Trabalho Prático 1:

Escalonador de URLs

Erick Sunclair Santos Batista - 2020026877

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

ericksunclair@ufmg.br

1. Introdução

Um dos principais componentes de uma máquina-de-busca é o coletor, responsável por varrer a internet realizando o download do conteúdo apontado pelo endereço de uma página (sua URL). O coletor necessita de um escalonador para definir a ordem em que as páginas apontadas pelas URLs serão coletadas. A ordem depende da estratégia de coleta adotada e as duas mais conhecidas são: depth-first (busca em profundidade), que coleta todas as URLs de um host antes de passar para o próximo; e breadth-first (busca em largura), que prioriza a variedade e coleta URLs de diferentes hosts simultaneamente, coletando uma URL de cada sítio.

Neste trabalho foi desenvolvido, na linguagem C++, um escalonador de URLs que usa a estratégia depth-first. Essa aplicação consegue armazenar URLs, fazer operações sobre essas URLs e imprimir as URLs como desejado pelo usuário, levando em conta conceitos como desempenho, robustez e abstração. O desenvolvimento desse sistema utilizou uma lista encadeada como Tipo Abstrato de Dados (TAD) para armazenamento das URLs, então este trabalho também trata de analisar o desempenho e os custos temporal e espacial da aplicação desenvolvida e de cada uma de suas operações individualmente.

A seção 2 aborda as características do código implementado e as configurações usadas para testar o programa. A seção 3 explica como acessar e executar o programa. A seção 4 apresenta uma análise de complexidade de tempo e espaço do programa. Por fim, a seção 5 possui a conclusão do trabalho desenvolvido.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilado pelo compilador G++ da GNU Compiler Collection e os testes foram realizados em um computador com Windows 10, intel core i5 3.4 GHz e 8GB de memória RAM.

2.1. A Lista Encadeada

O programa teve como base uma lista encadeada de vários itens, em que cada item representa uma URL. Uma lista encadeada é uma estrutura de dados composta de várias células e cada célula possui um ponteiro para a próxima célula da lista, de forma que todas as células estão interligadas através de ponteiros. Dessa forma a lista pode ter seu tamanho definido em tempo de execução (a alocação é dinâmica), então o tamanho da lista é variável e não custa espaço extra desnecessário ao computador, o que seria o caso se fosse utilizada

um vetor/arranjo como lista, ou seja, se a alocação fosse estática. Dessa forma o desempenho do programa é aprimorado. As desvantagens desse tipo de implementação são:

- Maior complexidade de implementação
- Se a lista for muito grande ela pode acabar exigindo muita memória do computador, já que não existe limite para o tamanho dessa lista (no entanto, esse limite pode ser implementado também para esse tipo de execução caso seja necessário).

No arquivo "item_url" foi criada uma classe para o tipo base de uma célula que vai representar uma URL armazenada na nossa lista. Esse tipo base possui uma string com a própria URL, um ponteiro para a próxima célula da lista e uma outra string com o host da URL dessa célula. O host é o sítio da URL sem o "http://" e o "www". A Imagem 2.1 abaixo ilustra como é a estrutura de uma URL em geral. O construtor desse tipo base apenas inicializa o host e o URL e o ponteiro para a próxima célula.

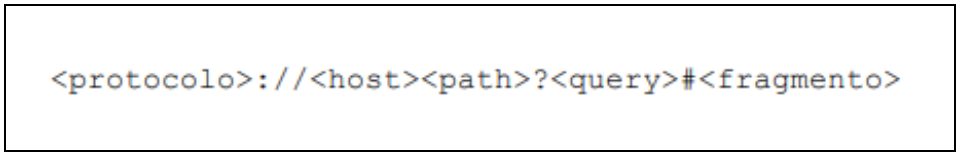


Imagem 2.1 – Composição de uma URL

Já o arquivo "lista_urls" possui uma classe que consiste na lista encadeada em si e seus métodos para trabalhar com as URLs. A classe lista_urls possui um ponteiro para o primeiro elemento da lista (ou a cabeça), um ponteiro para o último elemento da lista e um inteiro com o tamanho atual da lista. Dentre os métodos dessa classe estão um construtor, que inicializa as variáveis, um destrutor que desaloca todos os itens alocados até então.

