UFRN
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Trabalho Prático: *Java Concurrent Utilities*

Giovanne da Silva Santos
Noé Fernandes Carvalho Pessoa

28 de julho de 2020

1 Introdução

No presente relatório, apresentaremos nossa solução de como foi calculado o número de euler, mediante uma série infinita. utilizando os três tipos de thread pools fornecidos pela classe Executors: fixed thread pool, cached thread pool e work stealing thread pool. Nas seguintes seções, explicaremos como cada solução foi atingida, como as tarefas foram distribuidas entre as threads, como foi a performance em termos de tempo de execução e quantidade de threads utilizadas, entre outros resultados relevantes.

2 Metodologia

Na nossa solução, primeiro criamos a classe da tarefa a ser executada chamada *Serie_calculator*. A *Serie_calculator* será responsável por calcular um termo da série. Na classe, optamos por implementar a interface *callable* para o tipo específico Double, pois na tarefa há a necessidade de retornar o valor do termo calculado, o que não seria tão simples usando a interface *Runnable*, que retorna void. As seguintes soluções para o problema usando diferentes thread pools foram separadas em diferentes classes "main's". Como solicitado, em todas um valor para a quantidade de termos foi especificado(testando para 4, 8 , 16, 32 e 50 termos), colocado como uma variável global. Vale salientar que para o método *FizedThreadPool*, foi estabelecido o uso de 20 threads. Sendo que o número de threads foi obtido através do método activeCount da classe Thread.

Foram aplicados 30 testes em cada um dos métodos, tirando a média aritmética dos resultados obtidos para análise. O ambiente de testes foi em um notebook Lenovo com processador IntelR©CoreTMi7-7500U CPU @2.70GHz 4, memoria RAM de 3,6 GB e HD de 500 GB e sendo o sistema operacional o Ubuntu 18.0.

2.1 *Fixed Thread Pool*

Para a solução fixed, primeiro colocamos em uma variável global o número de threads que vão executar a tarefa concorrentemente. Logo após, criamos o objeto *Executor*, especializado para *fixed thread pool*, fornecendo o número de threads para o construtor. Criamos também um array de *Futures*, que será usado para armazenar o resultado das operações realizadas por cada thread sobre cada termo de forma que a atribuição só ocorra após a execução da thread.

O próximo passo foi, em um loop sobre a quantidade de termos, instanciar o objeto *Serie_calculator* e realizar o *submit* forneceno-o como parâmetro e recebendo o resultado em uma variável *Future*. Imediatamente, adicionamos o resultado obtido pela tarefa ao array de *Futures*. Os últimos passos são percorrer o array de *Futures* afim de obter a soma dos termos, mostrá-los na tela e solicitar o encerramento com o *shutdown*.

2.2 Cached Thread Pool

Na solução cached a estrutura do código da solução anterior foi pouco alterada. As únicas alterações foram: a retirada da variável global especificando o número de threads - já que a *Cached Thread Pool* por si só já as cria dinamicamente - e a mudança no tipo do objeto *Executor* atribuído.

2.3 Work Stealing Thread Pool

Na *Work Stealing Thread Pool* se passa o mesmo que na solução anterior, não houve necessidade de muitas mudanças. As únicas mudanças significativas foram exatamente as que ocorreram na *Cached Thread Pool* com relação à *Fixed Thread Pool*, lembrando também que a solução *Work Stealing* também possui a característica dinâmica ao lidar com as threads.

3 Resultados

Nas tabelas abaixo, teremos os resultados obtidos com cada método em relação ao número de threads e tempo de execução. Sendo que tabelas para 4, 8, 16, 32 e 50 termos respectivamente:

- Para 4 termos:

Método	**Tempo médio(ms)**	**Nº de threads**	**Resultado encontrado**
Fixed	8	5	2.6666666666666665
WorkStealing	11	5	2.6666666666666665
Cached	11	4	2.6666666666666665

- Para 8 termos:

Método	**Tempo médio(ms)**	**Nº de threads**	**Resultado encontrado**
Fixed	11	10	2.7182539682539684
WorkStealing	8	5	2.7182539682539684
Cached	9	4	2.7182539682539684

- Para 16 termos:

Método	**Tempo médio(ms)**	**Nº de threads**	**Resultado encontrado**
Fixed	20	12	2.718281828458995
WorkStealing	10	5	2.718281828458995
Cached	11	5	2.718281828458995

- Para 32 termos:

Método	**Tempo médio(ms)**	**Nº de threads**	**Resultado encontrado**
Fixed	59	19	2.7182818284590455
WorkStealing	10	5	2.7182818284590455
Cached	54	7	2.7182818284590455

- Para 50 termos:

Método	**Tempo médio(ms)**	**Nº de threads**	**Resultado encontrado**
Fixed	31	21	2.7182818284590455
WorkStealing	10	6	2.7182818284590455
Cached	21	8	2.7182818284590455

Como podemos perceber, para quantidade menores de termos(4 e 8 termos), e consequentemente quantidade menores de tarefas, o desempenho dos métodos foram muito próximos com algumas pequenas variações. Contudo, para 16 termos já vemos uma performance inferior no método fixed em relação ao tempo de execução, assim como a quantidade de threads utilizadas ser maior também. Já na maiores quantidade de tarefas, 32 e 50 termos, temos uma superioridade do método WorkSteling em relação à demais, pois o tempo médio de execução é de 10 milissegundos com o uso de 5 threads(para 32 termos) e 6 threads(50 termos) em detrimento ao tempo de 59 e 54 milissegundos pela Fixed e Cached para 32 termos. Na última tabela temos uma diminuição no tempo de execução pela Fixed e Cached, porém foi aumentado o número de threads, sendo que na Fixed foi passado o número de threads pedidos para ser instanciado.

4 Conclusão

Com isso, é notório a superioridade que o método de thread pool WorkingStealing tem sobre as demais por conta de ter uma menor tempo de execução e utilizar menos threads. Além, fica claro que o fixed é mais custoso por não exercer reaproveitamento das threads e ainda sim traz um rendimento inferior.

Para a execução do programa, basta utlizar uma IDLE de java e dar ”run”em cada uma das ”main’s”aos respectivos métodos. Vale salientar que em cada main será gerarado um .txt para os resultados de tempo, número de threads e resultado da série infinita.