Relatório de Caracterização da Localidade de Referência Utilizando Memlog e GProf Para o Problema da Torre de Hanói

Bruno Oliveira Souza Santos

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

brunooss@ufmg.br

1 Introdução

O objetivo da prática é apresentar a caracterização da localidade de referência no contexto do problema da Torre de Hanói, por meio da análise do padrão de acesso à memória durante a execução do algoritmo.

1.1 Descrição do Algoritmo

O problema da Torre de Hanói é um desafio matemático e de lógica que consiste em mover uma pilha de discos de tamanhos diferentes de um pino de origem para um pino de destino, utilizando um pino intermediário, com a restrição de que um disco maior nunca pode ser colocado sobre um disco menor.

A solução ótima para o problema envolve um procedimento recursivo, onde cada movimento é realizado em etapas menores, movendo uma subpilha de discos para o pino intermediário e, em seguida, movendo o disco restante para o pino de destino. A complexidade do problema cresce exponencialmente com o número de discos, tornando-o um desafio interessante para análise de algoritmos e avaliação do comportamento de memória.

2 Análise de Complexidade

2.1 Complexidade de Espaço

A complexidade de espaço do algoritmo da Torre de Hanói é linear em relação ao número de discos. Isso ocorre porque o espaço ocupado pelo programa depende diretamente da quantidade de discos que estão sendo movidos. O algoritmo utiliza uma estrutura de dados de pilha para armazenar os discos durante a execução, e o número máximo de discos que podem estar empilhados em um determinado momento é igual ao número total de discos envolvidos no problema. Portanto, a complexidade de espaço é representada por $O(n)$, onde n é o número de discos.

2.2 Complexidade de Tempo

A complexidade de tempo do algoritmo da Torre de Hanói é exponencial, especificamente de ordem $O(2^n)$, em que n é o número de discos a serem considerados na abordagem do problema. Dessa forma, o tempo de execução do algoritmo aumenta de forma exponencial com o número de discos envolvidos. Tal complexidade se dá pela natureza recursiva do algoritmo que, em cada chamada recursiva, divide o problema em subproblemas menores, até que sejam atingidos casos base, em que a solução é trivial. Portanto, o número total de movimentos necessários para resolver o problema da Torre de Hanói é $2^n - 1$, o que leva a uma complexidade exponencial da ordem de $O(2^n)$.

3 Metodologia

3.1 Seleção das Estruturas de Dados

As estruturas de dados a serem monitoradas durante a execução do programa são três classes "Torre", que se baseiam, essencialmente, em três estruturas de pilha, sendo estas as pilhas original, auxiliar e destino. Estas estruturas se baseiam em nós, cada um com seu respectivo valor e um ponteiro para o próximo, o que possui potencial para afetar diretamente a localidade de referência do programa em questão.

3.2 Seleção de Funções Instrumentadas

Foram selecionadas para a instrumentação as funções de criação (inicialização) da torre, copulando a estrutura com n discos, de resolução do problema, a partir da movimentação dos discos entre as três torres supracitadas. Essas funções são responsáveis por manipular as pilhas e têm influência na análise em questão.

3.3 Definição de Fases de Monitorização

A partir da instrumentação das funções acima, dividiu-se a monitoração em três fases: a fase de inicialização das torres a serem utilizadas, a fase de resolução do problema, por meio da manipulação dos discos entre as torres, e a fase de impressão dos elementos da estrutura de torre, após a sua ordenação.

3.4 Instrumentação

A instrumentação se deu utilizando ferramentas de monitorização de acesso à memória, como a biblioteca *memlog.h* e a ferramenta *valgrind*. A instrumentação foi aplicada, respectivamente, para a função de criação das estruturas de torre e para a função de ordenação destas, gerando torres copuladas com dados aleatórios para cada execução.

4 Experimentação

A fim de se observar o comportamento da localidade de referência do programa, foram definidos experimentos com $n = 16 discos$, para ambas as funções a serem monitoradas. Com isso, observou-se a duração da execução destas e as informações de leitura e escrita fornecidas pela biblioteca memlog.

