MAC5742 - Introdução à programação paralela e distribuída

1º semestre de 2015

Uso de programação funcional em paralelismo

Suzana de Siqueira Santos - 6909971

28 de junho de 2015

Sumário

1 Introdução

Atualmente, o paralelismo tem sido a chave do ganho de desempenho de diversas aplicações, uma vez que existem fatores que limitam o aumento da velocidade dos processadores (1). Contudo explorar o paralelismo ainda é um grande desafio. Para escrever um programa em paralelo, é preciso particionar o programa em tatefas, mapear tarefas em uma máquina paralela (possivelmente dinamicamente), e, sobretudo, garantir a corretude do resultado final.

Uma das dificuldades de programação paralela é a natureza assíncrona de muitas máquinas paralelas, que são usualmente programadas com linguagens não-determinísticas (2). Linguagens paralelas determinísticas podem ser fundamentadas da mesma forma que as linguagens sequenciais. Isso porque um programa paralelo determinístico devolve o mesmo resultado independentemente da ordem de escalonamento das tarefas. O mesmo não vale para linguagens não-determinísticas, pois ordens de execução diferentes podem gerar resultados diferentes. Em linguagens não-determinísticas, o programador deve, além de garantir que o resultado de qualquer execução será correto, prevenir o *deadlock*.

Testar um programa paralelo é mais desafiador do que testar um programa sequencial. O *deadlock* pode não ser detectado em testes, uma vez que ele geralmente ocorre apenas ocasionalmente, mesmo que sejam utilizados os mesmos dados de entrada. Além disso, como a execução de um programa paralelo não pode ser replicada, depurar esse tipo de programa se torna bastante difícil. A programação funcional vem sido apontada como uma possível solução para algumas das dificuldades encontradas em paralelismo.

Linguagens funcionais são linguagens de programação que expressam computação em termos de funções. Essas linguagens são, em muitos aspectos, diferentes das linguagens imperativas, e apresentam diversas vantagens em relação às linguagens convencionais.

Algumas das principais vantagens das linguagens funcionais são o alto nível de abstração de programação e a ausência de efeitos colaterais, que conforme discutiremos neste trabalho, representam uma grande facilidade em programação paralela.

2 Programação funcional

Neste capítulo trataremos de conceitos básicos de programação funcioanal por meio de uma adaptação do artigo da Wikipedia (3) e da documentação oficial da linguagem Haskell (4) sobre programação funcional.

Em programação funcional, a execução de um programa corresponde à avaliação de expressões, em oposição às linguagens imperativas, em que os programas contêm comandos que podem mudar o estado global quando executados. Programação funcional usualmente evita mudanças de estado.

A programação funcional requer que funções sejam de *primeira classe*, isto é, que elas sejam tratadas como um valor qualquer, podendo ser passadas como argumento para outras funções ou devolvidas como resultado de uma função. Ser de primeira classe também significa que é possível definir e manipular funções dentro de outras funções. Quando funções referenciam variáveis locais dentro de seu escopo, se a função é utilizada em outro escopo, as variáveis locais do bloco em que a função foi definida ficam retidas na memória. Assim, em geral, é frequentemente difícil determinar estaticamente quando essses recursos podem ser liberados e, então, é preciso utilizar gerenciamento automático da memória.

2.1 Conceitos

Existem diversos conceitos e paradigmas que são específicos de programação funcional. Há, contudo, algumas linguagens híbridas que utilizam diferentes paradigmas e, então, alguns desses conceitos também são utilizados por linguagens que não são puramente funcionais.

2.1.1 Funções de primeira classe e de ordem superior

Funções de ordem superior são funções que podem receber outras funções como argumentos ou retorná-las como resultado. Essas funções estão fortemente relacionadas com funções de primeira classe, pois tanto funções de primeira primeira classe quanto de ordem superior permitem que funções sejam recebidas como argumento e devolvidas como resultado. A distinção entre esses dois conceitos é sutil: "ordem superior" descreve um conceito matemático de funções que operam sobre outras funções. Enquanto "primeita classe" é um termo de ciência da computação que descreve entidades de linguagem de programação que não têm restrições de uso (assim, funções de primeira classe podem aparecer nos mesmos lugares do programa que quaisquer outras entidades de primeira classe, como os números).

Funções de ordem superior permitem fazer aplicações parciais, uma técnica em que uma função é aplicada aos seus argumentos um a um e cada aplicação devolve uma nova função que aceita o próximo argumento.

