26, 4.27 e 4.28 mostram os gráficos comparativos de *footprint, throughput* e tempo de execução avaliados para os três diferentes números de *threads* em execução nesta aplicação.

Após uma análise dos gráficos é possível perceber que o tempo de execução manteve um bom padrão de comportamento para todas as execuções. Os valores de *throughput* mantiveram-se altos, com exceção dos coletores paralelos para geração estável. Os melhores valores foram obtidos com os coletores CMS. Em relação ao *footprint*, as execuções seriais tiveram um consumo médio superior às execuções paralelizadas, enquanto para estas últimas os melhores valores ficaram com os coletores CMS incrementais.

Os melhores coletores para esta aplicação podem ser considerados os CMS incrementais, já que eles gerenciaram bem a memória, mantiveram alto o *throughput* e tiveram tempos de execução próximos da média.

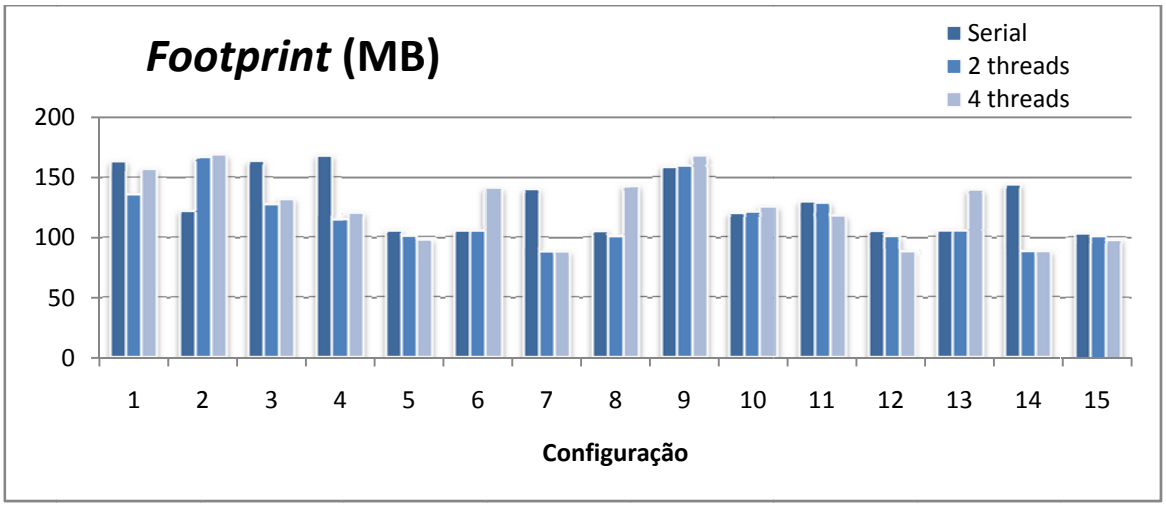

Figura 4.26 – Comparativo entre os *footprints* do aplicativo JGFMonteCarloBench

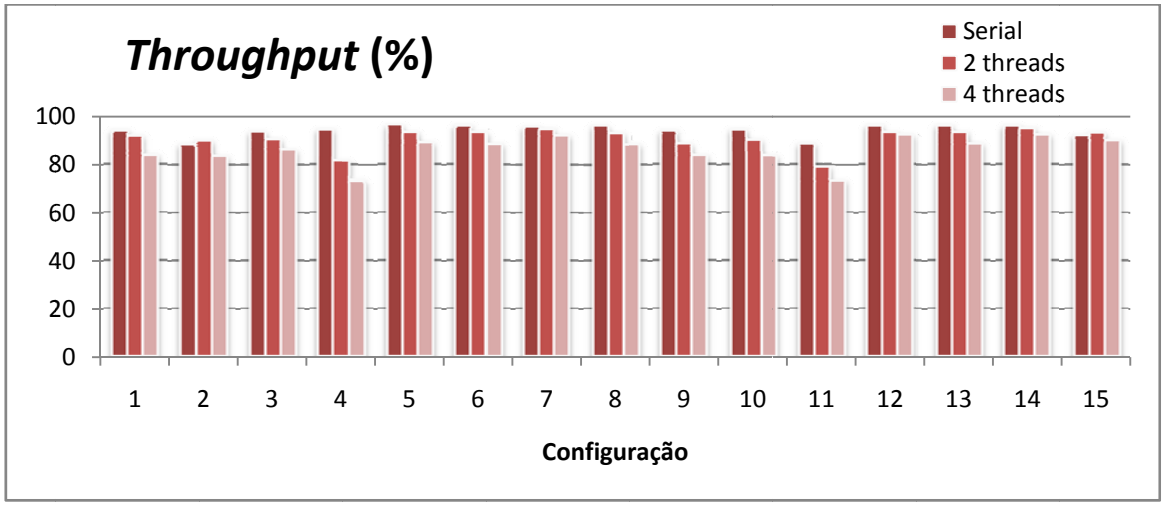

Figura 4.27 – Comparativo entre os *throughputs* do aplicativo JGFMonteCarloBench

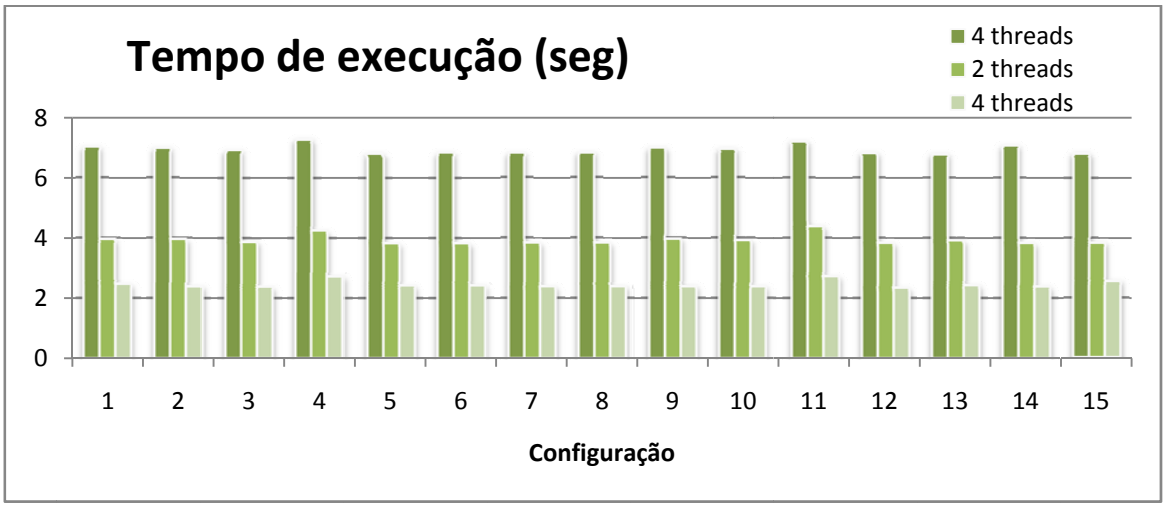

Figura 4.28 – Comparativo entre os tempos de execução do aplicativo JGFMonteCarloBench

4.5. Desempenho dos coletores