Figura 1: Saída do comando de execução da ferramenta cachegrind

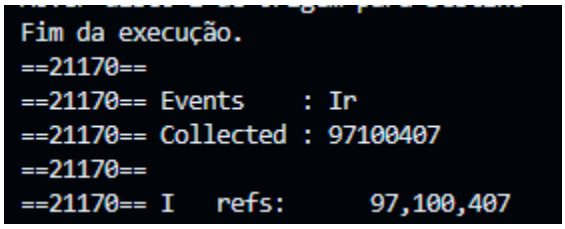


Figura 2: Saída do comando de execução da ferramenta cachegrind

5 Resultados e Análises

A partir do código desenvolvido em C++ para implementar o problema em questão, selecionou-se um valor de $n = 16$ (ou seja, 16 discos a serem ordenados pelo algoritmo) e, posteriormente, foram executados os seguintes comandos:

- valgrind –tool=cachegrind ./bin/main.out
- valgrind –tool=callgrind ./bin/main.out

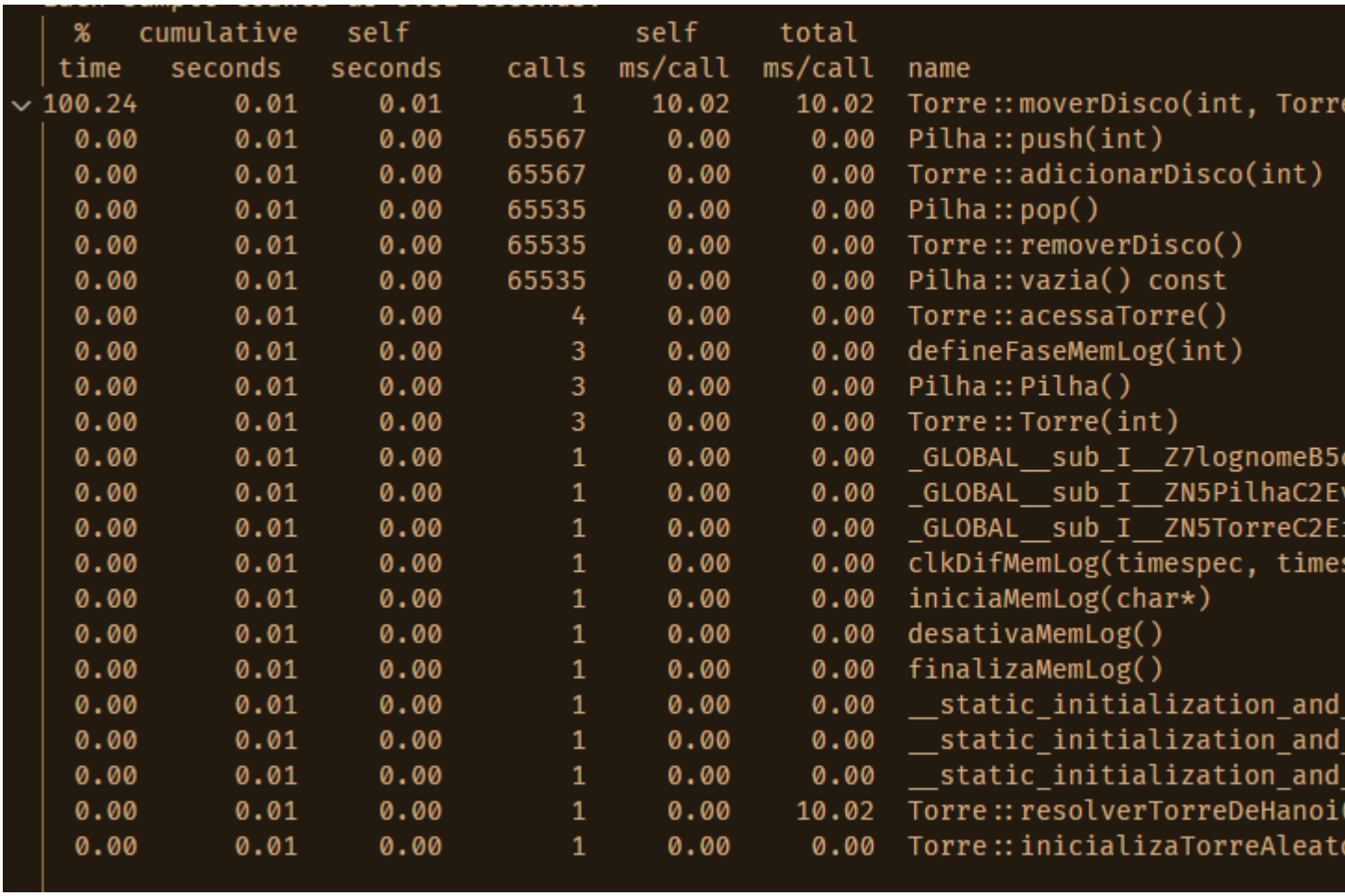
A partir disso, foram obtidos, respectivamente, os resultados a seguir:

Os resultados obtidos a partir das análises realizadas com as ferramentas callgrind e cachegrind forneceram informações a respeito da localidade de referência do programa em estudo, que serão analisadas a seguir.

