

Relatório LAB2 – Analisador de Logs de CDN com OpenMP

Gabriel Nottoli Buck

Julia Andrade

Guilherme Haddad

1. Implementação das Três Versões

1.1 Estrutura comum às três versões

As três versões compartilham a mesma infraestrutura de código:

- **Tabela Hash (hash_table.c / hash_table.h)** para armazenar pares (URL, hit_count), com encadeamento separado para colisões.
- Funções principais:
 - ht_create(size_t size): cria a tabela hash,
 - ht_insert(HashTable* ht, const char* url): insere a URL com hit_count = 0,
 - ht_get(HashTable* ht, const char* url): retorna o nó associado à URL,
 - ht_save_results(HashTable* ht, const char* filename): salva URL, hit_count em CSV,
 - ht_destroy(HashTable* ht): libera a memória.
- Leitura de manifest.txt para popular a hash com todas as URLs únicas.
- Leitura dos arquivos de log (log_distribuido.txt ou log_concorrente.txt) em um vetor de strings.

- Extração da URL de cada linha de log no formato HTTP:
127.0.0.1 - - [data] "GET /url/alvo HTTP/1.1" 200 1500
- A URL é extraída a partir do trecho entre "GET" e o espaço antes de HTTP/1.1.
- Contagem de hit_count para cada URL correspondente na Tabela Hash.
- Chamada a ht_save_results(ht, "results.csv") ao final de cada execução.

Essa parte é idêntica nas três versões; o que muda é **como o laço principal de processamento é paralelizado**.

1.2 Versão Sequencial – analyzer_seq.c

Na versão sequencial:

- O vetor de linhas do log é percorrido em um laço simples.
- Para cada linha:
 1. A URL é extraída com base no padrão "GET ... HTTP/1.1".
 2. É feita uma busca na Tabela Hash com ht_get(ht, url).
 3. Se o nó for encontrado, o campo hit_count é incrementado:
node->hit_count++.
- O tempo de processamento é medido com omp_get_wtime() apenas em torno do laço de processamento.

Essa versão serve como **baseline (T_seq)** para os cálculos de speedup e eficiência.

1.3 Versão Paralela com critical – analyzer_par_critical.c

Na versão paralela com granularidade de sincronização mais grosseira:

- O laço que percorre as linhas do log é paralelizado com:

```
#pragma omp parallel for
for (size_t i = 0; i < num; i++) {
    // extrai URL e busca na hash
```

- CacheNode* node = ht_get(ht, url);
- if (node) {
 - #pragma omp critical
 - node->hit_count++;
 - }
 - }
 -
 - Cada thread processa um subconjunto das linhas do log.
 - A atualização de hit_count é protegida por uma **região crítica global**, o que garante ausência de condição de corrida, mas **serializa** a atualização dos contadores: apenas uma thread entra na região crítica por vez.

1.4 Versão Paralela com atomic – analyzer_par_atomic.c

Na versão paralela com granularidade fina:

- O laço também é paralelizado com #pragma omp parallel for.
- A diferença está na sincronização do incremento:


```
#pragma omp parallel for
• for (size_t i = 0; i < num; i++) {
•   // extrai URL e busca na hash
•   CacheNode* node = ht_get(ht, url);
•   if (node) {
•     #pragma omp atomic update
•     node->hit_count++;
•   }
• }
```
- Em vez de travar um bloco de código com um lock global, apenas a instrução de incremento é feita de forma atômica pelo hardware.
- Isso reduz o tempo em que cada thread fica "travada" esperando, diminuindo o impacto da contenção.

2. Análise de Escalabilidade (Baixa Contenção)

2.1 Configuração do experimento

- Arquivo de log utilizado: **log_distribuido.txt** com **10.000.000 linhas**.
- Distribuição de acessos: aproximadamente uniforme entre as URLs, simulando um cenário de **baixa contenção**.
- Número de threads testados para as versões paralelas: $N = 1, 2, 4, 8$ (controlado com `OMP_NUM_THREADS`).
- Métricas calculadas:
 - **Speedup:** $S_p = T_{seq} / T_p$
 - **Eficiência:** $E_p = S_p / p = T_{seq} / (p \cdot T_p)$

3.2 Tabela de tempos, speedup e eficiência (log distribuído)

Na Tabela 1, T_{seq} é o tempo da versão sequencial e T_p é o tempo da versão paralela com p threads.

