Universidade Federal de Minas Gerais
Curso de Sistemas de Informação

Guilherme Teres Nunes

Information Retrieval

Trabalho Prático 02

Belo Horizonte
2021/01

Introdução

Essa é a segunda etapa do trabalho da disciplina de Recuperação de Informação, lecionada pelo professor Berthier Ribeiro-Neto. Para essa parte, os requisitos eram modificar o código anterior (trabalho prático 01) para receber uma lista de URLs sementes, consideradas de nível zero e, partindo dessas, coletar cem mil páginas de nível um.

Apenas páginas HTML deveriam ser coletadas. Não era necessário coletar páginas de nível dois ou superior, mas URLs de outros sites além das sementes fornecidas deveriam ser incluídas no processo do Crawler. A arquitetura pro projeto precisa incluir um scheduler de longo e curto prazo. O processo precisa ser multi threaded para lidar com várias requisições consecutivas para servidores diferentes.

Ao final, o programa deverá listar:

- Todos os websites percorridos.
- O número de URLS de nível 1 percorridas para cada website.
- O tamanho médio de cada página percorrida, por website.
- O tempo médio gasto para buscar cada um dos sites percorridos.

Desenvolvimento: Problemas observados

Imediatamente, vários pontos importantes foram considerados para desenvolver tal aplicação. Não só de tempo, como de performance.

Performance: As várias páginas web sendo carregadas irão requerer muita memória principal (RAM) se não forem tratadas de forma inteligente. Pouca memória disponível pode levar o sistema operacional a entrar em estado de *Trashing*. A quantidade de threads que será executada aumentará muito o uso de CPU. Se não houver um limite, isso poderá até mesmo afetar negativamente a performance, já que o sistema gastará muito tempo alternando entre os threads.

Race Condition também precisa ser levado em consideração, já que os threads precisaram acessar espaços de memória em comum para, por exemplo, verificar ou decrementar a quantidade de URLs restantes, adicionar um novo URL externo a fila para ser percorrido por um novo thread do Crawler assim que possível (scheduling), etc. É importante garantir que onde houver compartilhamento de memória, essa seja feita de forma segura.

Finalmente, o uso de disco também precisa ser observado, já que todos os resultados obtidos serão salvos na memória secundária para uso futuro e também para gerar os relatórios.

Tempo: A largura de banda (bandwidth) da conexão com a internet é um dos fatores que mais influencia no tempo gasto pelo crawler. Algumas páginas podem ser grandes demais e/ou o tempo de resposta muito alto. Antes de efetuar cada consulta, o programa aguarda um tempo de 100ms para evitar más práticas na internet, mas esse não é o maior gargalo: Como será observado nos resultados obtidos, o tempo gasto para obter cada página é consideravelmente maior do que essa espera.

A quantidade de threads inicializados também afetará, de certa forma, o tempo gasto para executar o programa. Cada thread é responsável por buscar um URL diferente, poucos threads significa que poucos URLs podem ser buscados simultaneamente. Muitos threads podem gerar muito overhead e diminuir a performance. Além disso, é necessário levar em consideração que, se a largura de banda já tiver alcançado seu limite, adicionar mais threads pouco ajudará no tempo de execução e também poderá congestionar a rede, tornando-a ainda mais lenta.

Desenvolvimento: Soluções criadas

O programa funciona em duas etapas, que, para fins de praticidade e depuração, podem ser desativadas individualmente removendo ou adicionando essas duas macros no topo do arquivo de código principal (Main.cpp):

```
#define CRAWLER_RUN
#define CRAWLER_ANALYZE
```

A primeira etapa, *CRAWLER_RUN*, executa o crawler a partir das sementes fornecidas, coleta e armazena cem mil URLs em um formato próprio em disco.