Como os coletores implementados na JVM são os mesmos para ambos os modos cliente e servidor, eles serão avaliados para as duas configurações em conjunto, já que a única diferença entre essas execuções foi a otimização dos *bytecodes* durante a compilação, o que não modifica a heurística existente no coletor.

O coletor serial (1 e 10) pode ser considerado um bom coletor para as aplicações avaliadas. Ele não obteve as melhores colocações dentre todos os coletores avaliados mas manteve alto os índices de desempenho para a maioria das aplicações. Este algoritmo sempre realiza mudanças no tamanho da *heap* e nos tamanhos das gerações durante um processo de limpeza, com o intuito de gerenciar bem o uso de memória. Como conseqüência, essas modificações introduzem algum custo adicional de tempo, fazendo com que os *throughputs* e tempos de execução não sejam sempre os menores valores atingíveis.

Outra característica deste coletor é que ele mantém uma geração estável quase sempre predominante na *heap*, deixando uma pequena porcentagem para a geração jovem. Essa propriedade faz com que as coletas maiores sejam menos freqüentes e mais demoradas. Isso é uma característica esperada deste tipo de coletor, já que ele foi desenvolvido para atender a todos os grupos de aplicações, que representam comportamentos de uso/alocação de memória variados. Essa característica não esteve presente nas aplicações JGFCreateBench e JGFForkJoinBench, onde o coletor foi capaz de identificar que a longevidade dos objetos é extremamente pequena e manteve proporções adequadas para esta situação.

