# *Otimizando a importação de candidatos do SISU no processo de matrícula da UFRJ*

*Vitória Mendes Cortes Chaves e Luiz Carlos Ferreira Carvalho*

## Relatório Final

**Programação Concorrente (ICP-361) — 2024/1**

[1]

## 1. Descrição do problema

O serviço de importação de candidatos funciona a partir de uma requisição na qual o arquivo é carregado e após isso processado para inserção dos dados no banco de dados, e em seguida, os dados são processados em lotes, executados sequencialmente um por vez para a criação das entidades, nesta etapa ocorrem diversos passos como tratamento de dados pessoais e dados enem, onde precisa-se verificar se já existem dados antigos para um mesmo candidato, tornando o processo bastante demorado. A importação dos candidatos do SISU segue duas etapas:

1. Carregar um arquivo CSV fornecido pelo SISU (Sistema de Seleção Unificado) inserindo os dados em uma tabela auxiliar do banco de dados.
2. Converter os dados processados na etapa anterior nas entidades utilizadas pelo sistema para o processo de matrícula.

O principal benefício de uma solução concorrente nesse cenário é a otimização de tempo de processamento, visto que temos cerca de 9 mil candidatos por ano em um único arquivo, na qual os lotes são sujeitos as mesmas tarefas de processamento.

## 2. Projeto e implementação da solução concorrente

A solução escolhida visou apenas implementar concorrência nas etapas de conversão e persistência de dados, pois, se tornou evidente que utilizar o padrão produtor/consumidor não seria efetivo, já que o arquivo CSV com os dados do SISU é enviado inteiramente por meio de uma única requisição e logo é carregado por completo na memória RAM do servidor. Não houve eficácia na utilização do produtor/consumidor para carregar os dados em lotes do banco de dados, pois a query para recuperar todos de uma vez se tornou mais eficiente.

Antes de implementar a concorrência nas etapas de conversão e persistência, decidimos experimentar também na etapa de pre-processamento, que consiste em executar queries que carregam do banco dados de candidatos que participaram de processos anteriores e guardamos tais dados em um memória, esta etapa é mais custosa de todo o processo pois não temos como buscar estes dados a partir de ids, somente utilizando informações como numero enem e CPF, o que torna a performance da query bastante lenta pois precisamos procurar dados para cada um dos 8842 candidatos. Imaginamos que poderiamos separar essas queries em lotes e executa-las em paralelo utilizando threads, então incluimos também esta etapa no escopo do trabalho.

Para cada uma das etapas implementamos a concorrência da seguinte forma:

1. Pre-processamento: Criamos um ThreadPoolExecutor com número fixo de threads, e um CountDownLatch que funciona como uma barreira com contador N, uma vez que separamos o total de dados em N lotes, iniciamos o CountDownLatch com valor N, passamos então cada lote para o ThreadPoolExecutor, responsavel por gerenciar e alocar as tasks para as threads, quando um lote chega ao fim a task decrementa o CountDownLatch, quando o contador chega a 0 significa que o pre-processamento foi finalizado. Para que não ocorresse corrida de dados quandos as threads fossem escrever no cache implementamos um semaforo com valor 1, sendo assim somente uma thread pode escrever no cache por vez.
2. Conversão (Processamento): Implementamos o mesmo ThreadPoolExecutor com número fixo de threads porêm desta vez como a conversão de dados retorna os dados para persistência, decidimos utilizar o CompletableFuture, onde podemos criar uma lista de futuros que executa as tasks de forma assincrona e paralela nas threads do ThreadPoolExecutor retornando para a lista os resultados das conversões. Optamos por aguardar o final do processamento de cada CompletableFuture, onde cada um equivale a um lote, antes de inciar a persistência dos dados, para isso utilizamos o join do CompletableFuture.
3. Persistência: Reutilizamos o ThreadPoolExecutor anterior com a barreira com contador do CountDownLatch implementada da mesma forma que no pré - processamento, e então submetemos cada lote de dados para eprsistência arguardando o término para enfim terminar a execução desta última etapa.

Também utilizamos Collections.synchronizedMap() e Collections.synchronizedList() para garantir que nossas estrturas de dados seriam threadsafe.