Depois, na segunda etapa, *CRAWLER_ANALYZE*, lê os arquivos criados e gera duas saídas em .csv:

Arquivo	Descrição
report.csv	Contém os dados principais requeridos no trabalho: As URLs percorridas, número de URLs nível um, média de tamanho e tempo gasto para cada um dos domínios.
urls.csv	Contém todas as URLs nível um e zero percorridas, bem como as informações (título, tamanho, etc) coletadas e armazenadas na classe *SiteResult* (ver abaixo).

Os detalhes da implementação serão discutidos a seguir.

A primeira adaptação feita no código é a criação de uma classe “File” para facilitar o uso de dados binários em um formato próprio. Essa decisão, além de facilitar a serialização das informações obtidas, gera um ganho de performance tanto na leitura quanto na escrita em disco, além de reduzir consideravelmente o tamanho dos arquivos finais. Ao final do processo de crawler, um relatório em texto no formato .csv é gerado para fins de análise e observação (mas não é necessário para o funcionamento do programa, uma vez que ele é capaz de ler o formato binário). Para título de comparação, o tamanho (em disco) das duas versões dos dados foi exposto abaixo:

Arquivo	Tamanho
Formato Binário Próprio	16MB
Formato em Texto, .csv	42,3MB

(Lembrando que ambos os arquivos contém a mesma informação).

Outra modificação realizada foi na classe *SiteResult*, responsável por armazenar informações de um único website obtido:

```
class SiteResult {
public:
	SiteResult();
	SiteResult(const SiteResult& other);
	virtual ~SiteResult();

	void Save(File* file);
	void Load(File* file);

	void Print();

	std::string title;
	std::string url;
	std::string keywords;
	std::string description;

	double crawlTimeMs;
	size_t pageSize;
private:
};
```

Foram criados métodos para salvar e carregar os dados em disco (utilizando a classe File criada) e alguns campos novos foram acrescentados. Agora cada página captura título, URL, palavras chaves, descrição, tempo gasto no processo de crawling e tamanho da página em bytes. Nem todas as informações obtidas são necessárias para esse trabalho, mas a biblioteca Chilkat disponibiliza tais valores com facilidade, então foi tomada a decisão de incluí-los nos relatórios.

Finalmente, para que o programa fosse possível, uma classe ***WebCrawler*** foi criada. Ela é a responsável por obter as URL sementes e executar o crawler para cada uma

delas. Além de gerenciar os threads e a fila para o agendamento de novas URLs externas. Abaixo, a API pública da classe para facilitar o entendimento:

```
class WebCrawler {
public:
      WebCrawler();
      virtual ~WebCrawler();

      WebCrawler(const WebCrawler& other) = delete;
      WebCrawler& operator=(const WebCrawler& other) = delete;

      void AddToQueue(const std::vector<std::string>& urls);

      void Run();

      ...
};
```

Através dessa API, é possível criar uma nova instância da classe, adicionar uma lista (std::vector<std::string>) de URLs sementes através do método AddToQueue() e finalmente chamar o método Run(), que inicia o processo de crawler, fazendo com que a classe adicione automaticamente um novo thread, se possível, para cada URL na fila. Exemplo de utilização, já com as três URLs sementes propostas para esse trabalho especificamente:

```
std::vector<std::string> out = {
      "http://www.tim.com.br",
      "http://jornaldotempo.uol.com.br",
      "http://casa.abril.com.br"
};

WebCrawler crawl;

crawl.AddToQueue(out);
crawl.Run();
```

É importante mencionar que o programa não inicia imediatamente os novos threads para cada url adicionada. Foi definido que o limite máximo de threads para essa aplicação, na máquina em que foi executada, é de mil threads simultâneos. A partir desse número, o gargalo da aplicação começa a ser a largura de banda, tornando desnecessário a adição de mais processos simultâneos.

Caso não seja possível inicializar mais threads, os URLs são mantidos na fila para serem processados futuramente.

