Resenha sobre Programação Paralela
Threads em Java

Sungwon Yoon,[1] Matheus P. C. Santos,[2] Ryan B. Ramos,[3] Wendel F. Lana[4]

Escola de Artes, Ciências e Humanidades - Universidade de São Paulo

sungwon.yoon@usp.br,[1] matheuspecoraro@usp.br,[2] ryanramos@usp.br,[3] wendel.lana@usp.br[4]

Resumo

Este trabalho tem por objetivo apresentar a linguagem Java e seu suporte para a programação paralela, bem como comparar o desempenho da execução sequencial e paralela em funções Java. Em computação, paralelismo é a execução simultânea de instruções. Dessa forma pode ocorrer, por exemplo, ao executar duas linhas de instruções de programas diferentes em dois processadores diferentes do mesmo computador, ou ao executar duas partes de um programa em computadores diferentes. Esses segmentos de execução de um programa são chamados de threads. A máquina virtual Java permite que um programa possa ter vários threads em execução simultânea e é a primeira linguagem de programação a incluir o conceito de threads na própria linguagem. Dessa forma, cada processo possui diversos threads, assim nós podemos dividir partes do programa em Java para trabalhar paralelamente. A implementação de threads em Java, principalmente em sua versão mais recente, é bem simples e direta. Na implementação de threads são compartilhados entre si o código, os dados e os ponteiros de arquivos. Dessa maneira, uma variável alocada na área de dados pode ser acessada por qualquer uma das threads, inclusive de forma concorrente. A programação paralela, ou computação paralela, é uma ótima prática de desenvolvimento que é bastante utilizada, principalmente, por seu desempenho no tempo de processamento de programas, permitindo múltiplas linhas de execução.

Programação Paralela

A programação paralela é o processamento de instruções do programa, dividindo-as entre vários núcleos com o objetivo de executá-las simultaneamente e em menos tempo como ilustrado na Figura 1. Dessa forma, é possível executar diferentes tarefas simultaneamente utilizando formas primitivas de comunicação e sincronização. Esses segmentos de execução de um programa são chamados de threads.

A máquina virtual Java (JVM) permite que um aplicativo possa ter vários threads em execução simultânea. Portanto, cada processo possui diversos threads (ou linhas de instruções), assim nós podemos dividir partes do programa em Java para trabalhar paralelamente de forma assíncrona ou síncrona.

Linguagem de Programação Java

Para implementação da programação paralela escolhemos a linguagem Java, pois desde o seu início foi projetada para dar suporte ao desenvolvimento de aplicações multitarefa e a facilitar tais implementações. Sendo inclusive a primeira linguagem de programação a incluir o conceito de threads na própria linguagem.

Além disso, com o crescimento da linguagem, novas características foram adicionadas, como a inclusão do framework Executor na versão 5 que passou a cuidar da criação, gerência e finalização das threads. Posteriormente no Java 7, houve o lançamento do framework Fork-Join que ajuda a simplificar ainda mais a escrita de código de execução paralela, permitindo o aproveitamento de todos os recursos oferecidos por múltiplos processadores.

Em sua versão mais recente, o Java 8, com os novos recursos de programação funcional, como as expressões lambda, a linguagem permite a produção de código paralelo, limpo e com qualidade. Se tornando uma das primeiras linguagens a padronizar utilitários e APIs de alto nível para lidar com threads.

Threads em Java

Na programação Java, threads são instâncias da classe Thread que implementam com o método run da interface Runnable os procedimentos que devem executar paralelamente. Como demonstrado na Figura 2, a implementação de threads em Java, principalmente em sua versão mais recente, é bem simples e direta.

Na implementação de threads são compartilhados entre si o código, os dados e os ponteiros de arquivos. Dessa forma, uma variável alocada na área de dados pode ser acessada por qualquer uma das threads, inclusive de forma concorrente. Porém, cada linha de execução (ou thread) possui seus próprios registradores e sua pilha de funções.

Comparação da Programação Paralela e Sequencial

Quando a máquina rodando o processo possui apenas um processador, essas threads são escalonadas para que apenas uma de cada vez faça uso do processador. No entanto, com mais processadores, podemos ter uma execução completamente paralela onde cada thread é escalonada em um processador diferente assim como ilustrado na Figura 1.

Portanto, é perceptível a vantagem da utilização da programação paralela, principalmente, utilizando mais de um processador. Porém, para compararmos o desempenho com a programação sequencial, criamos 2 funções de execução paralela e outra sequencial que realizam um loop imprimindo um contador na tela até o número 4, além de mostrar a quantidade de milissegundos gastos para sua execução. O código pode ser observado na Figura 3.

Pela Figura 4 podemos perceber a diferença no tempo de execução entre a função sequencial e paralela. É importante ressaltar que mesmo em computadores com apenas um processador ou com um núcleo, a função utilizando paralelismo é executado mais de forma mais rápida que a função sequencial.

Seções Críticas na Execução Paralela

Quando mais de uma thread tenta acessar uma variável ao mesmo tempo pode ocorrer um problema de disputa por esse recurso, denominamos esse evento como condição de corrida. O impacto dessa condição é que o resultado da execução dessas threads depende de qual thread foi primeiro processada, dessa forma não há garantias de que o resultado será o mesmo independentemente do número de execuções.