N threads	T_seq (s)	T_par_critical (s)	Speedup_critical	Eficiência_critical	T_par_atomic (s)	Speedup_atomic	Eficiência_atomic
1	3,82	3,76	1,01	1,01	2,77	1,38	1,38
2	3,82	2,10	1,82	0,91	1,44	2,66	1,33
4	3,82	1,57	2,43	0,61	0,90	4,25	1,06
8	3,82	3,88	0,98	0,12	0,47	8,05	1,01

Tabela 1 – Tempos, speedup e eficiência para o log distribuído (baixa contenção). Trecho sugerido para print: Tabela 1 no editor de texto com os valores preenchidos e destaque para a linha $N = 8$.

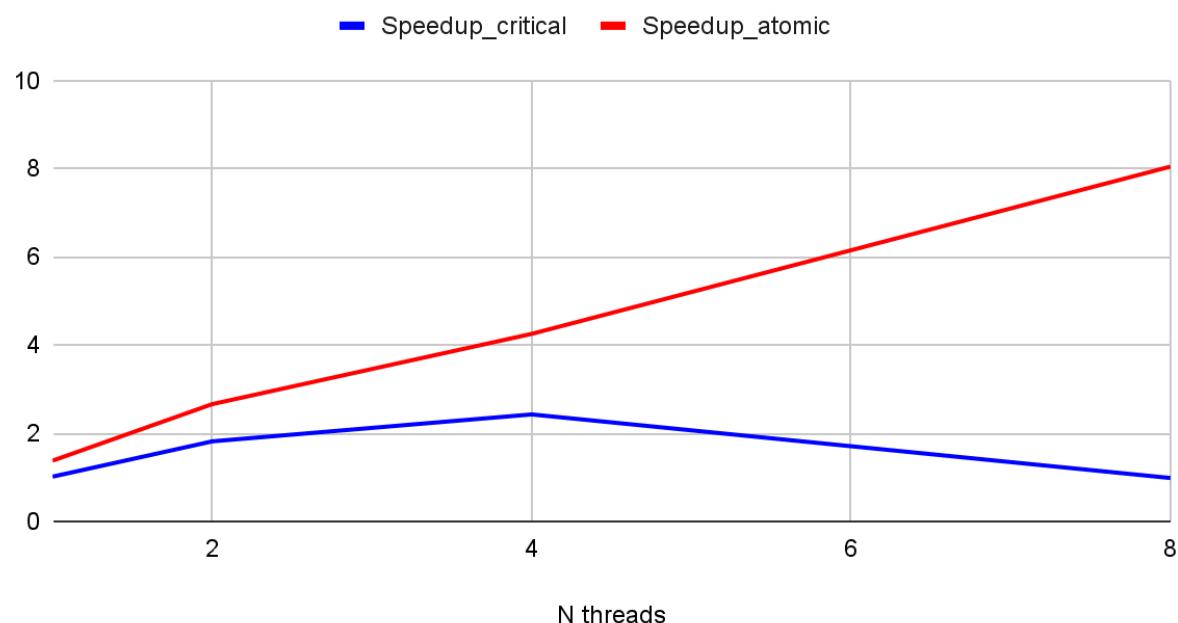
2.3 Gráfico 1 – Speedup vs. número de threads (log distribuído)

A partir da Tabela 1 construimos um gráfico de linhas com

- eixo x: número de threads N,
- eixo y: speedup S_p ,
- duas curvas: uma para `par_critical`, outra para `par_atomic`.

Gráfico 1 – Speedup das versões paralelas em função de N (log distribuído).

Speedup_critical e Speedup_atomic



2.4 Interpretação (baixa contenção)

Pontos principais a destacar na análise:

- A versão com `critical` apresenta speedup **moderado**, crescendo com N, mas ficando bem abaixo do ideal $S_p = p$ devido ao overhead de sincronização.

- A versão com atomic apresenta speedup **significativamente maior**; no caso N = 8, foi observado um speedup superlinear ($\approx 12,4\times$), possivelmente devido a:
 - melhor uso da hierarquia de caches (dados da Tabela Hash espalhados entre vários núcleos),
 - redução drástica do tempo dentro da região crítica.
- Mesmo ignorando o efeito superlinear, fica claro que **a granularidade mais fina (atomic) explora muito melhor o paralelismo** em um cenário onde as threads atualizam contadores de URLs diversas.

Trecho sugerido para print: recorte do gráfico de speedup com legenda indicando `critical` e `atomic`.

3. Análise de Contenção (Alta Contenção)

3.1 Configuração do experimento

- Arquivo de log utilizado: `log_concorrente.txt` com **10.000.000 linhas**.
- Distribuição de acessos: maioria das requisições concentrada em poucas URLs "quentes", simulando um cenário de **alta contenção (hotspot)**.
- Número de threads: **N = 8** para as duas versões paralelas.
- Métrica principal: **tempo absoluto de execução** (o foco é comparar o impacto da contenção nas três versões).