Internamente, quando um thread é iniciado para uma determinada URL, ele cria uma nova instância do *CkSpider* (da biblioteca Chilkat) e começa a solicitar o próximo site obtido por aquela instância, até que não haja mais nenhum não coletado naquele domínio ou que o limite de cem mil URLs seja alcançado. Também foi criado uma interrupção manual pelo usuário:

Durante a execução do programa, basta digitar “q” e apertar Enter no console que o programa irá, gentilmente, interromper os threads, salvar os resultados já coletados e fechar.

É importante mencionar que antes de cada requisição de um novo site, o thread aguarda cem milissegundos como medida de segurança. Após a requisição, ele também verifica se houve algum link outbound e em caso positivo, os adiciona na fila para serem percorridos, através do mesmo método público AddToQueue, que é thread safe.

Perceba que um thread do crawler pode, a qualquer momento, incrementar a quantidade de páginas coletadas e/ou adicionar novos URLs na fila, gerando um forte potencial para problemas de paralelismo, como o Race Condition.

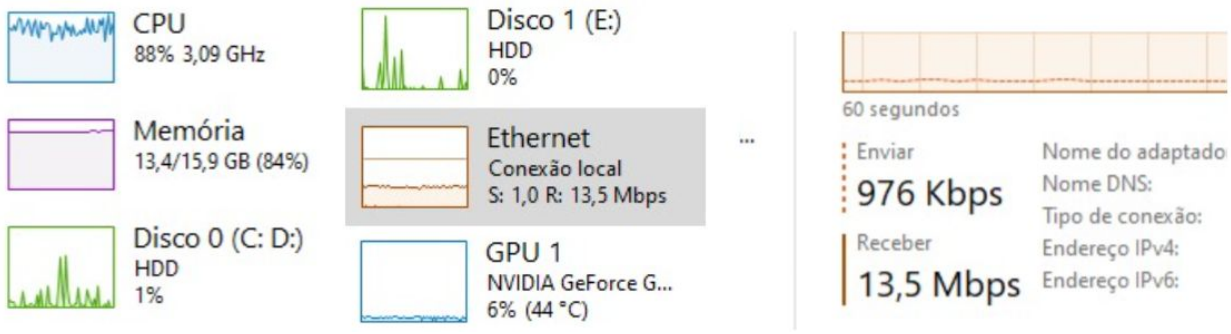
Felizmente, além da estrutura std::thread, a linguagem C++ fornece vários recursos nativos para resolver tais problemas de concorrência. Neste trabalho o formato std::atomic<T> foi utilizado para garantir a segurança de tipos de dados triviais como *int* e *size_t* e o std::mutex para lidar com dados mais complexos, como a fila de URLs e a lista de sites coletados (ambos usam o std::vector internamente).

Resultados Obtidos

Após longas tentativas, os resultados obtidos foram extremamente satisfatórios. O programa levou cerca de uma hora para concluir sua execução, rodando nas seguintes configurações:

- **Processador**: Intel Core I5-4440 @ 3.10GHz
- **Memória RAM:** 16 GB
- Tipo de memória secundária: HD SATA convencional (**não** possui SSD)
- **Internet**:
    - Download: Máximo 15 Megabits
    - Upload: Máximo 1 Megabits

Durante a execução, o uso de recursos do sistema permaneceu muito alto, principalmente o uso da Ethernet. A GPU não foi afetada e, surpreendentemente, o uso de disco foi pouco afetado. Imagem dos recursos no gerenciador de tarefas do Windows durante a execução do programa:



Durante a execução, apenas o Visual Studio e o programa estavam sendo utilizados no computador.

Para ponderar o uso de memória RAM, as únicas informações mantidas persistentemente durante toda a execução do programa eram a lista de threads, URLs na fila e sites já percorridos. Quanto ao restante dos dados, o programa foi desenvolvido para que cada thread salvasse em um arquivo binário novo em disco toda vez que a quantidade de novos sites percorridos ultrapassasse dois mil. Ao salvá-los, eles eram liberados da memória principal.