Como o coletor serial é o padrão de máquinas clientes, a compilação dos *bytecodes* com a opção `-client` gera um código otimizado para este coletor, fazendo com que, no geral, os resultados da configuração 1 fossem melhores do que a configuração 10.

O coletor paralelo (2) teve comportamento parecido com o coletor serial no que diz respeito aos tamanhos das gerações e da *heap*. Para as aplicações *multithreads* JGFForkJoinBench, JGFSORBench, JGFMonteCarloBench e GeneId, este coletor obteve tempos individuais de pausas mais elevados do que os outros algoritmos, mas ainda manteve resultados de tempo total em valores médios.

As principais diferenças em relação ao coletor paralelo vêm com os coletores paralelos na geração jovem (3 e 9) e paralelo na geração estável (4 e 11). Entre eles é

possível perceber uma tendência natural de manter maior a geração onde o paralelismo foi implementado, como tentativa de usufruir do mesmo.

Essas modificações nem sempre foram eficientes. O paralelismo na geração jovem (3 e 9) obteve bons resultados nas aplicações onde o tempo médio de vida dos objetos é menor, já que nestas situações a coleta foi mais rápida e bastante espaço foi liberado. Entretanto, essa abordagem não se mostrou eficiente no JGFCreateBench, já que os objetos por ela criados ao longo de toda a execução possuem tempo de vida extremamente curto. O tamanho da *heap* ficou minúsculo, se comparado com a memória disponível, e não foi modificado pelo coletor, que se dava por satisfeito com o trabalho executado por cada coleta liberando toda a geração e um tempo muito pequeno. Os resultados finais tiveram um número excessivo de limpezas nesta geração, aumentando assim o tempo total de coleta. Os melhores valores deste coletor ficaram para o JGFSORBench, onde as poucas limpezas que ocorreram foram as responsáveis por copiar os objetos da geração jovem para a geração estável, beneficiando-se do paralelismo neste processo. Uma conseqüência do uso deste coletor são os baixos *footprints* de memória quando a aplicação realiza muitas alocações durante a execução.

O paralelismo na geração estável (4 e 11) foi pouco eficiente e a sua utilização é desencorajada para várias situações. Cinco dos oito aplicativos obtiveram os piores valores, para quase todos os indicadores, quando utilizaram este coletor.

O tamanho da geração estável muito superior à geração jovem faz que com que quase todos os objetos que sobrevivam a poucas coletas sejam promovidos. Assim, passamos a ter coletas completas mais freqüentes e demoradas, em conseqüência de mais memória a ser analisada e muito lixo a ser eliminado. Como vimos na seção 3.3.2, este algoritmo realiza compactação dos objetos sobreviventes a partir de um prefixo denso. Se muitos objetos viram lixo, esse prefixo fica muito pequeno e quase toda a geração vai ser compactada, um processo lento para este caso, onde a fragmentação é grande.

Os próximos coletores a serem analisados sãos os coletores concorrentes (CMS). Uma característica importante desses coletores é que eles atingiram os objetivos das menores pausas e mantiveram, na maioria dos casos, valores de *throughputs* maiores que a média entre todas as configurações avaliadas. Os tempos de execução, entretanto, foram os maiores para o conjunto de testes, com exceção das aplicações JGFSerialBench e GeneId.

O coletor CMS padrão (5 e 12), o coletor CMS com remarcação em paralelo (6 e 13) e o coletor CMS com coleta paralela na geração estável (8 e 15) se comportaram praticamente iguais e as diferenças entre eles foram muito pequenas. Eles utilizam a

manipulação de memória das gerações para atingir os seus objetivos. Sempre que há memória disponível e o tempo de pausa ainda é alto, eles aumentam a *heap*, tentando manter grande a geração jovem. Para os aplicativos GCOld e GCBench estes coletores fizeram o uso máximo da memória disponível com a intenção de manter os objetos mais tempo no éden ou no espaço de sobreviventes. Nas aplicações *multithreads* GeneId, JGFForkJoinBench, JGFSORBench, os algoritmos optaram por não modificar o tamanho da *heap* graças à ausência de limpezas completas e resultados de tempo satisfatórios. No JGFMonteCarloBench, novos objetos são alocados a todo momento e os antigos vão permanecendo na *heap*. Assim, os coletores promovem os objetos antigos, aumentam a *heap* e mantém a geração jovem pequena, para não aumentar o tempo de coleta. A aplicação JGFCreateBench executou poucas coletas completas, que foram responsáveis por copiar alguns objetos da geração jovem para a estável. Sendo assim, o tempo de pausa foi muito pequeno para uma modificação no tamanho da *heap*. No aplicativo JGFSerialBench, estes coletores aumentaram o tamanho da geração estável devido às promoções realizadas e o tamanho foi mantido para evitar um *overhead* na mudança, mesmo próximo do fim da execução, onde ocorre uma limpeza completa na *heap* que diminui o consumo de memória.