Existem quatro métodos internos ao próprio programa que são responsáveis por testar as informações recebidas e formata-las corretamente:

- O método condiciona_url retira, se existir, o "www.", a "/" ao fim do endereço e todo o fragmento, a partir do "#" (vide Imagem 2.1) de uma URL.
- O método condiciona_host faz o mesmo que o método anterior além de também retirar o protocolo ("http://") para deixar um host formatado.
- O método get_host recebe uma URL e retorna apenas o host dessa URL.
- O método checar_consistencia recebe uma URL e retorna um booleano 1 se a URL atende aos requisitos para ser armazenada em nossa lista. Será retornado 0 se: o protocolo não for "http" ou a página tiver extensão diferente de ".html" (".jpg", ".gif", ".mp3", ".avi", ".doc" ou ".pdf").

Também existem os métodos que os usuários podem acessar e que definem as operações que eles querem fazer sobre a lista:

- O método add_urls lê uma quantidade especificada de URLs do arquivo de entrada (esse arquivo vai ter todos os comandos do usuário) e adiciona essas URLs à lista (se elas atenderem aos requisitos).
- O método escalona_tudo vai escalonar todos os URLs da lista ordenados por host e em cada host por ordem de profundidade (sendo escalonados primeiro os URLs de menor profundidade, que são os que possuem menos barras, sem contar a barra ao final do endereço, que deve ser retirada). Escalonando esses URLs o programa vai escrevê-los no arquivo de saída e apagá-los da lista.
- O método escalona vai escalonar a quantidade especificada de URLs, na mesma ordem definida anteriormente.

- O método escalona_host também vai escalonar certa quantidade de URLs mas apenas de um host.
- O método ver_host vai imprimir no arquivo de saída todos os URLs de um host especificado.
- O método lista_hosts vai listar no arquivo de saída todos os hosts que existem atualmente na lista.
- O método limpa_host vai apagar todas as URLs de um host.
- Por fim, o método limpa_tudo vai apagar todas as URLs e hosts da lista.

3. Instruções de Compilação e Execução

Para executar o programa recomenda-se usar o Makefile disponibilizado na matriz do diretório do programa pois ele já compila, linka e executa um programa para o arquivo de entrada. Por padrão, o arquivo que será lido e terá seus comandos executado pelo programa deve ser um ".txt" e estar também na matriz do diretório do programa. Além disso ele deve ser o único ".txt" na matriz do programa. Para executar deve-se seguir os seguintes passo:

- Utilizando um terminal acesse o diretório "/Erick_Sunclair_2020026877/TP".
- Execute o arquivo Makefile utilizando o seguinte comando: "make all".
- Será criado no diretório "/result" um arquivo de saída com o nome do arquivo de entrada mais a adição de um "-out" ao fim. Esse arquivo vai conter os resultados as operações do arquivo de entrada.
- Para apagar os objetos criados na compilação do programa basta executar no terminal outro comando do arquivo Makefile: "make clean".
- E podemos executar novamente o programa.

4. Análise de Complexidade

4.1. Complexidade Temporal

Com relação à complexidade temporal podemos citar como funções de complexidade O(1) o construtor da classe item_url (que representa uma célula da lista), o construtor da classe lista_urls, e as funções condiciona_url, condiciona_host, get_host e checar_consistencia pois todas essas funções fazem apenas operações de complexidade O(1). Já com complexidade O(tam), onde tam é o número atual de membros da lista encadeada, podemos listar o método limpa_tudo, o destrutor da lista_urls (pois ele chama a função limpa_tudo) e escalona_tudo pois elas percorrem toda a lista para desempenhar suas funções. O método escalona também possui complexidade O(tam) apesar de que ele pode percorrer um número de elementos menor que o tamanho da lista.

