AnagraMe Concorrente

Por João Marcelo Rodrigues de Andrade e João Pedro Coelho de Souza

Relatório Final

Programação Concorrente (ICP-361) — 2023/1

1. **Descrição do problema:**

   AnagraMe é um jogo cujo objetivo é pontuar o máximo possível formando palavras utilizando somente uma certa quantidade de letras que é disponibilizada ao jogador. Quanto mais complexas as palavras formadas, mais pontos o jogador ganha. Ao iniciar uma partida, o jogador escolhe um nível de dificuldade de 1 a 3 onde cada nível varia os fatores *Tempo* e *Quantidade de letras disponíveis* do jogo. Como esperado, quanto menos tempo e letras disponíveis, mais difícil será para acumular pontuações e, consequentemente, subir no ranking de jogadores. Para pontuar, o jogador deve entrar com palavras válidas, ou seja:

   i. Palavras contendo somente as letras disponibilizadas (inclusive respeitando a quantidade oferecida de cada letra);
   ii. Palavras existentes no dicionário;
   iii. Palavras não muito curtas (mínimo de três letras);
   iv. Palavras não validadas anteriormente.

   Para validar a regra (ii), isto é, se a palavra digitada pelo jogador existe de fato, nós utilizamos um arquivo de texto composto por todas as palavras da língua portuguesa, onde, no início da execução, cada palavra é lida e armazenada em um vetor de strings chamado *dicionario*. Com isto, a validação se dá através de uma iteração neste vetor, comparando a entrada com cada palavra armazenada. Atualmente, o dicionário da língua portuguesa possui algo por volta de 320.000 palavras, e todas as vezes que o jogador entra com uma nova palavra, a mesma pode ter que ser testada com grande parte desse montante antes de ser encontrada, podendo até chegar nas 320.000 comparações caso a palavra não exista. Pelo fato das partidas possuírem um tempo limitado, minimizar o tempo gasto na validação das palavras digitadas pelo jogador pode ser um fator decisivo para quebrar um recorde ou não, e por isso, talvez a concorrência possa nos auxiliar neste caso.

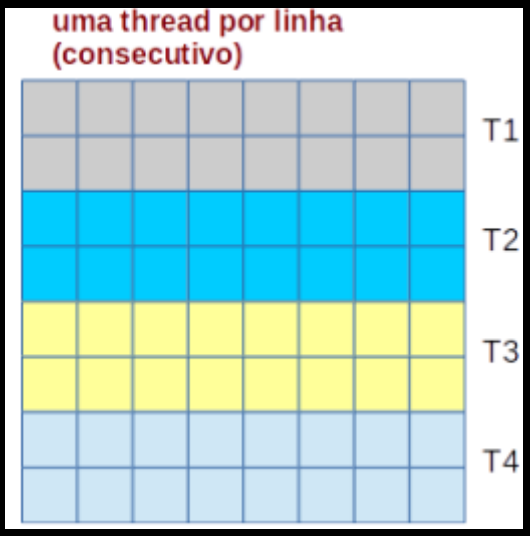
2. **Projeto da solução concorrente:**

A ideia de uma versão concorrente do AnagraMe basicamente passa por particionar entre vários fluxos de execução a missão de validar se a palavra digitada pelo usuário de fato existe no nosso dicionario, com a justificativa de que forma sequencial, aparentemente possui considerável custo computacional; já que o dicionário tem cerca de 320,000 palavras.

Neste caso, nosso recurso que será compartilhado entre as threads, ou seja, o dicionário, trata-se de vetor de strings, onde em cada posição do mesmo, está registrada uma palavra existente na língua portuguesa. Para definirmos a carga de processamento, diga-se, a quantidade de linhas pelas quais cada thread ficará responsável, existem algumas soluções:

a. Individual: Criar uma thread para processar cada uma das linhas do arquivo. Ex:

b. Blocos: Dividir o arquivo em blocos contínuos com a mesma quantidade de linhas e passar cada bloco para uma thread distinta. Ex:



c. Linhas alternadas: Dada uma quantidade definida de threads, implementar uma espécie de rodízio onde cada thread se responsabiliza pelas linhas de forma alternada entre si. Ex:

Sabemos que nosso arquivo de dicionário possui todas as palavras em ordem alfabética, porém, pelo fato de não ser possível prever qual será o valor lexicográfico da palavra que queremos validar (se é que ela existe no dicionário), não importa qual método de particionamento da tarefa utilizaremos, desde que possamos garantir que as thread tenha cargas de processamento balanceadas entre si. Dito isso, optamos por implementar a alternativa C, onde a distribuição das linhas do arquivo se dará de maneira alternada entre as threads, mas sempre buscando equilibrar o máximo possível a quantidade de linhas para cada fluxo.

