



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
DIM0124 - PROGRAMAÇÃO CONCORRENTE

Trabalho Prático

Um estudo empírico sobre programação concorrente com threads

João de Souza Fernandes Vieira
Rafael Fortunato de Paula Pereira

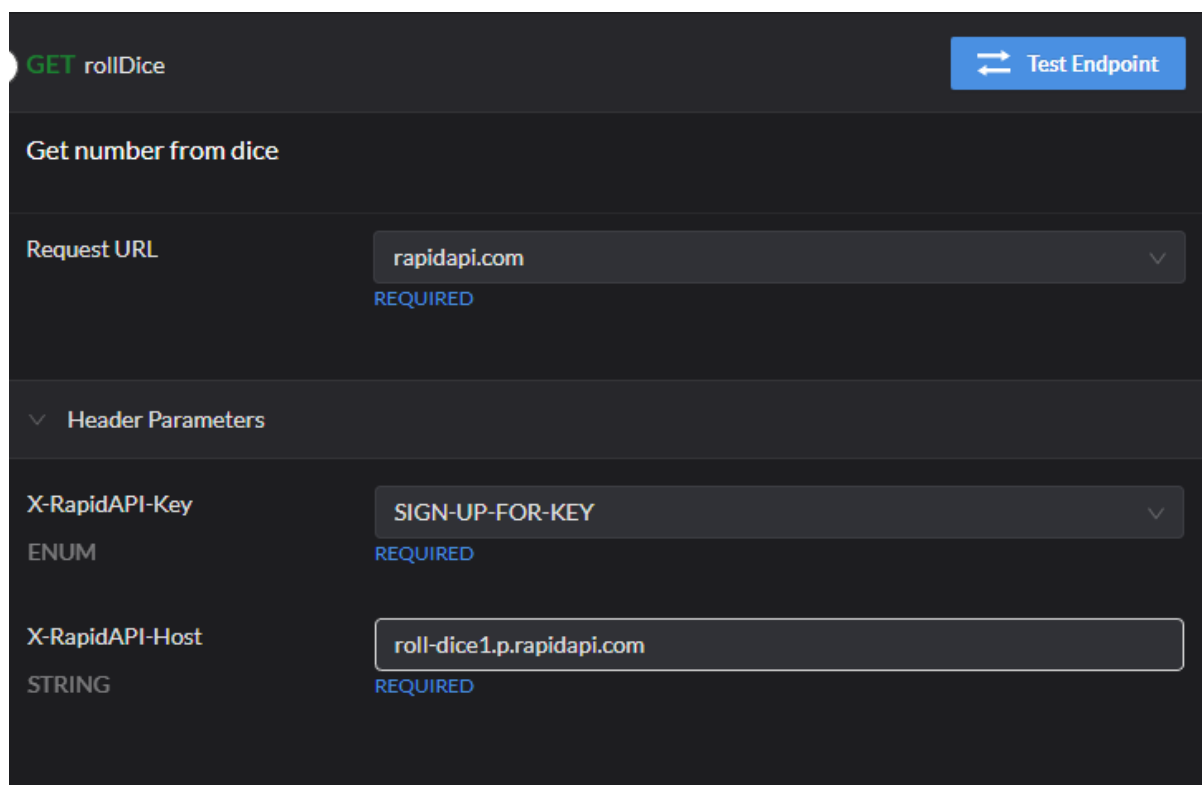
Natal - RN
2022

Introdução	3
Metodologia	5
Resultados	7
Tabelas	7
Implementação sequencial	7
Implementação concorrente	7
Ganho de desempenho (speedup)	8
Gráficos	9
Conclusões	11

Introdução

Foram desenvolvidos dois programas, na linguagem de programação C++, o primeiro sendo a versão sequencial, e o segundo uma solução alternativa utilizando concorrência. Para realizar as requisições *http*, foi utilizada a biblioteca *libcurl*. Para habilitar o suporte a *threads*, a biblioteca *threads.h* foi importada para o projeto.

O serviço escolhido foi o Roll Dice, encontrado no repositório RapidAPI. Esse serviço simula a rolagem de um dado, retornando um número inteiro entre 1 e 6 a cada requisição.



The image shows the RapidAPI interface for the 'rollDice' endpoint. At the top left, it says 'GET rollDice'. At the top right, there is a blue button with a double arrow icon and the text 'Test Endpoint'. Below this, the endpoint description 'Get number from dice' is shown. The 'Request URL' field is set to 'rapidapi.com' and is marked as 'REQUIRED'. Under the 'Header Parameters' section, the 'X-RapidAPI-Key' field is set to 'SIGN-UP-FOR-KEY' (marked as 'REQUIRED' and 'ENUM') and the 'X-RapidAPI-Host' field is set to 'roll-dice1.p.rapidapi.com' (marked as 'REQUIRED' and 'STRING').