Devido à implementação do programa, os métodos que deveriam percorrer apenas um host específico (escalona_host, ver_host) ou apenas a lista de hosts (lista_hosts) nesse programa precisa percorrer toda a lista encadeada de URLs (procurando pelas URLs com o host necessário), e por esse motivo as funções escalona_host, ver_host e lista_hosts também são de ordem de complexidade temporal O(tam). Caso fosse feita uma implementação baseada em uma lista encadeada de hosts em que cada host possui uma lista encadeada de URLs o programa teria melhor eficiência temporal já que escalona_host e ver_host teriam complexidade O(tam_host), onde tam_host e o tamanho do host que querermos percorrer, e lista_hosts teria complexidade O(num_hosts), onde num_hosts e o número de hosts salvos.

O método add_urls possui um loop que será executado quant vezes, em que quant é o número de URLs a serem adicionadas, e dentro desse loop existem mais dois loops que percorrem a lista toda (loop executa tam

vezes) e um loop que percorre todos os caracteres da url a ser adicionada (loop executa tam_url vezes). Com isso, a complexidade temporal dessa função é O(quant x (2 x tam + tam_url)).

4.2. Complexidade Espacial

Analisando agora a complexidade espacial temos:

- Construtor do item_url: O(1) pois só cria variáveis simples (as variáveis da classe).
- Construtor da lista_urls: O(1) pois só cria variáveis simples (as variáveis da classe).
- Destrutor da lista_urls: O(1), só cria variáveis auxiliares simples.
- condiciona_url: O(1), só cria variáveis auxiliares simples.
- condiciona_host: O(1), só cria variáveis auxiliares simples.
- get_host: O(1), só cria variáveis auxiliares simples.
- checar_consistencia: O(1), só cria variáveis auxiliares simples.
- add_urls: O(quant), sendo quant a quantidade de itens a serem adicionados na lista, pois essa função aloca o espaço para no máximo quant novas URLs (pode ser que algumas URLs não atendam aos requisitos e não entrem na lista), ou seja, até quant novos item_url são alocados.
- escalona_tudo: O(1), só cria variáveis auxiliares simples.
- escalona: O(1), só cria variáveis auxiliares simples.
- escalona_host: O(1), só cria variáveis auxiliares simples.
- ver_host: O(1), só cria variáveis auxiliares simples.
- lista_hosts: O(1), só cria variáveis auxiliares simples.
- limpa_host: O(1), só cria variáveis auxiliares simples.
- limpa_tudo: O(1), só cria variáveis auxiliares simples.

5. Análise dos Resultados

Foram feitos testes de carga para o programa no computador citado anteriormente para avaliar o desempenho do programa e seu acesso à memória em diferentes casos. Para isso foi usado um grupo padrão de comando que adiciona certa quantidade de URLs, lista os hosts, escalona tudo e limpa tudo. Em seguida foi usado o GNU Profiler (gprof) para relatar o número de chamadas e o tempo de uso de cada função do programa especificamente.