A grande diferença do coletor CMS no modo incremental (7 e 14) é o aumento no número de coletas maiores que ocasionam os resultados de tempos inferiores. Essas coletas são conseqüências diretas de constantes falhas no modo concorrente. Neste coletor, um processo de limpeza é iniciado antes que a geração jovem fique cheia e é executado em conjunto com a aplicação. Quando a aplicação tenta realizar uma alocação na geração jovem e não consegue, uma coleta menor é iniciada. Esta coleta pode promover objetos para a geração estável e o espaço necessário para o processo pode não ter sido liberado ainda, porque a *thread* responsável pela coleta não concluiu seu trabalho. Situações como esta provocam uma falha no modo concorrente, fazendo com que a aplicação seja pausada para a coleta completa ser concluída. Nestas situações o tamanho da geração estável é incrementado com bases estatísticas para evitar futuras falhas no processo.

Três das oito aplicações avaliadas (GCBench, GCOld e JGFCreateBench) obtiveram os piores resultados, no geral, com o modo incremental ativado. Este mecanismo foi desenvolvido para simular os coletores concorrentes já apresentados em máquinas com apenas um processador. Para os casos avaliados, este coletor fez constantes preempções do processo em execução para realizar uma parte da limpeza. Nestas três aplicações seriais, somente um dos quatro processadores disponíveis foi utilizado e compartilhado entre a aplicação e o coletor, provocando um atraso na execução geral da

aplicação. A aplicação JGFSerialBench também mostrou as mesmas características para este coletor, entretanto os resultados não foram os piores.

No aplicativo JGFMonteCarloBench, onde a demanda por memória pode ser aproximada por uma função linear, este coletor foi capaz de prever o tamanho necessário para as alocações, fazendo com que nesta aplicação não houvessem falhas no modo incremental e que limpezas completas fossem desnecessárias. Assim, foi possível manter a geração jovem grande o suficiente para atingir os objetivos de pausas e bons resultados gerais.

4.6. Reduzindo os custos da coleta de lixo

Após as análises de comportamento das oito aplicações selecionadas, são sugeridas algumas modificações nas opções existentes na JVM com o intuito de aprimorar a eficiência dos coletores.

É possível analisar em cada aplicativo o consumo inicial de memória e definir um valor inferior para o tamanho da *heap* através da opção `-Xms`. Sendo assim, aplicativos como o GCOld e o GCBench não precisariam da fase de inicialização, onde o principal objetivo é alongar a *heap* para as análises. Essa modificação também afetaria aplicativos como o GeneId, que inicialmente aloca cerca de 200MB de dados que serão mantidos durante toda a execução. Uma modificação deste tipo pode praticamente eliminaria os processos de coleta de lixo em aplicações científicas, onde praticamente não há demanda por memória durante a execução. Este é o caso do JGFSORBench, que se tivesse um tamanho inicial de 32MB e, se possível, metade deste espaço alocado na geração jovem, executaria somente algumas poucas limpezas, que seriam responsáveis por promover os objetos da geração jovem para a geração estável. É valido lembrar que o tamanho da geração jovem não pode exceder metade do tamanho total da *heap*. Para realmente eliminar as coletas seria necessário um tamanho inicial de 64MB ou superior, para que 50% desse valor fosse reservado à geração jovem, sendo suficiente para acomodar todos os objetos.

Alguns aplicativos testados, como o JGFCreateBench fazem praticamente uso só da geração jovem. Para estes aplicativos seria interessante manter um tamanho inicial desta geração o maior possível, o que faria com que coletas ocorressem com menos freqüência e ainda sim obteríamos bons resultados. Para esta aplicação, uma boa tentativa seria definir a proporção da geração estável para a geração jovem em 1:1, através da opção

`-XX:NewRatio=1` e definir um tamanho inicial da *heap* maior, por exemplo, 256MB (`-Xmx256m`).