Os resultados foram positivos: O uso de RAM cresceu linearmente durante a execução e no fim não foi o suficiente para ultrapassar a memória disponível no sistema.

Resultados: Dados Coletados

NOTA: Os arquivos .csv com os dados coletados de forma detalhada foram adicionados a uma planilha .xlsl e está disponível junto a essa documentação.

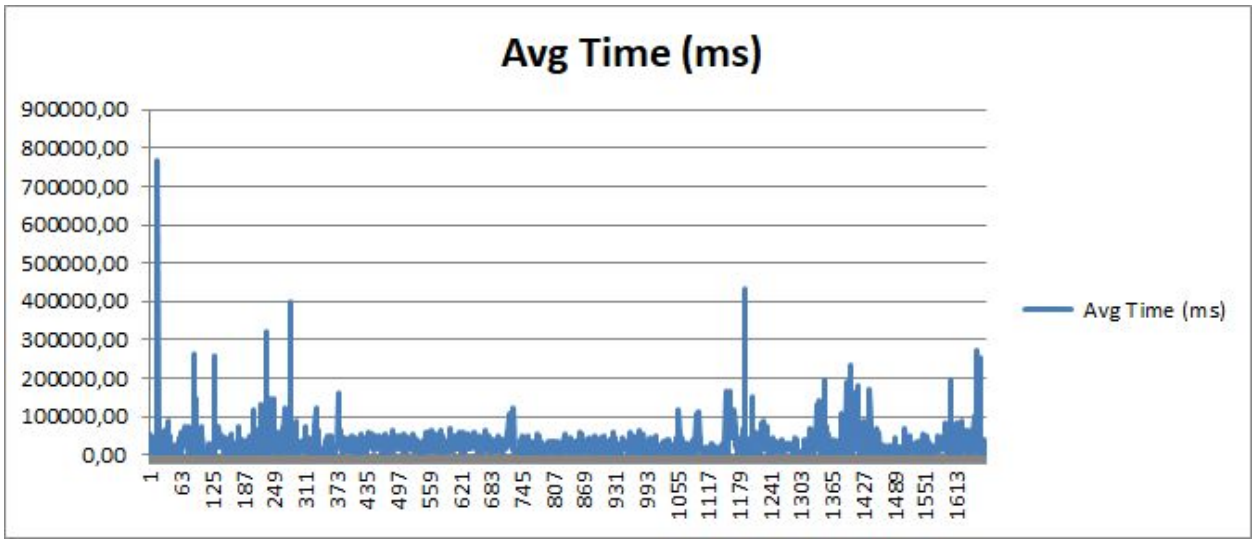
Foram coletadas cem mil páginas. Dessas, uma boa parte era conteúdo não HTML (pdfs, imagens, principalmente, etc). Após a filtragem, foram salvas um pouco mais de cinquenta e sete mil delas.

Contando as três URLs sementes iniciais e todos os links externos contidos dentro delas, foram obtidas 1673 websites. Cada um responsável por uma média de 34 urls nível um.