## 3. Casos de teste

Para testar e avaliar o desempenho executamos o programa com 3 conjuntos de dados, o primeiro contendo mil entradas, o segundo 4 mil entradas e o terceiro os dados em sua totalidade com 8842 entradas, para verificar a corretude utilizamos as seguintes estratégias:

- Verificamos que todas as entidades esperadas foram criadas, por exemplo, se temos 8842 entidades quando rodamos um select count(1) esperamos como resultado 8842 entradas nas tabelas onde somente uma entidade deve ser criada.
- Após executar o código sequencial copiamos os dados para uma tabela auxiliar e então rodamos o programa de forma concorrente, para poder comparar os dados utilizando um select com except, onde caso não existam diferenças o resultado deve ser vazio.

Outra validação extra é proporcionada pelo próprio SQL Server já que por estarmos trabalhando com um banco relacional, ele possui constraints para garantir a integridade dos dados, sendo assim podemos suspeitar que o código executou como deveria em grande parte se ele chegar ao final sem nenhum erro de SQL.

Os resultados demonstraram que a solução concorrente manteve a consistência dos dados, pois em nossos testes sempre obtivemos o número desejado de entradas nas tabelas além do select execpt não retornar diferença entre as tabelas.

## 4. Avaliação de desempenho

Os conjuntos de casos de testes foram dimensionados em 1000, 4000 e 8842 linhas da chamada regular SISU de 2024.

- Os testes de desempenho foram realizados em uma máquina com processador AMD Ryzen 7 5700g com Nº de núcleos de CPU: 8, Nº de threads: 16 e Nº de núcleos de GPU: 8 e 16gb de memória RAM DDR4.
- Os casos de teste foram executados 5 vezes, e a média dos tempos estimados foi utilizada para calcular a aceleração e eficiência obtida, cuja as fórmulas seguem abaixo:

$$aceleração = \frac{T_{seq}}{T_{conc}}$$

$$eficiência = \frac{a}{p}$$

Onde o p corresponde ao número de processadores (ou núcleos) utilizados. No caso deste trabalho, utilizamos 16 núcleos lógicos. Abaixo, segue uma tabela com os resultados gerados:

| Tamanho | Threads | Pre-Processamento (s) | Processamento (s) | Persistência (s) |
|---|---|---|---|---|
| 1000 | 1 | 52 | 4 | 20 |
| | 4 | 50,6 | 1 | 6,16 |
| | 8 | 50,6 | 0,864 | 4,94 |
| | 16 | 50,8 | 0,542 | 3,4 |
| | | | | |
| 4000 | 1 | 208,2 | 15,76 | 80,45 |
| | 4 | 203 | 4,476 | 22,938 |
| | 8 | 202 | 2,4 | 14,128 |
| | 16 | 202 | 1,662 | 10,66 |
| | | | | |
| 8842 | 1 | 460,2 | 37 | 197,6 |
| | 4 | 447,6 | 8,8 | 53,6 |
| | 8 | 447,2 | 5 | 30,8 |
| | 16 | 447,4 | 3 | 22,6 |
| | | | | |

**Figure 1. Tempo sequencial vs Concorrente**

- Após a obtermos os tempos de execução sequenciais e concorrentes, com base nas fórmulas citadas acima, conseguimos os seguintes resultados:

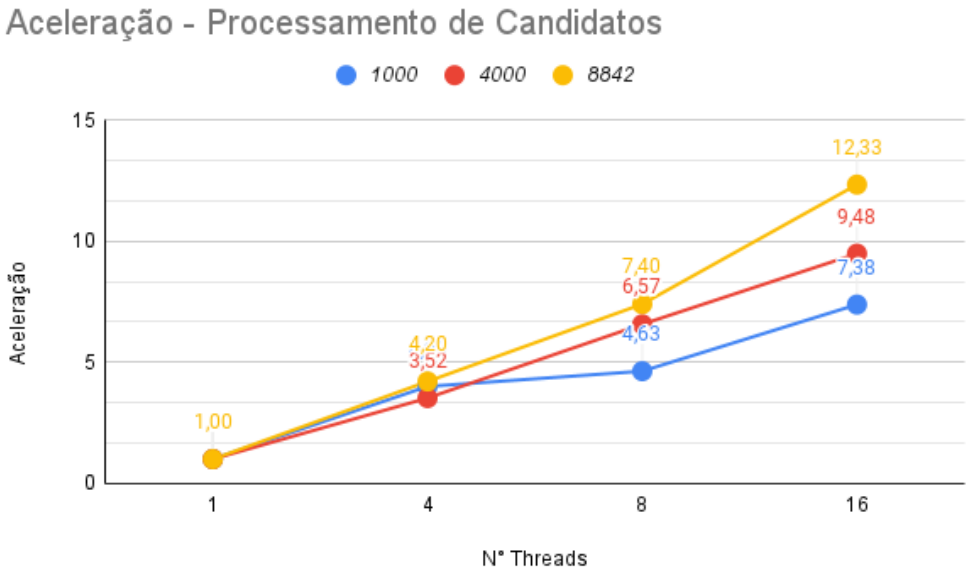


**Figure 2. Aceleração - Etapa de Processamento de Dados**

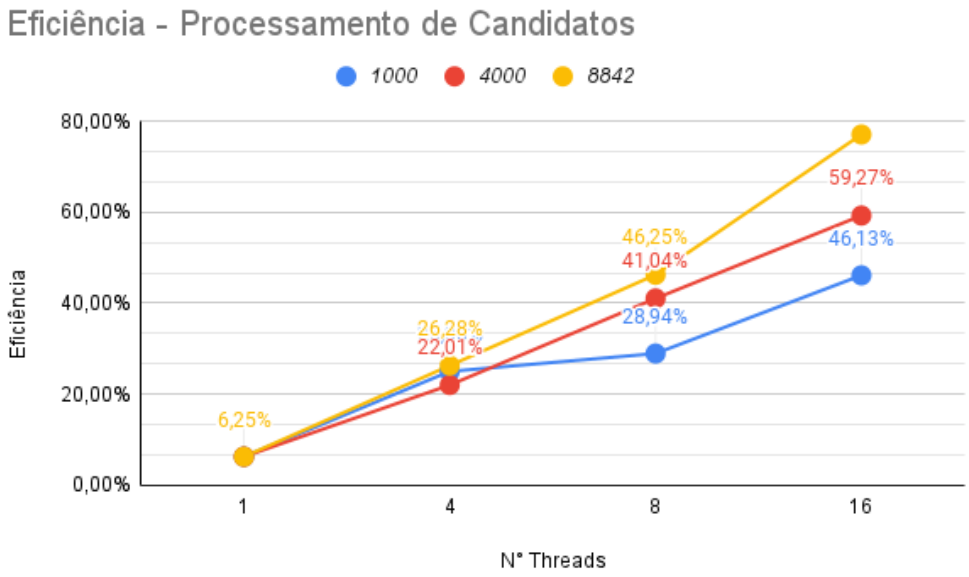


**Figure 3. Eficiência - Etapa de Processamento de Dados**

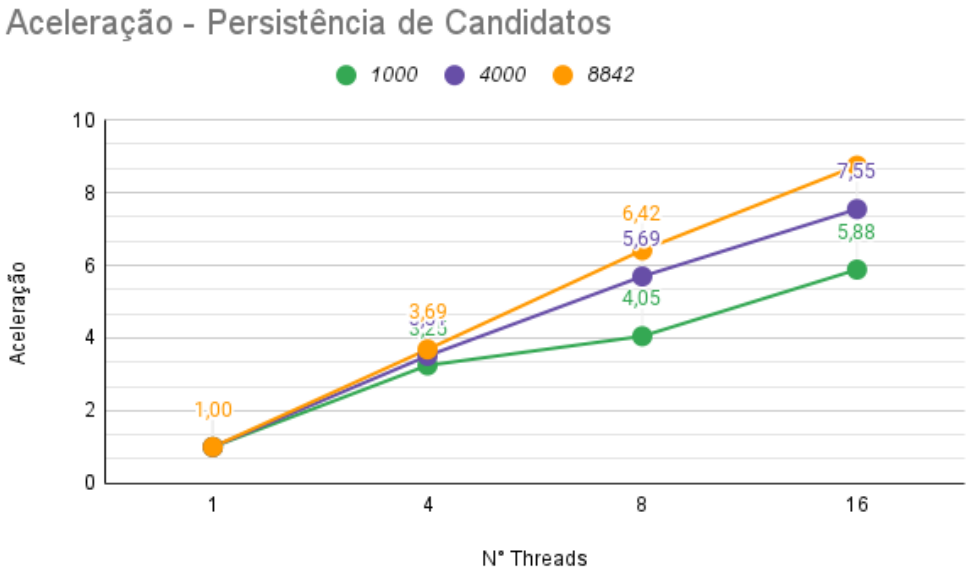


**Figure 4. Aceleração - Etapa de Persistência de Dados**

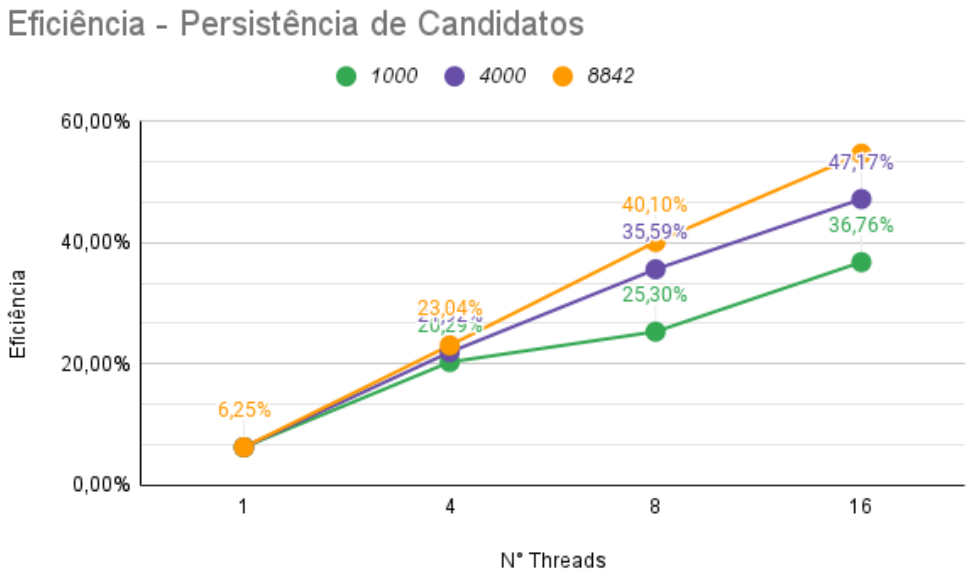


**Figure 5. Eficiência - Etapa de Persistência de Dados**

## 5. Discussão

Considerando os resultados obtidos, o desempenho nas etapas de conversão de dados e persistência estiveram dentro do esperado, já que alcançamos uma melhora substancial no tempo de execução saindo de 3 minutos para 20 segundos por exemplo na persistência de dados com 16 threads.

Porém o desempenho concorrente da etapa de pre-processamento não foi superior ao sequencial, na verdade ele apresentou um tempo praticamente igual, este caso nos gerou grande estranheza ainda mais considerando que as outras etapas tiveram ganho substancial, investigamos diversas causas, como gerenciamento de conexões com o banco, experimentamos até mesmo gerenciar as conexões de forma manual, porém não obtivemos sucesso em melhorar o tempo de execução o que nos levar a acreditar que o problema possa ser na forma que o banco de dados gerencia seus recursos nesse caso.

Nós também observamos que as threads executam em paralelo, e encontramos no banco de dados conexões equivalentes para cada threads, toda com tempo de execução incrementando com estado ”running” e muitas terminando quase que ao mesmo tempo, todos indicativos que de fato foram executadas em paralelo, porém acabaram tendo um performance individual inferior a sua execução sequencial.

Esta etapa foi a maior dificuldade do trabalho, pois dedicamos bastante tempo a ela investigando as possiveis causas tentando entender o que estaria acontecendo para não termos ganho algum em relação a implementação concorrente.

No final conseguimos melhorar a execução do programa, porém talvez exista alguma forma de melhorar a etapa de pre-processsamento que nós não conseguimos identificar totalmente a causa, e que talvez possa apresentar uma maior ganho no futuro.

## 6. Referências bibliográficas


- An Introduction to Parallel Programming 1ª Edição, Pacheco Peter.
- Java Concurrency - Baeldung
- SQL Except - SQL Server
- Batch Insert/Update with Hibernate/JPA
- An Introduction to Synchronized Java Collections
- Aceleração e Eficiência geradas