Ao avaliar o *cache* de instruções, observamos que o número total de referências de instrução coletadas pelo callgrind foi de 97,100,407.

Em relação ao *cache* de dados, os dados coletados pelo cachegrind revelaram que o programa realizou um total de 41.183.614 referências de dados, sendo 26.964.572 leituras e 14.219.042 escritas. A taxa de *miss* do *cache* de nível 1 de dados foi registrada como 0.0%, com 15,602 *misses*, enquanto o *cache* de último nível de dados apresentou uma taxa de *miss* de 0.0%, com 9,887 misses. *Esses*

resultados indicam que o programa possui uma alta localidade de referência em relação ao *cache* de dados, uma vez que a grande maioria dos acessos aos dados é atendida pelo *cache*, resultando em um número consideravelmente baixo de *misses*.



% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.24	0.01	0.01	1	10.02	10.02	Torre::moverDisco(int, Torre
0.00	0.01	0.00	65567	0.00	0.00	Pilha::push(int)
0.00	0.01	0.00	65567	0.00	0.00	Torre::adicionarDisco(int)
0.00	0.01	0.00	65535	0.00	0.00	Pilha::pop()
0.00	0.01	0.00	65535	0.00	0.00	Torre::removerDisco()
0.00	0.01	0.00	65535	0.00	0.00	Pilha::vazia() const
0.00	0.01	0.00	4	0.00	0.00	Torre::acessaTorre()
0.00	0.01	0.00	3	0.00	0.00	defineFaseMemLog(int)
0.00	0.01	0.00	3	0.00	0.00	Pilha::Pilha()
0.00	0.01	0.00	3	0.00	0.00	Torre::Torre(int)
0.00	0.01	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z7lognomeB5
0.00	0.01	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5PilhaC2E
0.00	0.01	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5TorreC2E
0.00	0.01	0.00	1	0.00	0.00	clkDifMemLog(timespec, times
0.00	0.01	0.00	1	0.00	0.00	iniciaMemLog(char*)
0.00	0.01	0.00	1	0.00	0.00	desativaMemLog()
0.00	0.01	0.00	1	0.00	0.00	finalizaMemLog()
0.00	0.01	0.00	1	0.00	0.00	__static_initialization_and
0.00	0.01	0.00	1	0.00	0.00	__static_initialization_and
0.00	0.01	0.00	1	0.00	0.00	__static_initialization_and
0.00	0.01	0.00	1	0.00	10.02	Torre::resolverTorreDeHanoi
0.00	0.01	0.00	1	0.00	0.00	Torre::inicializaTorreAleat