Também é possível melhorar o desempenho dos coletores onde houve problemas com o tempo de coletas, como no aplicativo JGFMonteCarloBench, GeneId, JGFSerialBench, GCOld e GCBench. Uma boa solução é definir um tempo máximo de pausa para um valor pouco maior que o valor médio entre os resultados obtidos, fazendo com que nenhum processo de limpeza produza valores discrepantes. Essa modificação pode ser feita na linha de comando através da opção `-XX:MaxGCPauseMillis=<n>` É importante definir tempos de pausas que podem ser atingidos, já que valores pequenos farão a *heap* diminuir e aumentar o número de coletas, reduzindo ainda mais a eficiência.

No aplicativo GCBench, o comportamento anormal resultante de uma áspera mudança no padrão de alocação de objetos na *heap* pode ser amenizado através de modificações na opção `-XX:CMSExpAvgFactor=<n>`, que é a responsável por atribuir um "peso" nas informações da última coleta no cálculo dos dados estatísticos da JVM . Em outras palavras, definir um valor de *n* baixo indica que a JVM manterá estatísticas menos fiéis às últimas modificações na memória, diminuindo a gravidade da situação apresentada.

Outra sugestão de modificação é forçar o uso da memória disponível na *heap* durante a execução de aplicativos como o JGFSerialBench e o JGFSORBench, principalmente nos algoritmos de coleta paralela na geração estável, onde foram obtidos os piores resultados. Isso pode ser feito definindo um valor alvo alto para *throughput* através do uso da opção `-XX:GCTimeRation=<n>`. A definição de um *throughput* alto para ser atingido faz com que o coletor aumente o tamanho da *heap*, já que *heaps* maiores tendem a demorar mais para ficarem cheias.

Para os coletores paralelos, pode-se também modificar o número de *threads* utilizadas nos processos de limpeza através da opção `-XX:ParallelGC`*`Threads`*`=<n>`. Essa modificação pode ser útil para os aplicativos seqüenciais analisados, onde os núcleos de processamentos disponíveis na máquina são sub-utilizados e podem ser designados para os processos de coleta de lixo. Esses resultados devem ser ainda mais visíveis nos coletores CMS, já que na maioria das aplicações analisadas eles obtiveram bons resultados em termos de pausa mas ainda continuam com tempo total de execução superior aos valores médios. Uma mudança com essas características faria com que nenhuma *thread* tivesse sua execução interrompida, melhorando o tempo total da aplicação.

Capítulo 5

Conclusões

Neste trabalho apresentamos os mecanismos de gerenciamento automático de memória e mostramos como eles podem trazer benefícios para os desenvolvedores, eliminando erros comuns em linguagens onde esse controle é feito manualmente. Em contrapartida deste benefício, os coletores de lixo introduzem alguns custos adicionais para as aplicações em execução, que podem variar em relação ao uso de memória e o tempo de aplicação.

Apresentamos ainda neste trabalho uma avaliação de desempenho dos coletores disponíveis na Máquina Virtual Java. Acreditamos que os resultados possam auxiliar o usuário na escolha do coletor mais adequado para sua aplicação, levando-se em consideração as características da mesma e da máquina onde a aplicação será executada. Alguns quesitos de escolha foram abordados, como a seleção entre coletores seriais versus paralelos; coletores que realizam compactação versus coletores sem compactação versus coletores com cópia e coletores concorrentes versus coletores dedicados (stop-the-world). Para auxiliar nestas decisões, foram analisados indicadores como o throughput das aplicações, os tempos de pausas (valores médios e extremos) e tempos de execução; além da distribuição da memória da heap entre as gerações da JVM.

Com a análise dos resultados, foi possível identificar algumas configurações nos coletores que não obtiveram êxito para os testes realizados, como coletores paralelos na geração estável e coletores concorrentes incrementais. Os coletores seriais e paralelos na geração jovem foram os que obtiveram os melhores resultados no geral.

As opções padrão da JVM podem ser uma boa saída para usuários com pouca experiência em mecanismos de coleta de lixo. Caso o usuário deseje melhorar ainda mais o desempenho, é preciso que ele saiba modelar o padrão de alocação de memória, identificar quais as características que ele espera como resultado e comparar com as análises realizadas neste trabalho, fazendo do mesmo um guia inicial de modificações.