O programa sequencial recebe o número **N** de iterações desejadas como argumento, e então executa um laço *for* **N** vezes. A cada iteração, uma requisição *GET* é enviada ao serviço, e é esperado e verificado que a resposta recebida possua um *status code* *OK*, indicando que a requisição foi processada e respondida com sucesso.

```

if (curl) {
    curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "GET");
    curl_easy_setopt(curl, CURLOPT_URL, "https://roll-dice1.p.rapidapi.com/rollDice");

    struct curl_slist *headers = NULL;

    headers = curl_slist_append(headers, "X-RapidAPI-Key: 0197282ec9msh98c513dba9e4a78p140cd4jsna0d89d7bc822");
    headers = curl_slist_append(headers, "X-RapidAPI-Host: roll-dice1.p.rapidapi.com");
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
    curl_easy_setopt(curl, CURLOPT_NOBODY, 1);

    auto start = std::chrono::high_resolution_clock::now();
    // CALL iterations TIMES TO API
    for (int i = 0; i < iterations; ++i){
        res = curl_easy_perform(curl);
        if (res != CURLE_OK){
            fprintf(stderr, "curl_easy_perform() returned %s, Aborting execution because samples were compromised...\n", curl_easy_strerror(res));
            return 1;
        }
    }
    curl_easy_cleanup(curl);
}

```

De forma similar, a solução concorrente possui a mesma lógica de envio de requisições e tratamento de resposta positiva. A diferença, entretanto, é que cada requisição será processada por uma *thread* diferente, em paralelo.

```

std::vector<std::thread> threads;

auto start = std::chrono::high_resolution_clock::now();

for (int i = 0; i < iterations; ++i){
    threads.emplace_back([&]{ //creates and starts a thread
        CURL *curl = curl_easy_init();
        if (curl) {
            curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "GET");
            curl_easy_setopt(curl, CURLOPT_URL, "https://roll-dice1.p.rapidapi.com/rollDice");
            struct curl_slist *headers = NULL;
            headers = curl_slist_append(headers, "X-RapidAPI-Key: 0197282ec9msh98c513dba9e4a78p140cd4jsna0d89d7bc822");
            headers = curl_slist_append(headers, "X-RapidAPI-Host: roll-dice1.p.rapidapi.com");
            curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
            curl_easy_setopt(curl, CURLOPT_NOBODY, 1); // Comment to see request printed on terminal
            curl_easy_perform(curl);
            curl_easy_cleanup(curl);
        }
    });
}

for (auto& t : threads) { // wait for all threads to finish
    t.join();
}

auto end = std::chrono::high_resolution_clock::now();

```

Metodologia

A máquina utilizada para execução de ambos os programas e geração dos dados possuía as seguintes características técnicas:

```
Intel® Core™ i7-7700HQ CPU @ 2.80GHz  
8gb RAM  
Ubuntu 22.04 LTS  
g++-11 (Ubuntu 11.2.0-19ubuntu1) 11.2.0
```

Para a coleta dos tempos de execução para cada requisição, foi utilizada a biblioteca `chrono` para C++. Capturamos a hora do sistema antes e após cada requisição `http`, subtraindo o “após” do “antes”, assim obtendo o tempo despendido para cada requisição. Esses valores são formatados em segundos e milissegundos, e impressos em arquivos de texto armazenados na pasta `logs` na raiz do projeto.

```
logs > concurrency.txt  
51 Requisitions: 20 | Time elapsed: 3661 ms | 3.661 s  
52 Requisitions: 20 | Time elapsed: 3134 ms | 3.134 s  
53 Requisitions: 20 | Time elapsed: 5455 ms | 5.455 s  
54 Requisitions: 20 | Time elapsed: 5720 ms | 5.72 s  
55 Requisitions: 20 | Time elapsed: 5982 ms | 5.982 s  
56 Requisitions: 20 | Time elapsed: 3007 ms | 3.007 s  
57 Requisitions: 20 | Time elapsed: 7761 ms | 7.761 s  
58 Requisitions: 20 | Time elapsed: 3099 ms | 3.099 s  
59 Requisitions: 20 | Time elapsed: 2927 ms | 2.927 s  
60 Requisitions: 20 | Time elapsed: 3400 ms | 3.4 s  
61 Requisitions: 20 | Time elapsed: 2930 ms | 2.93 s  
62 Requisitions: 20 | Time elapsed: 2886 ms | 2.886 s  
63  
64 Requisitions: 50 | Time elapsed: 10538 ms | 10.538 s  
65 Requisitions: 50 | Time elapsed: 7515 ms | 7.515 s  
66 Requisitions: 50 | Time elapsed: 9787 ms | 9.787 s  
67 Requisitions: 50 | Time elapsed: 8667 ms | 8.667 s  
68 Requisitions: 50 | Time elapsed: 8212 ms | 8.212 s
```