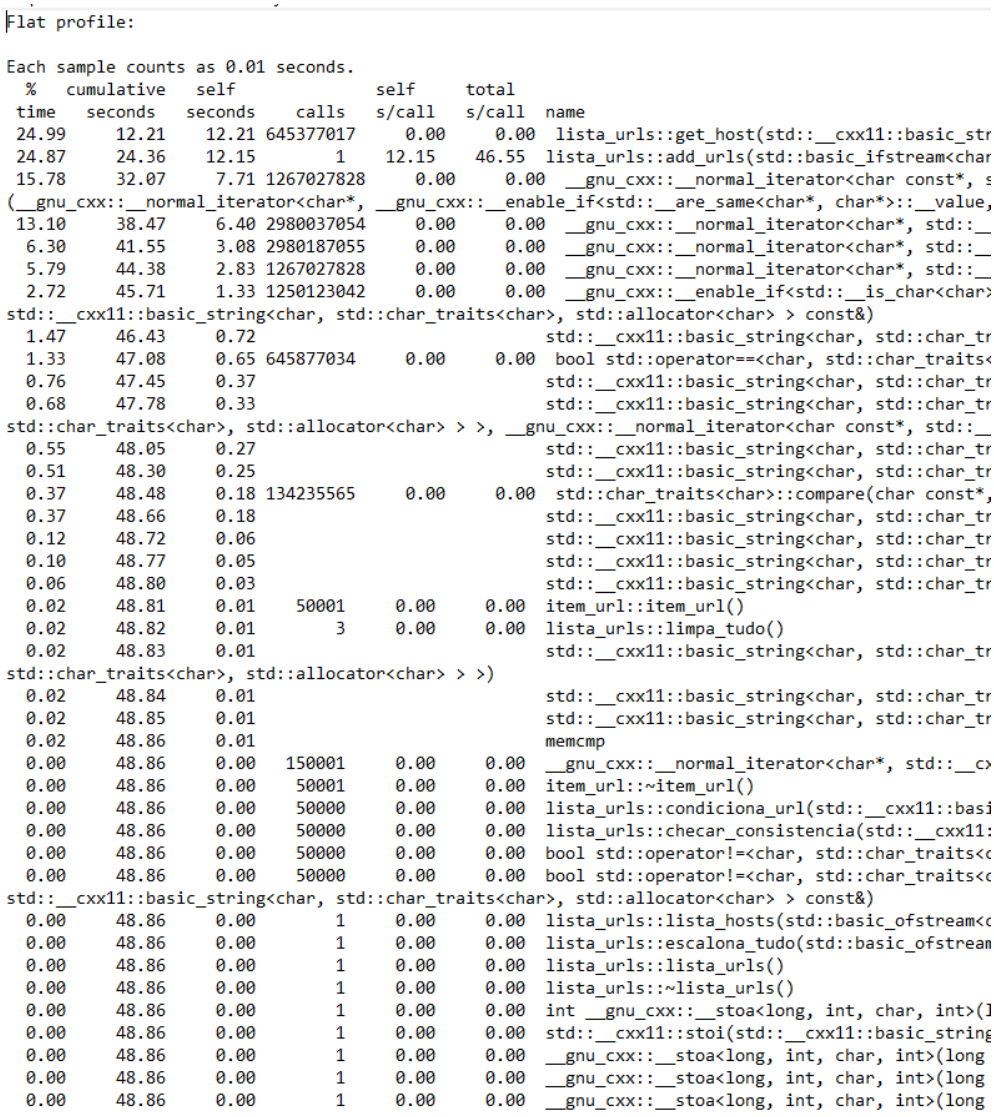
Na Figura 5.1 podemos ver o resultado para o caso em que foram adicionados e escalonados 26 URLs. Na figura 5.2 o mesmo para 506 URLs e na Figura 5.3 para 5000 URLs. A primeira informação que podemos tirar desses perfis é que o tempo gasto pelo programa é quase imperceptível para poucas URLs, sendo que para 26 URLs nem temos tempo nenhum registrado, apesar de todas as chamadas de funções. Em 506 URLs esse valor temporal começa a aparecer (apesar de ainda ser pequeno) mas em 5000 URLs podemos ter uma noção verdadeira das funções que gastam mais tempo de operação.