Para o estudo realizado até então, algumas sugestões de modificações nas opções da JVM foram propostas, com a finalidade de melhorar o desempenho para as aplicações com características semelhantes às aplicações estudadas. As avaliações destas modificações, bem como o teste das aplicações em máquinas com um único processador são exemplos de possíveis trabalhos futuros nesta área. É ainda importante avaliar, para este trabalho, qual o *overhead* introduzido pelos mcanismos de *log* da JVM.

Referências Bibliográficas

BACON, David; CHENG, Perry; RAJAN, V. **A Unified Theory of Garbage Collection.** In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, 19, 2004, New York, USA. PROC… New York: ACM Press, 2004. p. 50-68.

BAKER , Henry G. **The Treadmill**: Real-time garbage collection without motion sickness. In: ACM SIGPLAN NOTICES, New York, v.27, n.3, p. 66-70, Março 1992.

BARABASH, K. *et al*. **A parallel, incremental, mostly concurrent garbage collector for servers**. ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS (TOPLAS), ACM Press, New York, NY, USA, v. 27, n. 6, p. 1097–1146, 2005.

BARROS, Alexandra**. Uma Análise da Evolução da Coleta de Lixo Distribuída.** Rio de Janeiro, [2006?]. 39p. Monografia (Seminários de sistemas distribuídos) – Curso de Ciência da Computação, PUC-Rio.

BOEHM, H.-J.; DEMERS, A. J.; SHENKER, S. **Mostly parallel garbage collection**. In: CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION. [s.n.], 1991. v. 26, n. 6, p. 157–164.

CAUDILL, Patrick J; WIRFS-BROCK, Allen. **A third-generation Smalltalk-80 implementation**. In: CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 11, 1986, Portland, USA. PROC… New York: ACM Press, 1986. p. 119-130.

DETLEFS, D. **GCOld: a benchmark to stress old-generation collection.** 2003. Disponível em <http://192.18.108.226/ECom/EComTicketServlet/BEGIN2AA579EA8800A237BE9DF8AAB2DDA84E/-2147483648/2469782379/1/547718/547706/2469782379/2ts+/westCoastFSEND/ES-Gcold-1.0-G-F/ES-Gcold-1.0-G-F:1/GCold-1.0.tar.gz>. Acesso em: 02 set. 2007.

DEUTSCH, L. Peter; BOBROW, Daniel G. **An efficient, incremental, automatic garbage collector**. In: COMMUNICATIONS OF THE ACM, New York, v.19, n.9, p. 522-526, Setembro 1976.

EPCC. **The Java Grande Forum Benchmark Suite**. 2007. Disponível em <http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html>. Acessado em: 10 out. 2007.

GCVIEWER. 2006. Disponível em <http://www.tagtraum.com/gcviewer.html>. Acesso em: 1º Nov. 2007.

GENOME BioInformatics Research Lab. 2002. Disponível em <http://www1.imim.es/software/geneid/>. Acessado em 13 nov. 2007.

GOETZ, Brian. **A brief history of garbage collection.** Disponível em: <http://www.ibm.com/developerworks/java/library/j-jtp10283/>. Acesso em: 31 out 2003.

HERTZ, Matthew; FENG, Yi; BERGER, Emery. **Garbage Collection without paging.** In: CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 18, 2005, New York, USA. PROC… New York: ACM Press, 2005. p. 143-153.

LERMEN , Claus-Werner; MAURER, Dieter. **A protocol for distributed reference counting.** In: ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, 1986, Cambridge, USA. PROC… New York: ACM Press, 1986. p. 343-350.

LIEBERMAN, Henry; HEWITT, Carl. **A real-time garbage collector based on the lifetimes of objects**. **In:** COMMUNICATIONS OF THE ACM, New York, v.26, n.6, p. 419-429, Junho 1983.

LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. 2.ed. [s.l.]: Addison-Wesley, 1999. Disponível em: <http://java.sun.com/docs/books/vmspec/download/vmspec.2nded.html.tar.gz>. Acesso em: 20 out 2007.

MASAMITSU, J. **What the Heck's a Concurrent Mode?** 2006a. Disponível em <http://blogs.sun.com/jonthecollector/date/20060413>. Acesso em: 16 ago. 2007.