Caso uma requisição tenha resposta diferente de 200 (*HTTP STATUS CODE OK*), o programa informa o erro no terminal e encerra. Dessa forma, eliminamos execuções com falhas das amostras de dados, que influenciariam nas médias e desvios padrões.

Foram realizados sete cenários diferentes, com 20 repetições de N requisições. As 20 repetições por amostra foram necessárias para gerar dados consistentes, além de podermos eliminar valores mínimos e máximos, e nos aproximarmos da realidade. Por limitações de tempo, não foram realizados cenários com números de requisições superiores a 500, que levou cerca de 1h20m de processamento para realizar as 20 execuções.

Quanto à metodologia de análise dos dados obtidos, foram gerados gráficos de linhas e tabelas. Além disso, foram calculados os desvios padrões e o ganho de desempenho (*speed-up*) entre as duas implementações.

Resultados

Tabelas

Implementação sequencial

Número de Requisições	Tempo Mínimo (ms)	Tempo Médio (ms)	Tempo Máximo (ms)	Desvio Padrão (ms)
1	800 ms	969,55 ms	1.213 ms	121,52 ms
10	5.402 ms	6.069,15 ms	9.913 ms	963,38 ms
20	10.524 ms	11.015,1 ms	13.889 ms	797,35 ms
50	26.642 ms	27.306,4 ms	33.759 ms	6.200,39 ms
100	51.671 ms	67.214,8 ms	111.643 ms	14.559,79 ms
200	104.020 ms	111.307,5 ms	149.085 ms	10.672,39 ms
500	254.047 ms	261.243,05 ms	317.733 ms	14.177,50 ms

Implementação concorrente

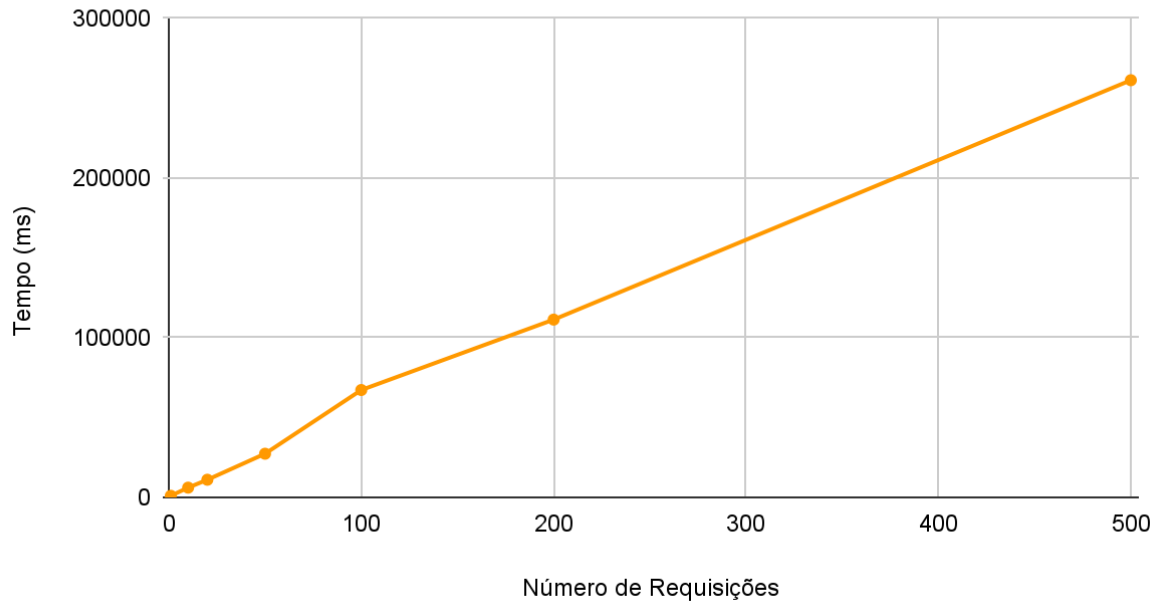
Número de Requisições	Tempo Mínimo (ms)	Tempo Médio (ms)	Tempo Máximo (ms)	Desvio Padrão (ms)
1	796 ms	918,35 ms	1.110 ms	89,38 ms
10	1.859 ms	3.044,7 ms	4.320 ms	628,68 ms
20	2.766 ms	4.315,31 ms	7.888 ms	1.830,26 ms
50	7.050 ms	8.359,05 ms	11.152 ms	1.208,66 ms
100	12.085 ms	13.043,52 ms	14.586 ms	768,10 ms
200	25.503 ms	26.784,52 ms	31.577 ms	1.269,17 ms
500	62.992 ms	66.781,94 ms	70.775 ms	1.547,47 ms