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00     3199     0.00     0.00  __gnu_cxx::__normal_iterator<char co
(__gnu_cxx::__normal_iterator<char*, __gnu_cxx::__enable_if<std::__are_same<char*, char*>:
  0.00      0.00     0.00     3199     0.00     0.00  __gnu_cxx::__normal_iterator<char*,
  0.00      0.00     0.00     1758     0.00     0.00  __gnu_cxx::__normal_iterator<char*,
  0.00      0.00     0.00     1679     0.00     0.00  __gnu_cxx::__normal_iterator<char*,
  0.00      0.00     0.00      598     0.00     0.00  bool std::operator==<char, std::char
  0.00      0.00     0.00      374     0.00     0.00  __gnu_cxx::__enable_if<std::__is_cha
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.00     0.00      321     0.00     0.00  lista_urls::get_host(std::__cxx11::b
  0.00      0.00     0.00       79     0.00     0.00  __gnu_cxx::__normal_iterator<char*,
  0.00      0.00     0.00       64     0.00     0.00  std::char_traits<char>::compare(char
  0.00      0.00     0.00       27     0.00     0.00  item_url::item_url()
  0.00      0.00     0.00       27     0.00     0.00  item_url::~item_url()
  0.00      0.00     0.00       26     0.00     0.00  lista_urls::condiciona_url(std::__cx
  0.00      0.00     0.00       26     0.00     0.00  lista_urls::checar_consistencia(std:
  0.00      0.00     0.00       26     0.00     0.00  bool std::operator!=<char, std::char
  0.00      0.00     0.00       26     0.00     0.00  bool std::operator!=<char, std::char
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.00     0.00        3     0.00     0.00  lista_urls::limpa_tudo()
  0.00      0.00     0.00        1     0.00     0.00  lista_urls::lista_hosts(std::basic_o
  0.00      0.00     0.00        1     0.00     0.00  lista_urls::escalona_tudo(std::basic
  0.00      0.00     0.00        1     0.00     0.00  lista_urls::add_urls(std::basic_ifst
  0.00      0.00     0.00        1     0.00     0.00  lista_urls::lista_urls()
  0.00      0.00     0.00        1     0.00     0.00  lista_urls::~lista_urls()
  0.00      0.00     0.00        1     0.00     0.00  int __gnu_cxx::__stoa<long, int, cha
  0.00      0.00     0.00        1     0.00     0.00  std::__cxx11::stoi(std::__cxx11::bas
  0.00      0.00     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, i
  0.00      0.00     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, i
  0.00      0.00     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, i
```

Imagem 5.1 – Resultado para 26 URLs

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.01     0.01   664659     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c
  0.00      0.01     0.00   664659     0.00     0.00  __gnu_cxx::__normal_iterator<char const*, st
(__gnu_cxx::__normal_iterator<char*, __gnu_cxx::__enable_if<std::__are_same<char*, char*>::__value
  0.00      0.01     0.00   355050     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c
  0.00      0.01     0.00   353531     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c
  0.00      0.01     0.00   128857     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>:
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.01     0.00    81720     0.00     0.00  bool std::operator==<char, std::char_traits<
  0.00      0.01     0.00    76643     0.00     0.00  lista_urls::get_host(std::__cxx11::basic_str
  0.00      0.01     0.00    15461     0.00     0.00  std::char_traits<char>::compare(char const*,
  0.00      0.01     0.00     1519     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c
  0.00      0.01     0.00      507     0.00     0.00  item_url::item_url()
  0.00      0.01     0.00      507     0.00     0.00  item_url::~item_url()
  0.00      0.01     0.00      506     0.00     0.00  lista_urls::condiciona_url(std::__cxx11::bas
  0.00      0.01     0.00      506     0.00     0.00  lista_urls::checar_consistencia(std::__cxx11
  0.00      0.01     0.00      506     0.00     0.00  bool std::operator!=<char, std::char_traits<
  0.00      0.01     0.00      506     0.00     0.00  bool std::operator!=<char, std::char_traits<
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.01     0.00        3     0.00     0.00  lista_urls::limpa_tudo()
  0.00      0.01     0.00        1     0.00     0.00  lista_urls::lista_hosts(std::basic_ofstream<
  0.00      0.01     0.00        1     0.00     0.00  lista_urls::escalona_tudo(std::basic_ofstrea
  0.00      0.01     0.00        1     0.00    10.00  lista_urls::add_urls(std::basic_ifstream<cha
  0.00      0.01     0.00        1     0.00     0.00  lista_urls::lista_urls()
  0.00      0.01     0.00        1     0.00     0.00  lista_urls::~lista_urls()
  0.00      0.01     0.00        1     0.00     0.00  int __gnu_cxx::__stoa<long, int, char, int>(
  0.00      0.01     0.00        1     0.00     0.00  std::__cxx11::stoi(std::__cxx11::basic_strin
  0.00      0.01     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long
  0.00      0.01     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long
  0.00      0.01     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long
```