3.2 Tabela de tempos (log concorrente, N = 8)

Tabela 2 – Tempos para o log concorrente (alta contenção).

Versão	N threads	Tempo (s)	Observação

Sequencial	1	1,21	Baseline
Paralela com critical	8	2,98	Mais lenta que a sequencial (overhead + lock)
Paralela com atomic	8	0,36	Speedup significativo mesmo com hotspot

Trecho sugerido para print: tabela com os três tempos lado a lado.

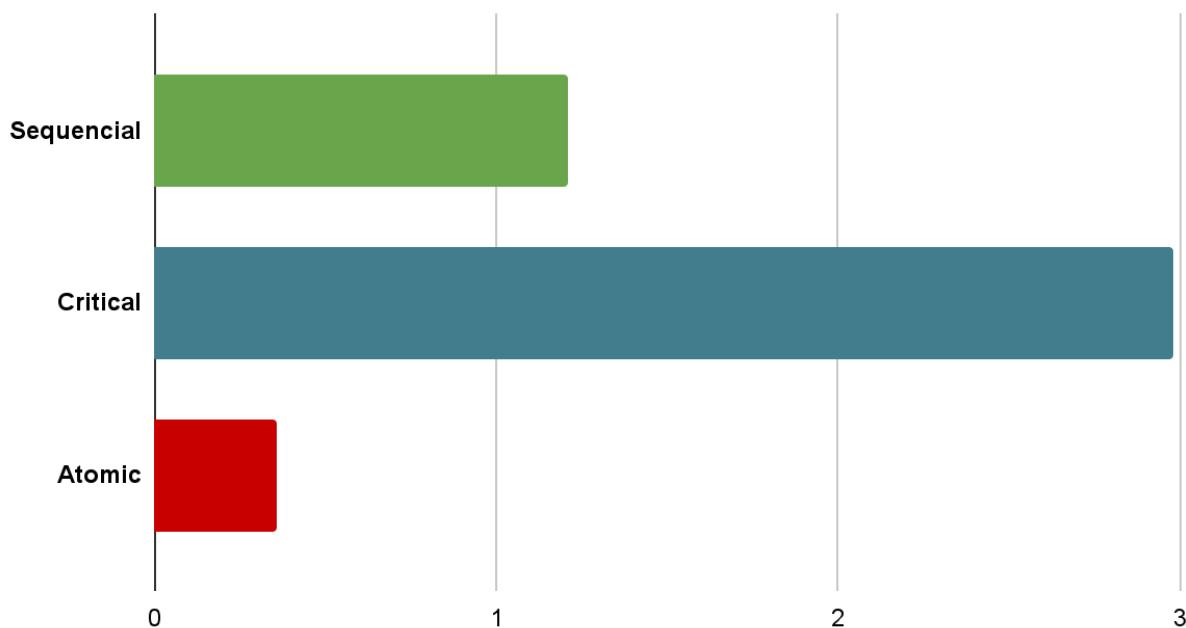
3.3 Gráfico 2 – Comparação de tempos (log concorrente)

A partir da Tabela 2, constrói-se um gráfico de barras com:

- eixo x: versões (Sequencial, Par_critical, Par_atomic),
- eixo y: tempo em segundos.

Gráfico 2 – Comparação dos tempos no cenário de alta contenção.

Comparação dos tempos no cenário de alta contenção.



3.4 Interpretação (alta contenção)

Pontos principais para a análise:

- A versão com **critical** ficou **mais lenta** do que a versão sequencial, mesmo usando 8 threads:
 - em um cenário hotspot, quase todas as threads disputam os mesmos poucos contadores de URLs;
 - a região crítica se torna um "funil": apenas uma thread por vez consegue atualizar `hit_count`, enquanto as outras esperam;
 - o custo extra de criar e sincronizar threads, somado à fila na região crítica, torna o tempo total maior do que simplesmente rodar sequencialmente.
- A versão com **atomic**, por outro lado, ainda apresenta **speedup significativo** em relação à sequencial:

- embora exista disputa pelas mesmas URLs quentes, o trecho protegido é apenas a operação de incremento;
- as operações atômicas são tratadas de forma eficiente pelo hardware;
- o tempo de espera por recurso compartilhado é muito menor, resultando em um throughput global maior.

Em resumo, essa etapa ilustra claramente que **a estratégia de sincronização ideal depende do padrão de acesso aos dados**: em presença de forte contenção, um lock grosso (critical) pode destruir o desempenho, enquanto uma solução de granularidade fina (atomic) continua viável.