Com isso em mente, podemos esboçar um pequeno escopo do que seria uma partida do jogo utilizando a validação de existência da palavra concorrente:

```
Enquanto (tempo disponível):
      imprime letras disponíveis();
      lê palavra digitada();
      se ( valida palavra() ):
            pontua();
      se não:
            imprime mensagem de erro();
```

Dentro de `valida palavra()` , uma série de testes com a palavra digitada a fim de descobrir se ela cumpre os requisitos para realizar a pontuação. Uma dessas validações é onde entra a lógica concorrente do programa: `valida palavra existente()` é uma sub rotina que cria e dispara as (por default, quatro) threads, onde cada uma executará um trecho de código parecido com:

```
busca palavra(palavra):
      para cada ( dicionario [ t_id * num_threads ] ):
            se (  !achou ):
                  se ( palavra == dicionario[i] ):
                      achou = true;
```

`busca palavra()` é a função executada pelas threads que de fato realiza as iterações no vetor dicionário procurando pela palavra fornecida. Cada iteração conta com o resultado de uma aritmética baseada no id da thread em questão, e na quantidade total de threads para garantir que cada fluxo procure somente nos indexes designados, e não tenha o retrabalho de testar uma posição já testada por outro fluxo, mas ao mesmo tempo, não deixe nenhuma posição do vetor vacante.

`achou` é uma variável estado global, que serve de condição para evitar que as demais threads continuem procurando pela palavra que eventualmente já foi encontrada por algum outro fluxo. Levando em consideração que a natureza do problema é basicamente de leitura, optamos por não tratar da atomicidade da variável *achou*, pois no pior dos casos, após uma thread encontrar a palavra e alterar o valor da variável, as outras threads podem fazer no máximo mais um loop procurando pela palavra, o que nunca vai afetar o resultado final, e o custo computacional dessa operação adicional é insignificante.

3. **Casos de teste de corretude:**

Seção 1 - Validar a execução do jogo no nível 1.

i. Seção 1 - CT1 - Validar quantidade de letras das palavras.
Resultado esperado: As palavras selecionadas possuem a

quantidade de letras respectivamente com o nível selecionado, como descrito na documentação.

ii. Seção 1 - CT2 - Validar pontuação.
Resultado esperado: O programa registrará a pontuação e exibirá a mensagem “>>> Palavra Válida”.

iii. Seção 1 - CT3 - Validar a palavra digitada, com letra errada.
Resultado esperado: O programa retornará a mensagem “>>> Por favor, use somente letras disponiveis”.

iv. Seção 1 - CT4- Validar a palavra digitada com letra repetida.
Resultado esperado: O programa retornará a mensagem “>>> Por favor, use somente a quantidade de letras disponiveis”.

v. Seção 1 - CT5 - Validar a NÃO existência da palavra.
Resultado esperado: O programa exibe a mensagem “>>> Essa palavra não existe!”.

vi. Seção 1 - CT6 - Validar palavras menos de 3 caracteres.
Resultado esperado: O programa exibirá a mensagem “>>> Sua palavra eh muito pequena!”.

vii. Seção 1 - CT7 - Validar palavra repetida.
Resultado esperado: O programa exibirá a mensagem “>>> Você ja digitou essa palavra!”.

4. Avaliação de desempenho

Abaixo segue os cenários de testes referentes a análise de desempenho, para esses foi criado uma tabela que colherá o tempo sequencial e o tempo concorrente, Desta forma rodamos o programa de ambas as formas sequencial e concorrente por 5 ciclos de teste. Com isso podemos coletar os dados e encontrarmos: média sequência e concorrente do tempo gasto por cada caso de teste em cada ciclo, a partir disso a aceleração máxima média obtida. Vale também ressaltar os gráficos que ilustram as médias de tempos no fluxo concorrente, sequencial e também a aceleração máxima média obtida nos casos de teste levantados**.**

Cenários de teste:

Seção 2 -Validar desempenho do programa nos diferentes níveis com 1 threads em funcionamento.

I. **Seção 2 - CT1 - Validar busca da palavra montada no dicionário.**
Resultado esperado: O programa demora um determinado tempo para validar se a palavra existe no dicionário.

II. **Seção 2 - CT2 - Validar busca de palavra no inicio do dicionário.**
Resultado esperado: O programa demora um determinado tempo para validar se a palavra existe no dicionário.

III. **Seção 2 - CT3 - Validar palavra no meio do dicionário.**
Resultado esperado: O programa demora um determinado tempo para validar se a palavra existe no dicionário.