Imagem 5.2 – Resultado para 506 URLs



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 24.99     12.21    12.21 645377017     0.00     0.00  lista_urls::get_host(std::__cxx11::basic_str
 24.87     24.36    12.15        1    12.15    46.55  lista_urls::add_urls(std::basic_ifstream<char
 15.78     32.07     7.71 1267027828     0.00     0.00  __gnu_cxx::__normal_iterator<char const*, s
(__gnu_cxx::__normal_iterator<char*, __gnu_cxx::__enable_if<std::__are_same<char*, char*>::__value,
 13.10     38.47     6.40 2980037054     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__
  6.30     41.55     3.08 2980187055     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__
  5.79     44.38     2.83 1267027828     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__
  2.72     45.71     1.33 1250123042     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  1.47     46.43     0.72                               std::__cxx11::basic_string<char, std::char_tr
  1.33     47.08     0.65 645877034     0.00     0.00  bool std::operator==<char, std::char_traits<
  0.76     47.45     0.37                               std::__cxx11::basic_string<char, std::char_tr
  0.68     47.78     0.33                               std::__cxx11::basic_string<char, std::char_tr
std::char_traits<char>, std::allocator<char> > >, __gnu_cxx::__normal_iterator<char const*, std::__
  0.55     48.05     0.27                               std::__cxx11::basic_string<char, std::char_tr
  0.51     48.30     0.25                               std::__cxx11::basic_string<char, std::char_tr
  0.37     48.48     0.18 134235565     0.00     0.00  std::char_traits<char>::compare(char const*,
  0.37     48.66     0.18                               std::__cxx11::basic_string<char, std::char_tr
  0.12     48.72     0.06                               std::__cxx11::basic_string<char, std::char_tr
  0.10     48.77     0.05                               std::__cxx11::basic_string<char, std::char_tr
  0.06     48.80     0.03                               std::__cxx11::basic_string<char, std::char_tr
  0.02     48.81     0.01    50001     0.00     0.00  item_url::item_url()
  0.02     48.82     0.01        3     0.00     0.00  lista_urls::limpa_tudo()
  0.02     48.83     0.01                               std::__cxx11::basic_string<char, std::char_tr
std::char_traits<char>, std::allocator<char> > >)
  0.02     48.84     0.01                               std::__cxx11::basic_string<char, std::char_tr
  0.02     48.85     0.01                               std::__cxx11::basic_string<char, std::char_tr
  0.02     48.86     0.01                               memcmp
  0.00     48.86     0.00   150001     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__c>
  0.00     48.86     0.00    50001     0.00     0.00  item_url::~item_url()
  0.00     48.86     0.00    50000     0.00     0.00  lista_urls::condiciona_url(std::__cxx11::basi
  0.00     48.86     0.00    50000     0.00     0.00  lista_urls::checar_consistencia(std::__cxx11:
  0.00     48.86     0.00    50000     0.00     0.00  bool std::operator!=<char, std::char_traits<c
  0.00     48.86     0.00    50000     0.00     0.00  bool std::operator!=<char, std::char_traits<c
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00     48.86     0.00        1     0.00     0.00  lista_urls::lista_hosts(std::basic_ofstream<c
  0.00     48.86     0.00        1     0.00     0.00  lista_urls::escalona_tudo(std::basic_ofstream
  0.00     48.86     0.00        1     0.00     0.00  lista_urls::lista_urls()
  0.00     48.86     0.00        1     0.00     0.00  lista_urls::~lista_urls()
  0.00     48.86     0.00        1     0.00     0.00  int __gnu_cxx::__stoa<long, int, char, int>(l
  0.00     48.86     0.00        1     0.00     0.00  std::__cxx11::stoi(std::__cxx11::basic_string
  0.00     48.86     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long
  0.00     48.86     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long
  0.00     48.86     0.00        1     0.00     0.00  __gnu_cxx::__stoa<long, int, char, int>(long
```

Imagem 5.3 – Resultado para 5000 URLs

5.1. Com Relação ao Tempo