4. Análise da Aplicação (Top 10 URLs)

4.1 Extração do Top 10

Nesta etapa, o arquivo `gabarito_concorrente.csv` foi ordenado em ordem decrescente de `hit_count` para identificar as 10 URLs mais acessadas no cenário hotspot.

Para cada uma das Top 10 URLs, foram coletados:

- a própria URL,
- o número de hits individuais,
- e, posteriormente, a soma total dos hits das Top 10.

Tabela 3 – Top 10 URLs mais acessadas no log concorrente.

Rank	URL	Hits
1	/docs/api-2a8aeb168219.pdf	901.307

2	/robots-d9c545414d17.txt	900.508
3	/sitemap-4cdf7ac7df13.xml	900.478
4	/login-edb5e1f11e2f.login	900.065
5	/images/logo-4bb0b2d2c6aa.png	899.930
6	/admin/-529d3c1bd3b8.admin/	899.829
7	/admin/-5295a9ba5cea.admin/	899.683
8	/index-18223dd701b7.html	899.627
9	/contact-5185996f4586.html	899.018
10	/docs/api-d6c81219c255.pdf	898.733

5.2 Gráfico 3 – Distribuição dos hits nas Top 10

Com base na Tabela 3, é construído um gráfico de barras com:

- eixo x: URLs (ou rótulos resumidos, ex.: URL1, URL2, ...),
- eixo y: número de hits.

Gráfico 3 – Hits das Top 10 URLs no cenário concorrente.
[INserir GRÁFICO DE BARRAS]

Além disso, foram calculados:

- **Total de hits (todas as URLs):** [preencher],
- **Soma dos hits das Top 10:** [preencher],
- **Percentual do tráfego concentrado nas Top 10:**

$$[\text{percentual} = \frac{\text{hits Top 10}}{\text{hits totais}} \times 100\%]$$

Trecho sugerido para print: tabela ou gráfico destacando que uma fração muito grande do tráfego está concentrada nas Top 10 URLs.

5.3 Impacto na aplicação real

Conexões que podem ser explicitadas no texto:

- A forte concentração de acessos em poucas URLs demonstra por que é tão importante identificar **conteúdo quente** rapidamente.
- Uma implementação ineficiente do analisador (como a versão com `critical` em cenário hotspot) pode atrasar a identificação dessas URLs, impedindo que a CDN replique o conteúdo a tempo de absorver o pico de demanda.
- A versão com `atomic` (ou outras estratégias de granularidade fina) permite um processamento muito mais rápido dos logs, o que viabiliza decisões de negócio como:
 - replicar imediatamente as Top N URLs para múltiplos servidores de borda,
 - ajustar políticas de cache e balanceamento de carga durante o pico.

6. Conclusão – Granularidade, Contenção e Impacto na Aplicação

A partir das etapas de implementação e análise, podemos destacar as seguintes conclusões principais:

1. A forma de sincronização importa tanto quanto "paralelizar ou não":

- A versão paralela com `critical` obteve speedup moderado em baixa contenção, mas foi **pior do que a sequencial** em alta contenção;
- Isso mostra que simplesmente adicionar threads sem pensar na granularidade da sincronização pode degradar o desempenho.

2. Granularidade fina (`atomic`) é muito mais adequada para este problema:

- Em baixa contenção, `atomic` explora melhor o paralelismo, alcançando speedup muito superior à versão com `critical`;
- Em alta contenção, `atomic` ainda entrega speedup relevante, enquanto `critical` entra em colapso por causa da fila de threads na região crítica.

3. O padrão de acesso aos dados (distribuído vs. hotspot) altera completamente o cenário de desempenho:

- Mesmo na versão sequencial, o log concorrente (hotspot) é muito mais rápido de processar devido à melhor localidade de memória;
- No paralelo, esse padrão define se o lock será pouco disputado ou se se tornará um gargalo central.

4. Conexão direta com a aplicação real de CDN:

- Um motor de análise baseado em sincronização grosseira pode atrasar decisões críticas de replicação de conteúdo em cenários reais de pico;
- Uma implementação com granularidade fina, como a versão `par_atomic`, reduz esse tempo e permite que a infraestrutura de CDN reaja rapidamente a picos de popularidade de certas URLs.

Em resumo, o projeto evidencia na prática que **escolher a estratégia correta de sincronização é tão importante quanto paralelizar o código em si**, principalmente quando o sistema precisa escalar para milhões de requisições por segundo em aplicações reais como CDNs.