IV. **Seção 2 - CT4 - Validar Palavra no final do dicionário.**
Resultado esperado: O programa demora um determinado tempo para validar se a palavra existe no dicionário.

V. **Seção 2 - CT5 - Validar Palavra no final do dicionário.**
Resultado esperado: O programa demora um determinado tempo para validar se a palavra existe no dicionário.

OBSERVAÇÃO 1: Os testes foram realizados em um PC linux (xubuntu) municiado de um Intel core I5-5200U, 2 núcleos e 4 threads.

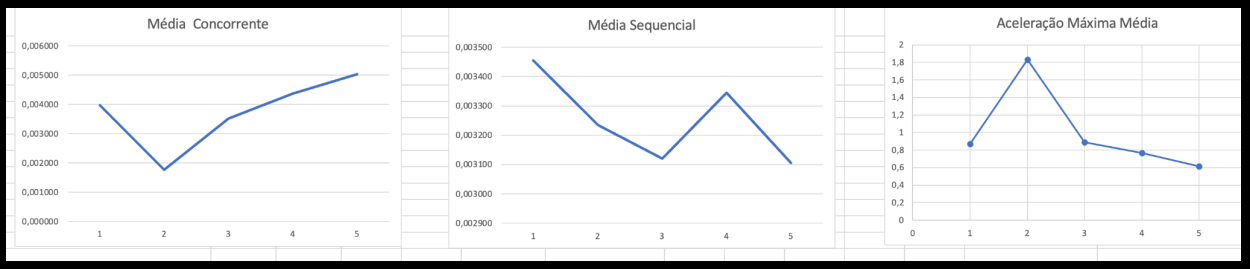
OBSERVAÇÃO 2: Para realizarmos os testes de desempenho, optamos por uma função de teste. Assim sendo na chamada do programa será passado como parâmetro o número 4, a partir disso entramos no fluxo de teste. Em seguida, a função de teste solicitará que seja digitada uma palavra e o número de threads que deseja usar, desta forma ambos os fluxos concorrentes e sequenciais vão buscar a palavra e imprimir o tempo gasto.

Vale observar também que por conta das demais tarefas sequências do programa depender muito da interação do usuário não há como prever tanto o tempo gasto pelo mesmo, e por conta disso fica inviável o cálculo de uma aceleração máxima teórica do jogo como um todo. Todavia por conta disso optamos por focar na busca ao dicionário e trazer a diferença entre o custo do fluxo sequencial e o fluxo concorrente e suas respectivas diferenças.

5. Discussão

Para realizar os testes de desempenho como descrito acima, colhemos amostras de tempo com fluxo sequencial e concorrente, sendo o concorrente com 1, 4 e 10 threads. Desta forma calculamos a média de tempo sequencial, concorrente e a aceleração máxima média. A partir disso foram gerados 3 gráficos dispostos da seguinte forma, no eixo x temos os 5 casos de teste para os 3 cenários (1, 4 e 10 threads), já no eixo y está representado o tempo médio gasto em cada cenário. Dito isso segue abaixo os gráficos e as análises:

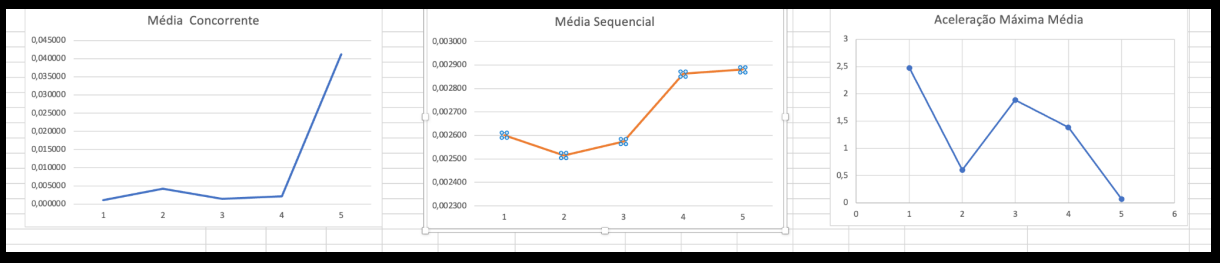
I. Cenário com 1 Thread



Fazendo análise do cenário de teste acima podemos concluir que: no caso de teste 1 (caso onde a palavra procurada se encontra em qualquer lugar aleatório do dicionário) o fluxo sequencial sai na frente; no segundo caso de teste (palavra no início) o fluxo concorrente dispara na frente e realiza a tarefa num tempo consideravelmente menor. Agora se formos analisar os demais casos de testes 3, 4 e 5 (palavra em posições intermediárias, perto do fim, e não existentes no dicionário) é notório que o fluxo sequencial realiza a tarefa num tempo bem menor do que o fluxo concorrente (dada a escala dos resultados).