Analisando o caso em que são adicionadas 5000 URLs notamos que as funções get_host e add_urls são as que mais custam tempo de operação, no caso da get_host porque ela é chamada aproximadamente 600 milhões de vezes e no caso da add_urls pela sua complexidade temporal O(quant x (2 x tam + tam_url)), já que ela foi chamada apenas uma vez e mesmo assim teve ao segundo maior custo temporal de todas as funções chamadas nesse programa.

Os próximos na lista de custo temporal foram operações com o tipo string (atribuição, begin, find, erase, size, end, etc.), que são realmente muito utilizadas no programa e claramente muito custosas com relação a tempo de uso.

Em seguida vemos os métodos item_url (construtor da célula da lista) e limpa_tudo que já quase não possuem custo temporal, sendo o maior problema com relação a desempenho temporal as funções get_host e add_urls e operações com strings pelos motivos citados anteriormente.

5.2. Com Relação à Memória

Também podemos avaliar a quantidade de acessos às funções e perceber que as operações que foram mais acessadas na memória são os operadores de strings (já que trabalhamos o tempo todo com esses tipos de dados em nosso programa) e a função get_host que, como falado anteriormente, foi chamada mais de 600 milhões de vezes. Essa última já tinha grande número de chamadas no caso de 506 URLs, em que ela foi chamada 76 mil vezes, porém no último teste o número de chamadas ultrapassou o dos métodos das strings, o que significa que o crescimento do número de chamadas da função get_host e mais rápido que o do número de chamadas dos métodos de manipulação de strings para esse programa.

O método checar_consistencia sempre é chamado uma vez para cada URL a ser adicionada e o método condiciona_url nesse caso também, pois todos os URLs passaram no teste de consistência. Já o construtor e o destrutor do item_url são chamados uma vez a mais pois temos a primeira posição da lista encadeada, que é vazia.

6. Conclusão

Neste trabalho foi criado em C++ um escalonador de URLs baseado em uma lista encadeada de URLs e foram feitos perfis de análise de custo e desempenho do programa e de suas funções separadamente.

Com os testes realizados foi possível notar que as funções que são chamadas diversas vezes são muito custosas, mas a função que adiciona URLs à lista, mesmo com número de chamadas muito menor, pode ter custo igual ou superior a essas funções com várias chamadas. Também pudemos perceber como as operações e métodos do tipo string do C++ podem ser custosas ao programa, pois são chamadas um número enorme de vezes.

Uma observação importante é que, como já foi explicitado no item 4.1 a implementação utilizada nesse programa acaba por gerar aumentar o custo de alguns métodos pois eles precisam percorrer toda a lista quando poderiam percorrer apenas itens específicos, além de que algumas funcionalidades foram perdidas, como a possibilidade de armazenar um host mesmo após esse host ser esvaziado.

Com esse trabalho foi possível aprender ou se aprofundar em conceitos importantíssimos como listas encadeadas, operações com dados e arquivos em C++, tipo string em C++ (e o seu custo), entre outros. Apesar da falta de entendimento da ideia inicial do problema, o que acabou por levar a uma implementação alternativa do sistema requerido, o aprendizado adquirido na execução deste trabalho foi imenso e muito engrandecedor.

Referências

EMBARCADOS. **Análise de desempenho com GNU Profiler gprof**. Disponível em: https://www.embarcados.com.br/desempenho-gnu-profiler-gprof/. Acesso em: 11 nov. 2021.

UFSC.BR. **C++ - Operações com arquivos**. Disponível em: https://moodle.ufsc.br/pluginfile.php/2377820/mod_resource/content/0/exercisios%20arquivos.pdf. Acesso em: 17 dez. 2021.

USP.BR. **Listas encadeadas**. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/lista.html#:~:text=Uma%20lista%20encadeada%20é%20uma,segunda%2C%20e%20assim%20por%20diante.. Acesso em: 17 dez. 2021.

WIKIPEDIA. **Busca em largura**. Disponível em: https://pt.wikipedia.org/wiki/Busca_em_largura. Acesso em: 17 dez. 2021.

WIKIPEDIA. **Lista ligada**. Disponível em: https://pt.wikipedia.org/wiki/Lista_ligada. Acesso em: 17 dez. 2021.

WIKIPEDIA. **Motor de busca**. Disponível em: https://pt.wikipedia.org/wiki/Motor_de_busca. Acesso em: 17 dez. 2021.