MASAMITSU, J. **What Were We Thinking?** 2005b. Disponível em <http://blogs.sun.com/jonthecollector/date/20051024>. Acesso em: 16 ago. 2007.

MASAMITSU, J. **Where Are the Sharp Edges?** 2005a. Disponível em <http://blogs.sun.com/jonthecollector/date/20051005>. Acesso em: 16 ago. 2007.

MASAMITSU, J. **Why now?** 2006b. Disponível em <http://blogs.sun.com/jonthecollector/date/20061026>. Acesso em: 16 ago. 2007.

MCBETH , J. Harold. **On the reference counter method**. In: COMMUNICATIONS OF THE ACM, 6, 1963.

MCBETH, H. J. **Letters to the editor**: on the reference counter method. **In:** COMMUNICATIONS OF THE ACM, New York, v.6, n.8, Setembro 1963.

PAWLAN, M. **Reference Objects and Garbage Collection**, Sun Microsystems, JDC, 1998. Disponível em <http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj/>. Acesso em: 10 out. 2007.

PLAINFOSSÉ, David; SHAPIRO, Marc. **A survey of distributed garbage collection techiniques**. In: INTERNATIONAL WORKSHOP ON MEMORY MANAGEMENT, 1992, Kinross, Escócia. PROC… Londres: Springer-Verlag, 1995. P. 211-249.

PRINTEZIS, T. **Garbage Collection in the Java HotSpot Virtual Machine.** 2004. Disponível em <http://www.devx.com/Java/Article/21977>. Acesso em: 10 set. 2007.

RAMAKRISHNA, Y. Srinivas. **Automatic memory management in the Java HotSpot virtual machine**. In JAVAONE CONFERENCE, 2006. Disponível em: <http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf>. Acesso em: 14 set 2007.

ROCHA, Helder. **Gerência de memória em Java.** 2005. Disponível em <http://www.argonavis.com.br/cursos/java/j100/java-5-mem-2.pdf>. Acesso em: 29 set. 2007.

SUN Microsystems. Diagnosing a Garbage Collection problem. 2003. Disponível em <http://java.sun.com/docs/hotspot/gc1.4.2/example.html>. Acesso em: 21 nov. 2007.

SUN Microsystems. Tuning java se 6 hotspot virtual machine garbage collection. Sun Microsystems, 2006. Disponível em: <http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html>. Acesso em: 17 set 2007.

UNGAR, David M. **Generation scavenging**: A non-disruptive high-performance storage reclamation algorithm. In: ACM SIGSOFT/SIGPLAN SOFTWARE ENGINEERING SYMPOSIUM ON PRACTICAL SOFTWARE DEVELOPMENT ENVIRONMENTS, 1, 1984, New York, USA. PROC… New York: ACM Press, 1994. p.157-167.

VAUGHAN, F. A; BRODIE-TYRRELL, W; FALKNER, K; MUNRO, D. **Bounded parallel garbage collection**: implementation and adaptation. In: AUSTRALIAN PARALLEL AND REAL TIME CONFERENCE, 7, 2000, Sydney, Australia. PROC… Sydney: [s.n.], 2000.

WIKIPEDIA. **Java (linguagem de programação).** 2007. Disponível em: <http://pt.wikipedia.org/wiki/Java_(linguagem_de_programa%C3%A7%C3%A3o)>. Acesso em: 21 out. 2007

WILSON, Paul R; JOHNSTONE, Mark S. **Real-Time Non-Copying Garbage Collection**. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, 8, 1993, Washington, USA. PROC… Washington: ACM Press, 1993.

WILSON, Paul. **Uniprocessor garbage collection techniques.** In: INTERNATIONAL WORKSHOP ON MEMORY MANAGEMENT, 1992, Saint-Malo, France. PROC… Saint-Malo: Springer-Verlag, 1992. N. 637

WILSON, Paul; LAM, Michael; MOHER, Thomas. **Caching considerations for general garbage collections.** In: ACM CONF. ON LISP AND FUNCTIONAL PROGRAMMING, 1992, San Francisco, USA. PROC… [s.l.]: [s.n.], 1992. p. 32-42.

WITHINGTON, P. T. **How Real is "Real Time" Garbage Collection?**. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, 6, 1991, Phoenix, USA. PROC… Phoenix: AMC SIGPLAN, 1991.