Quanto à aceleração obtida, podemos observar que somente no CT2 houve de fato ganho de desempenho com a implementação da versão concorrente.

II. Cenário com 4 Threads

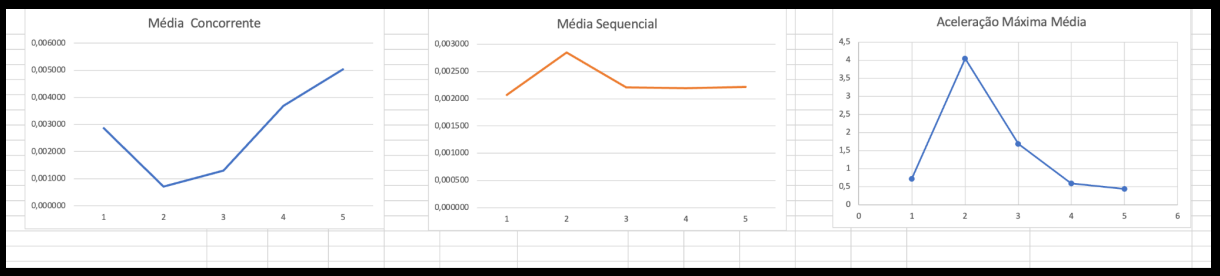


Fazendo análise do cenário de teste acima podemos concluir que: no caso de teste 1 (caso onde a palavra procurada se encontra em qualquer lugar aleatório do dicionário) o fluxo concorrente sai na frente; no segundo caso de teste (palavra no início) o fluxo sequencial tem um melhor resultado, dando continuidade nos casos de teste 3 e 4 (palavra em posições intermediárias e perto do fim) o fluxo

concorrente se destaca melhor. Por fim, no caso de teste 5 (palavra inexistente no dicionário), temos o destaque para o fluxo sequencial. Vale ressaltar que nesse cenário de teste com 4 threads o melhor tempo varia muito entre sequencial e concorrente.

Quanto à aceleração obtida, podemos observar que somente os CT`s 1, 3 e 4 houve de fato ganho de desempenho com a implementação da versão concorrente.

III. **Cenário com 10 Threads**



Fazendo análise do cenário de teste acima podemos concluir que: no caso de teste 1 (caso onde a palavra procurada se encontra em qualquer lugar aleatório do dicionário) o fluxo sequencial sai na frente; já no caso de teste 2 e 3 (palavra no início e palavra no meio do dicionário) o fluxo concorrente dispara na frente e realiza a tarefa num tempo consideravelmente menor. Agora se formos analisar os demais casos de testes 4 e 5 (palavra perto do fim, e não existentes no dicionário) é notório que o fluxo sequencial realiza a tarefa num tempo bem menor do que que o fluxo concorrente; com isso no terceiro gráfico podemos observar a aceleração máxima média deste cenário com 10 thread.

Quanto à aceleração obtida, podemos observar que somente os CT`s 2 e 3 houve de fato ganho de desempenho com a implementação da versão concorrente.

Conclusão

Realizando todas as análises de desempenho e colhendo todos os dados referentes aos fluxos concorrentes e sequenciais podemos concluir que em alguns casos, a versão concorrente ganha sim vantagem de desempenho sobre a versão sequencial, mas como na prática, esse ganho é praticamente imperceptível ao usuário, e casualmente existe até perda de desempenho, o esforço de elaborar e implementar uma versão concorrente é em vão, a não ser que o objetivo seja melhorar a organização do código através da modularização.

Sobre possíveis melhorias, poderíamos imaginar maneiras de otimizar a busca levando em consideração, por exemplo, que o nosso vetor dicionário está em ordem alfabética. Contudo, ao que tudo indica, projetar tal solução não traria um custo-benefício satisfatório para desempenho suficiente para valer a pena o esforço, uma vez que nem mesmo a solução concorrente em si se demonstrou necessária.

6. **Referências bibliográficas:**

   Como referências foram utilizados principalmente os conteúdos dados em aula, os slides e os laboratórios. Além disso, foi utilizado como base o nosso trabalho final de Computação 1.
   Repositório do projeto:: https://github.com/jotapecds/Anagrame/tree/refactor

   OBSERVAÇÃO: A documentação de teste se encontra na pasta ‘Documentos/Computação Concorrente’ do repositório do projeto.