Ganho de desempenho (speedup)

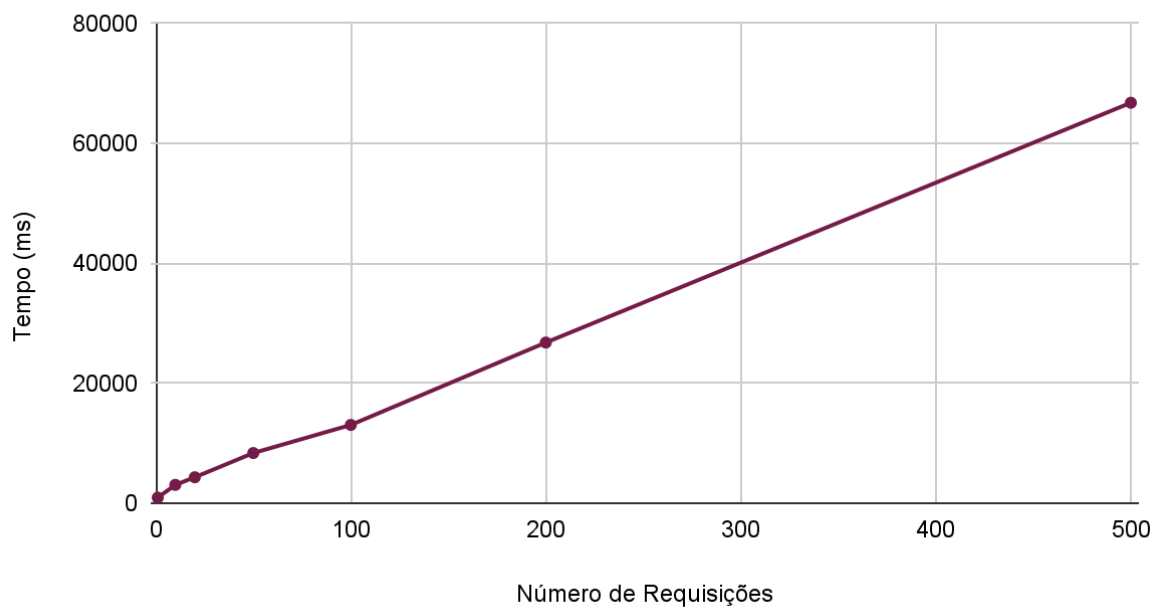
Número de Requisições	Ganho de desempenho
1	1,055
10	1,993
20	2,552
50	3,266
100	5,153
200	4,155
500	3,911