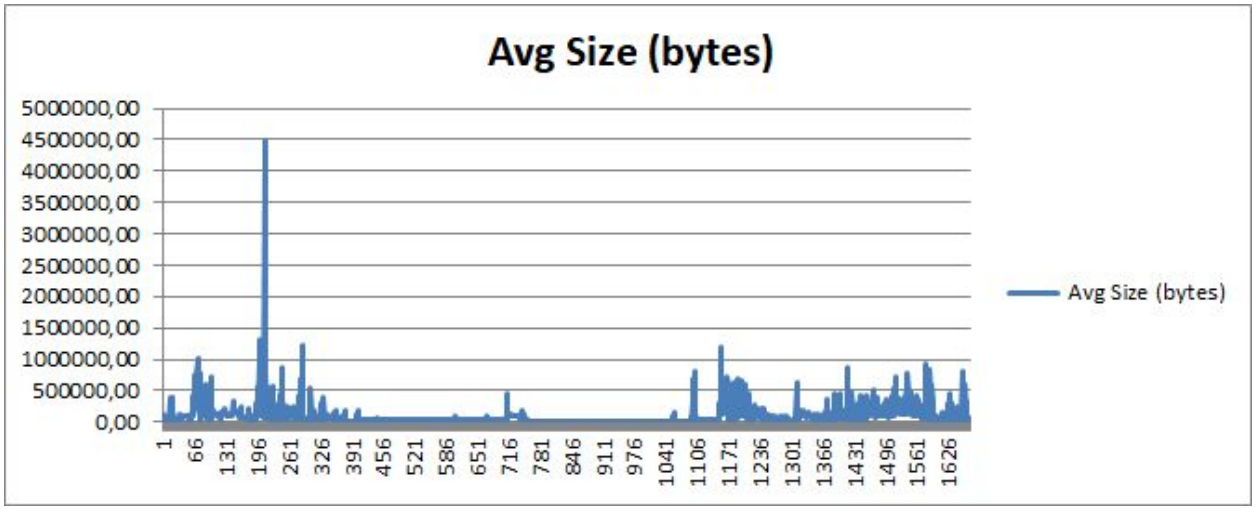
O tempo médio gasto para cada website foi de 28 segundos (descontando dos 100ms de espera entre cada um). Bem maior do que o tempo gasto na primeira parte do trabalho, mas já que haviam até mil threads trabalhando simultaneamente, o resultado foi superior.

Um fator interessante é que o tamanho médio do HTML das páginas foi de 107kb. Ou seja: Foram obtidas seis gigas de HTML ao todo.

No gráfico abaixo, que mostra o tempo médio gasto para acessar a página, é possível observar que há alguns picos extremamente fora da curva:



O mesmo para o tamanho médio das páginas:

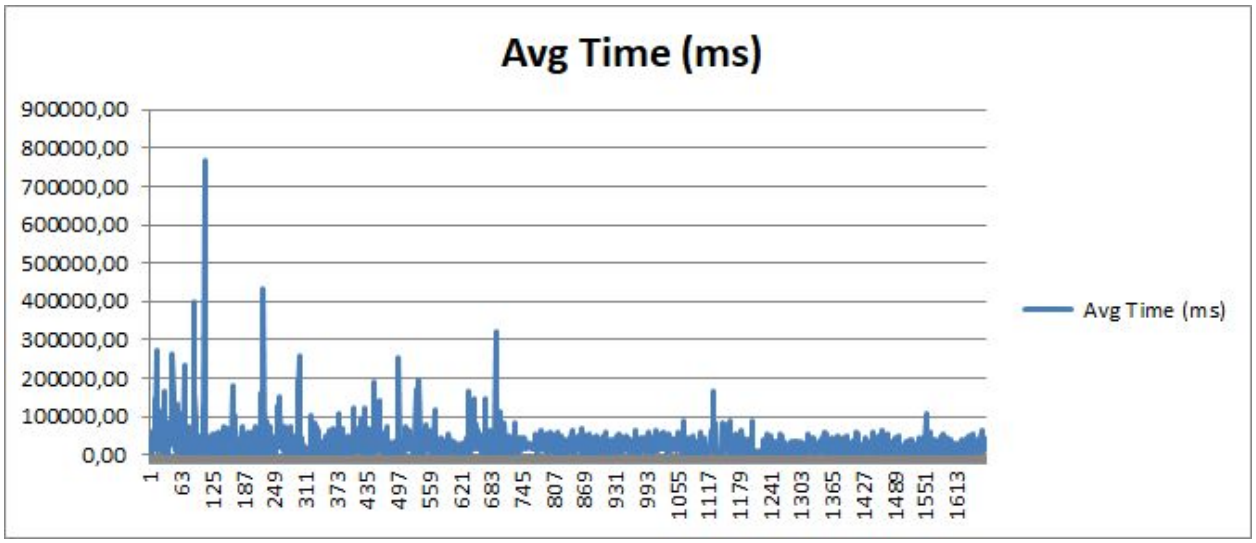


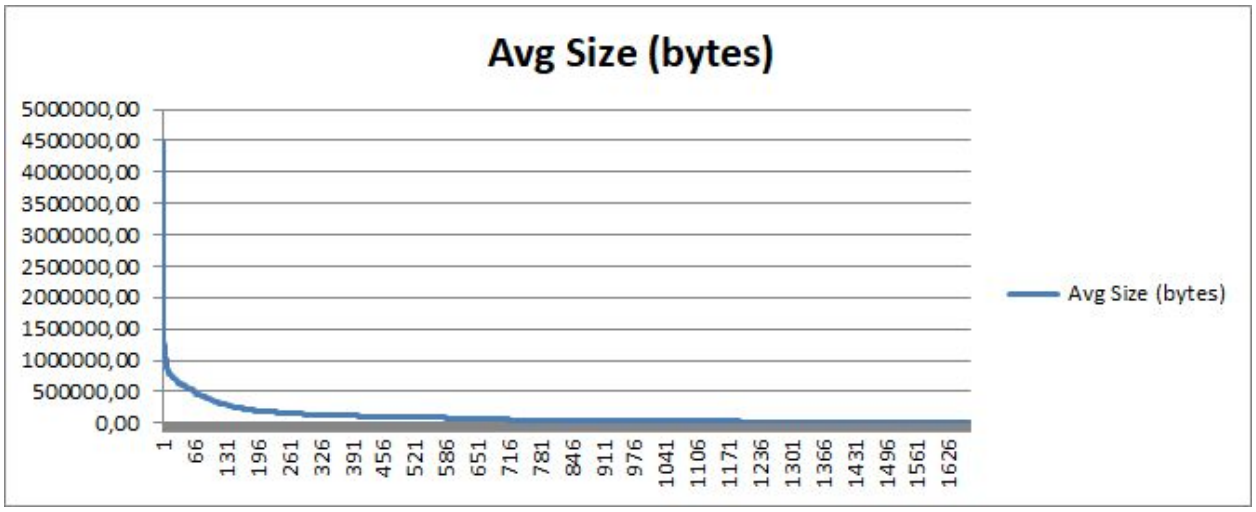
Nota: Os gráficos utilizam as informações médias de cada URL gerador.

Um resultado interessante é que, ao ordenar os dados pelo tamanho das páginas e plotar os gráficos novamente, é possível observar que há uma relação, mas não única,

entre o tamanho da página e o tempo de carregamento (páginas maiores tendem a demorar mais para carregar). Entretanto, é possível observar alguns pontos fora da curva.

Mesmos gráficos, porém ordenados pelo tamanho médio das páginas:





Conclusão

Com essa etapa do trabalho, agora possuo um acervo que contém dezenas de milhares de páginas web e suas informações adicionais, insumo essencial e ideal para o desenvolvimento das próximas etapas.

Graças ao código desenvolvido, as informações podem ser facilmente obtidas novamente e tratadas dentro de programas em C++.

O código fonte desse trabalho pode ser acessado através deste Link:
https://github.com/UnidayStudio/UFMG-InformationRetrieval

Referências

- Chilkat Software: <https://www.chilkatsoft.com/>
- std::Mutex: <https://en.cppreference.com/w/cpp/thread/mutex>
- std::Thread: <https://en.cppreference.com/w/cpp/thread/thread>
- std::Atomic: <https://en.cppreference.com/w/cpp/atomic/atomic>
- URLs sementes propostos para esse trabalho:
    - <http://www.tim.com.br>
    - <http://jornaldotempo.uol.com.br>
    - <http://casa.abril.com.br>
- Código fonte desenvolvido durante o trabalho: <https://github.com/UnidayStudio/UFMG-InformationRetrieval>