2.1.2 Funções puras

Funções (ou expressões) puramente funcionais não têm efeitos colaterais (de memória ou de entrada/saída). Isso significa que funções puras têm várias propriedades úteis que oferecem facilidades como:

- Se o resultado de uma expressão pura não é utilizada, ela pode ser retirada sem afetar outras expressões.
- Se uma função pura é chamada com argumentos que não provocam efeitos colaterais, o resultado é contante em relação àquela lista de argumentos, isto é, se a função puramente funcional é chamada novamente com os mesmos argumentos, o mesmo resultado será devolvido (o que permite otimizar o código, fazendo, por exemplo, memoização). Essa característica é conhecida como *transparência referencial*.
- Se não existe dependência nos dados entre duas expressões puras, então a ordem de avaliação pode ser revertida, ou elas podem ser executadas em paralelo sem que uma interfira na outra (ou seja, a avaliação de quaisquer expressões é *thread-safe*).
- Se a linguagem não permite efeitos colaterais, então o compilador pode ter a liberdade para reordenar a avaliação das expressões em um programa.

2.1.3 Recursão

Recursão é amplamente utilizada em programação funcional como a forma canônica e, frequentemente, a única forma de iterar. Muitas linguagens de programação incluem otimizações para recursão em cauda a fim de economizar memória.

2.1.4 Avaliação preguiçosa

Como computações puras são referencialmente transparentes, elas podem ser executadas a qualquer momento e ainda obter o mesmo resultado. Isso faz com que seja possível postergar a computação de valores até que eles sejam necessáros, isto é, computá-los preguiçosamente. Avaliações preguiçosas evitam computações desnecessárias e permitem, por exemplo, as chamadas estruturas de dados infinitas, que não poderiam ser representadas em linguagens que não fazem avaliação preguiçosa, pois não haveria espaço suficiente para guardá-las.

2.1.5 Eficiência

Linguagens de programação funcional são tipicamente menos eficientes no uso da CPU e da memória do que linguagens imperativas como C e Pascal. Isso está relacionado ao fato de que algumas estruturas de dados mutáveis como vetores tem uma implementação direta usando hardware Contudo, imutabilidade dos dados também pode, em muitos casos, proporcionar eficiência permitindo que o compilador faça suposições que não seriam seguras em uma linguagem imperativa.

A avaliação preguiçosa também pode acelerar o desempenho do programa, mesmo assintoticamente. Contudo, a maioria das implementações atuais de avaliação preguiçosa não apresenta uma boa performance.

3 Programação paralela e concorrente

Neste capítulo, resumiremos os problemas clássicos enfrentados em programação paralela e concorrente. Para isso, fizemos adaptações de trechos do capítulo de livro escrito por Kevin Hammond (5), da Universidade de St. Andrews, e de um artigo de Simon Marlow (trabalhou na Microsoft Research, e trabalha atualmente no Facebook) (6) publicado no blog do GHC (https://ghcmutterings.wordpress.com/).

Como vimos, programas paralelos envolvem muitos desafios que não estão presentes no código sequencial. Alguns dos principais desafios são:

1. **Decomposição**: é preciso projetar o código de forma que possamos dividir uma determinada tarefa em subtarefas a serem executadas em paralelo.

2. **Race conditions**: em alguns casos, a ordem em que cada expressão é avaliada e/ou a ordem de comunicação podem afetar o resultado do programa paralelo.

3. **Locking**: a memória compartilhada deve ser bloqueada para evitar conflitos que possam provocar resultados incunsistentes; esse bloqueio é geralmente caro e sujeito a falhas.

4. **Deadlock/Livelock**: o programador deve evitar criar dependências que possam bloquear a execução do programa.

5. **Granularidade**: é necessário atingir o nível adequado de granularidade.

6. **Escalabilidade**: programas deveriam tirar vantagem do aumento do número de processadores.

7. **Balanceamento de carga**: pode ser que o trabalho fique distribuído desigualmente entre os recursos de processamento disponíveis, especialmente quando as tarefas tem uma glanularidade irregular. Algumas vezes pode ser necessário rebalancear a alocação de tarefas.

3.1 Paralelismo *versus* concorrência

No estudo de linguagens de programação é fundamental diferenciar paralelismo de concorrência. Um programa *concorrente* é aquele que possui múltiplas threads de controle. Cada thread tem efeito no mundo e essas threads sao intercaladas por um escalonador. Nós dizemos que uma linguagem de programação concorrente é *não-determinística*, porque o resultado final

do programa pode depender da ordem do escalonamento. O programador deve então controlar esse não-determinismo por meio de sincronização, para, assim, se certificar de que o programa terminará com o resultado desejado independentemente da ordem do escalonamento. A tarefa de garantir que o resultado será correto é "traiçoeira" uma vez que não há nenhuma forma sistemática de verificar se todos os possíveis escalonamentos foram cobertos.

Por outro lado, um programa *paralelo* é aquele que é simplesmente executado em vários processadores com o objetivo de melhorar a performance em relação à versão sequencial. Uma vez que definimos programas paralelos e programas concorrentes, podemos nos perguntar por que é necessário diferenciá-los, se, usualmente, paralelismo envolve concorrência.

De fato, em linguagens com efeitos-colaterais, a cada vez que é preciso executar mais de uma coisa ao mesmo tempo, podemos ter o não-determinismo quando os efeitos de cada operação são intercalados. Assim, nesse tipo de linguagem, trabalhar com paralelismo requer tratar concorrência.