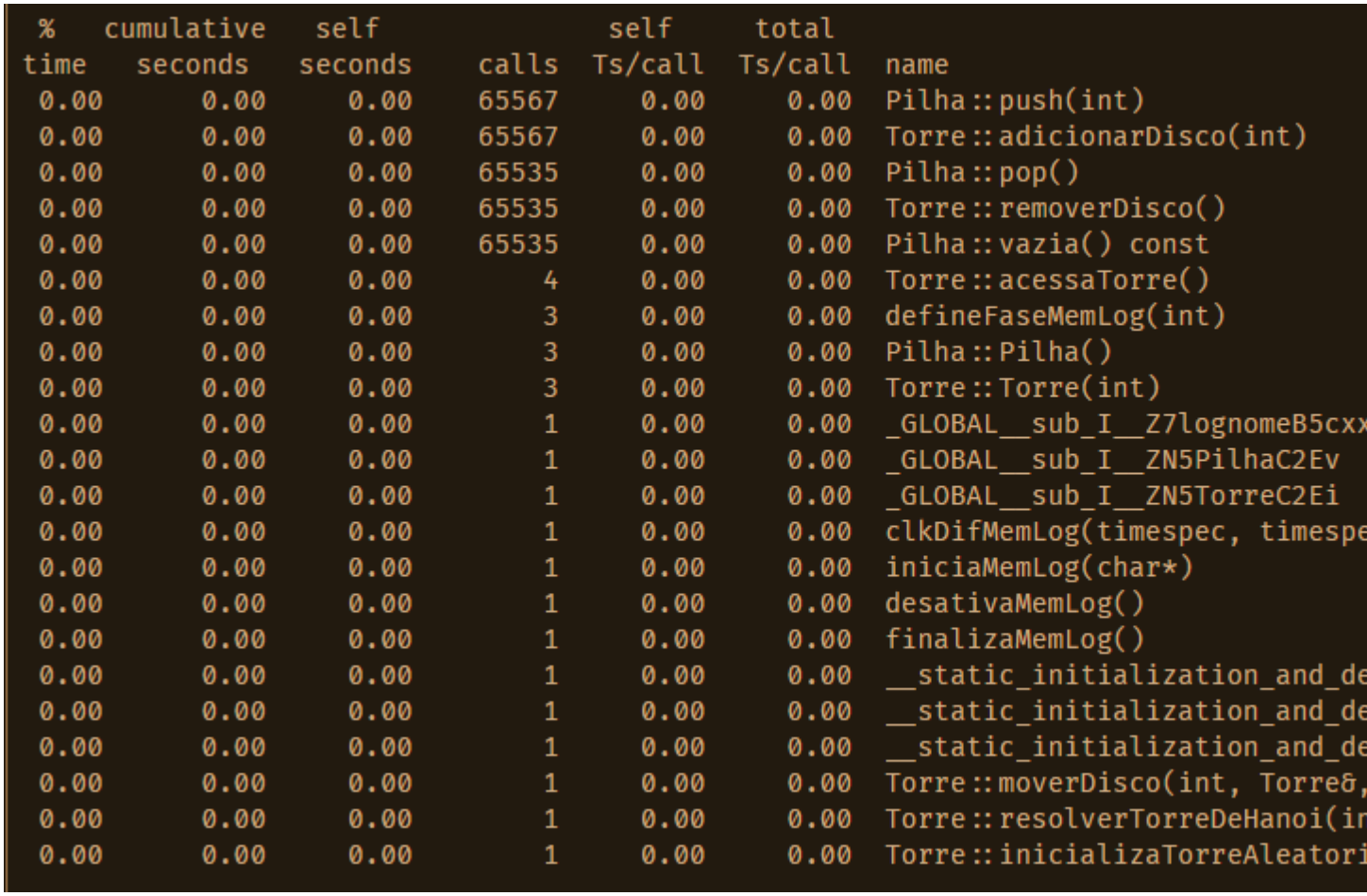
Figura 3: Saída do comando de execução da ferramenta gprof para a função de inicialização das estruturas de Torre

Além disso, executou-se o programa $gProf$, a fim de analisar o tempo de execução de cada função, cujos resultados podem ser observados por meio das imagens a seguir:

Em relação à saída do gprof, foi possível observar que a função de movimentação de disco consumiu majoritariamente o tempo total de execução durante o monitoramento da funcionalidade de ordenação. Sendo assim, esta função foi a mais chamada percentualmente em relação às demais.

6 Conclusão

Com base na análise realizada anteriormente, é possível concluir que, devido à natureza da função de movimentação de discos durante a sua ordenação, que se baseia na escrita dos discos entre as estruturas de torre, esta contribui para uma alta localidade de referência do programa.



% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	65567	0.00	0.00	Pilha::push(int)
0.00	0.00	0.00	65567	0.00	0.00	Torre::adicionarDisco(int)
0.00	0.00	0.00	65535	0.00	0.00	Pilha::pop()
0.00	0.00	0.00	65535	0.00	0.00	Torre::removerDisco()
0.00	0.00	0.00	65535	0.00	0.00	Pilha::vazia() const
0.00	0.00	0.00	4	0.00	0.00	Torre::acessaTorre()
0.00	0.00	0.00	3	0.00	0.00	defineFaseMemLog(int)
0.00	0.00	0.00	3	0.00	0.00	Pilha::Pilha()
0.00	0.00	0.00	3	0.00	0.00	Torre::Torre(int)
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z7lognomeB5cx
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5PilhaC2Ev
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__ZN5TorreC2Ei
0.00	0.00	0.00	1	0.00	0.00	clkDifMemLog(timespec, timespe
0.00	0.00	0.00	1	0.00	0.00	iniciaMemLog(char*)
0.00	0.00	0.00	1	0.00	0.00	desativaMemLog()
0.00	0.00	0.00	1	0.00	0.00	finalizaMemLog()
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_de
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_de
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_de
0.00	0.00	0.00	1	0.00	0.00	Torre::moverDisco(int, Torre&,
0.00	0.00	0.00	1	0.00	0.00	Torre::resolverTorreDeHanoi(in
0.00	0.00	0.00	1	0.00	0.00	Torre::inicializaTorreAleatori

Figura 4: Saída do comando de execução da ferramenta gprof para a função de ordenação das estruturas de Torre