Gráficos

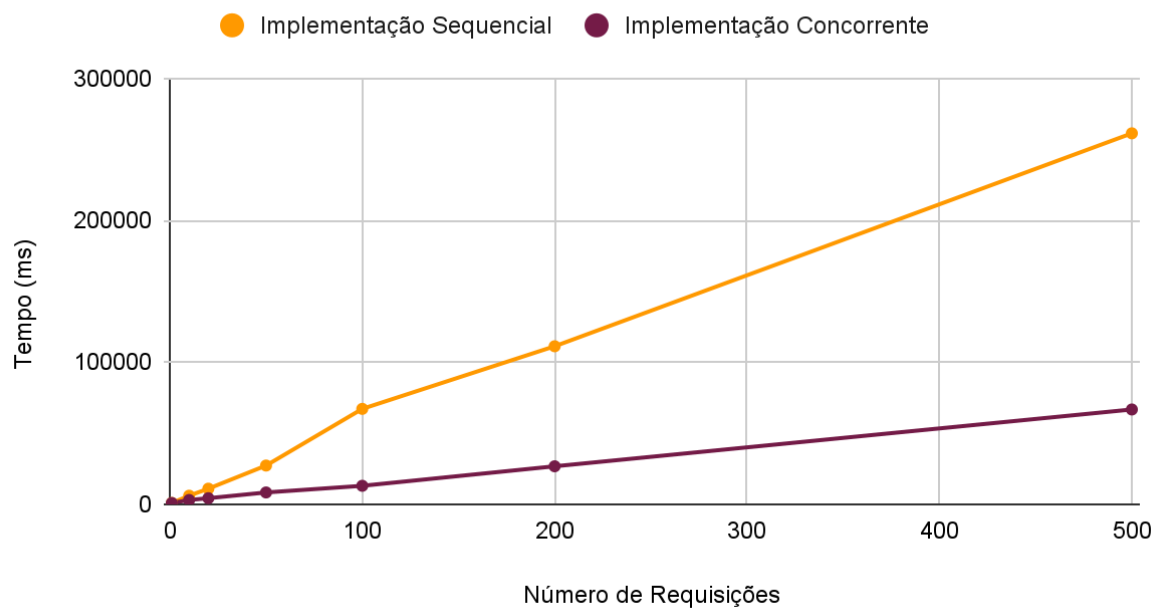
Implementação Sequencial



Implementação Concorrente



Sequencial x Concorrente

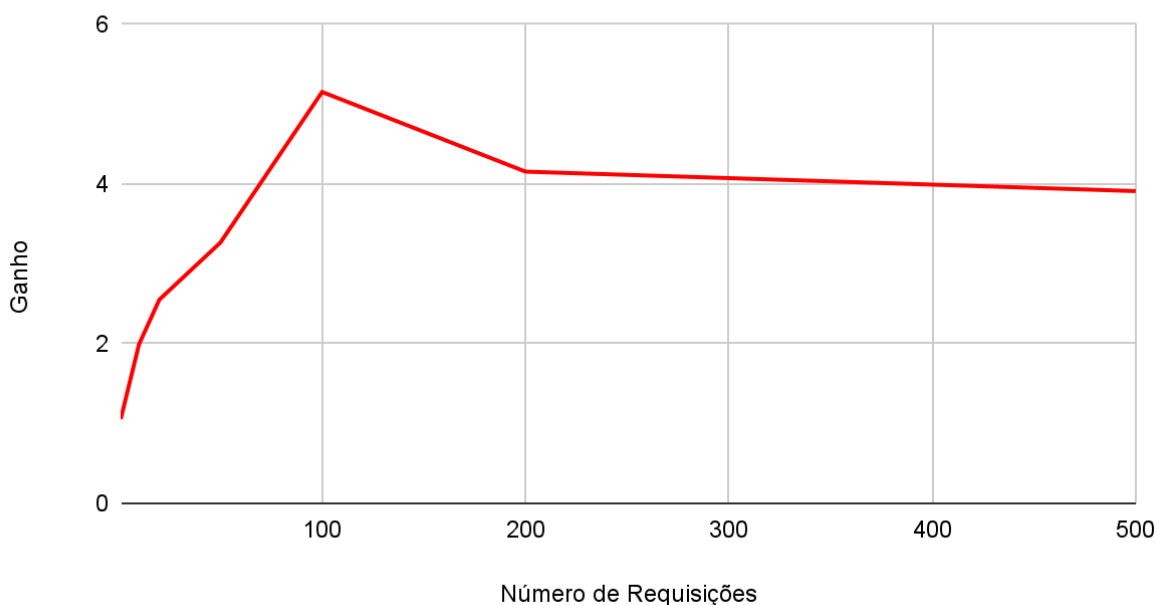


Conclusões

Analisando as tabelas e gráficos gerados, percebemos que a versão concorrente da aplicação obteve resultados melhores em todos os casos (mínimos, médios e máximos). Além disso, pode-se notar que o desvio padrão, menos no cenário de 20 requisições, foi consideravelmente menor na aplicação que faz uso de *threads* concorrentes.

Os dados de *speed-up*, apresentaram um resultado interessante. O ganho de desempenho quase dobra ao se subir um cenário, até alcançar surpreendentes 5,153 no momento das 100 requisições. Entretanto, a partir daí, o *speed-up* apresenta uma tendência de queda, indicando que o ganho ao se utilizar concorrência, em detrimento do modelo tradicional sequencial, não é tão expressivo quanto nos outros cenários investigados.

Speed-up x Número de Threads



Esse resultado nos leva a crer que para execuções com cenários maiores ainda, com 1000 ou 10.000 requisições, e consequentemente *threads*, a vantagem em se adotar a estratégia concorrente não seja tão grande, como nos cenários iniciais. O trabalho prático realizado incita a curiosidade e a vontade de prosseguir a experimentação com novos cenários, aplicações e metodologias.