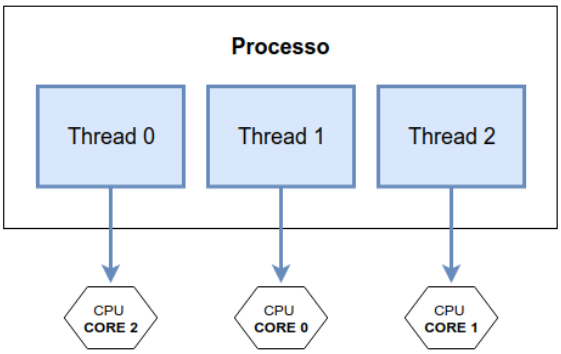
Os trechos do código em que a condição de corrida pode ocorrer são chamados de seções críticas. Uma forma de solucionar o problema é com o mutex, ou exclusão mútua, em que garantimos que apenas uma thread esteja executando sua seção crítica por vez. Porém, pode ocorrer das threads que aguardam a execução da seção crítica entrem em um loop de espera por aquele recurso, o que chamamos de espera ocupada.

Existem muitas soluções para a condição de corrida, uma bem conhecida é o semáforo binário que também implementa a exclusão mútua, mas não ocorre espera ocupada, ou seja, não ocorre uma espera interminável. Nessa implementação, cada thread quando tentar executar sua seção crítica irá verificar antes se o semáforo permite o acesso ao recurso. Dessa forma, quando não está disponível, a thread fica em um estado de espera e quando o recurso é liberado o sistema operacional sinaliza, através de eventos, para que o processo seja executado.

Conclusão

Como demonstrado, é notório a redução do tempo de processamento do programa utilizando o paralelismo, uma vez que as threads são recursos que permitem um mesmo programa ter várias linhas de execução. Contudo, diversos problemas devem ser levados em consideração como condição de corrida, seção crítica e espera ocupada.

Portanto, a programação paralela deve ser sempre utilizada com bastante atenção, pois ao manipular dados compartilhados entre threads pode acarretar em alguns cenários de erros. Devemos também levar em conta as soluções como o mutex e o semáforo que permitem resolver esses problemas, já que o Java fornece uma base sólida para a criação de uma ampla variedade de soluções paralelas.



O paralelismo sem dúvidas é uma ótima prática, que está se dispersando por todos os softwares do mercado, tendo em vista que atualmente qualquer computador, até mesmo celular tem mais de um núcleo de processamento.

Figura 1. Divisão de tarefas entre as CPUs

```
new Thread(() -> {
    //Código da Tarefa
}).start();
```

Figura 2. Código Java 8 para criação de thread utilizando expressões lambda

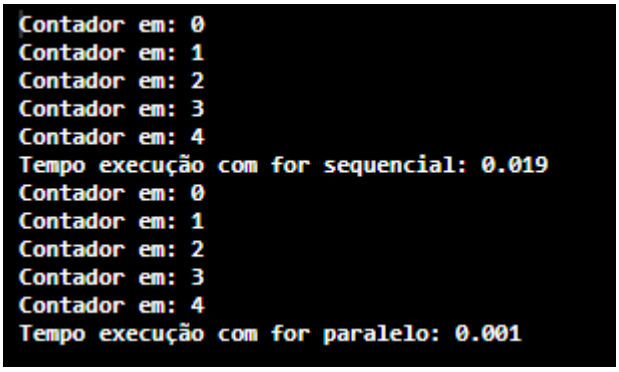


Figura 3. Código Java com funções de programação paralela e sequencial

```java
public static void executaParalelo(){
    new Thread(() -> {
        long start = System.currentTimeMillis();
        for(int i=0; i<5; i++){
            System.out.println("Contador em: " + i);
        }
        long elapsedTime = System.currentTimeMillis()-start;
        float elapsedTimeSec = elapsedTime/1000F;
        System.out.println("Tempo execução com for paralelo: "+ elapsedTimeSec);
    }).start();
}

public static void executaSequencial(){
    long start = System.currentTimeMillis();
    for(int i=0; i<5; i++){
        System.out.println("Contador em: " + i);
    }
    long elapsedTime = System.currentTimeMillis()-start;
    float elapsedTimeSec = elapsedTime/1000F;
    System.out.println("Tempo execução com for sequencial: "+ elapsedTimeSec);
}
```

Figura 4. Resultado da execução das funções paralela e sequencial

Referências

Programação Paralela. https://www.devmedia.com.br/programacao-paralela/21405. Acesso em 27/12/2021

Concorrência, Paralelismo, Processos, Threads, Programação Síncrona e Assíncrona. https://www.treinaweb.com.br/blog/concorrencia-paralelismo-processos-threads-programacao-sincrona-e-assincrona. Acesso em 29/12/2021.

Programação Paralela com Java. https://www.devmedia.com.br/programacao-paralela-com-java/33062. Acesso em 30/12/2021

Trabalhando com Threads em Java. https://www.devmedia.com.br/trabalhando-com-threads-em-java/28780. Acesso em 02/01/2022

Threads: paralelizando tarefas com os diferentes recursos do Java. https://www.devmedia.com.br/threads-paralelizando-tarefas-com-os-diferentes-recursos-do-java/34309. Acesso em 03/01/2022