Contudo, em liguagens sem efeitos colaterais, como as linguagens puramente funcionais, é possível executar diferentes partes do programa simultaneamente sem observar nenhuma diferença no resultado final.

Apesar do não-determinismo ser indesejável na maior parte das vezes, utilizar concorrência é muito útil para estruturar um programa que precisa se comunicar com diversos clientes externos simultaneamente ou responder a múltiplas entradas assíncronas. Concorrência é ideal, por exemplo, para interfaces gráficas que precisam responder à entrada do usuário enquanto executa outras tarefas. Programação concorrente permite que o programa seja estruturado como se cada comunicação inidividual fosse uma tarefa sequencial, ou uma thread, e, nesses casos, a concorrência é frequentemente a abstração ideal.

4 Paralelismo com linguagens funcionais

Neste capítulo, discutiremos paralelismo em linguagens funcionais com base no artigo de Loidl (2003) (7) e na tese de Roe (1991) (2).

Programação paralela é inerentemente mais difícil que programação sequencial. Tradicionalmente o programador não deve se preocupar apenas em descrever um algoritmo correto, mas também em como organizar as subtarefas na arquitetura em que a aplicação será executada. As linguagens funcionais contemporâneas têm três propriedades chaves que as tornam atrativas para programação paralela: elas possuem mecanismos poderosos para abstrair ambas computação e coordenação de processos; elas eliminam dependências desnecessárias (não têm mudança de estado); e elas permitem um estilo de programação independente da arquitetura da máquina.

4.1 Propriedades que facilitam o paralelismo

4.1.1 Abstração

Linguagens funcionais têm excelentes mecanismos de abstração que podem ser aplicados em ambas computação e coordenação de processoa. Dois importantes mecanismos de abstração são a composição de funcções e as funcções de ordem superior. Funções de ordem-superior manipulam outras funções, permitindo que novas tarefas sejam definidas à medida que elas são requesitadas.

O princípio da abstração pode ser realizado para toda a programação paralela, onde as funções de ordem superior formam a base da programação. Tipicamente, programas funcionais paralelos irão abstrair detalhes como a localização do processo, o tempo e o tamanho da comunicação e as questtões de sincronização. Assim, mais esforços podem ser dedicados ao algoritmo paralelo em si. Maiores níveis de abstração encorajam experimentar paralelizações alternativas, o que frequentemente leva a soluções novas ou melhores para problemas paralelos.

4.1.2 Eliminação de dependêncas desnecessárias

A ausência de efeitos colaterais faz com que seja razoavelmente direto identificar um potencial paralelismo. Assim, dependências sequenciais acidentais não são introduzidas dentro do código fonte. A única fonte de dependência sequencial está nos argumentos de uma função que precisam ser avaliados antes de serem usados. Isto é, as dependências são identificadas somente quando os dados forem utilizados. Como os valores não mudam (em funções puramente

funcionais) uma vez que eles tenham sudo computados, não é necessário analisar o fluxo de dados para verificar o resultado da computação, mesmo no nível de inter-processos.

4.1.3 Provas formais de corretude

A garantia de que o resultado de uma computação paralela será igual ao da versão sequencial pode ser obtida facilmente com programação funcional pelo fato de um programa funcional poder ser descrito em termos de funções matemáticas (cálculo lambda).

Por exemplo, o Teorema de Church-Rosser garante que um programa funcional executado em paralelo devolverá o mesmo resultado final independentemenre da ordem em que as subtarefas forem executadas. Isto é, um programa funcional paralelo é determinístico (em oposição a um programa paralelo escrito em linguagens imperativa, que é não-determinístico).

4.1.4 Prevenção de deadlock

Deadlock ocorre apenas quando o programa (na versão sequencial) tem resultado indefinido. Para compreendermos como essa propriedade é garantida pela programação funcional, considere a seguinte expressão, escrita em pseudocódigo:

$$res = x + y \tag{4.1}$$
$$\text{onde} \tag{4.2}$$
$$x = 1 + 2 \tag{4.3}$$
$$y = x + 3 \tag{4.4}$$

Podemos representar a expressão (4.1) por um grafo, em que o símbolo @ indica as chamadas de funções, as subárvores mais à esquerda mostram a função chamada (neste exemplo, a função soma +) e as subárvores mais à direita mostram os argumentos passados para a função. Esse grafo é refresentado na Figura 4.1.

Cada execução em paralelo pode ser representada por uma ou mais avaliações de expressões que correspondem a reduções do grafo represetando uma expressão. A operação da redução da expressão (4.3) é representada pela Figura 4.2.

Um deadlock pode ser interpretado como um ciclo em um grafo que representa um programa. Contudo, isto só irá acontecer quando um valor depender dele mesmo, por exemplo, se tivermos uma expressão do tipo $a = a + 1$. Assim, em programação funcional, o deadlock é mais facilmente controlável pelo programador, uma vez que este só irá ocorrer quando o programa tiver um resultado indefinido mesmo na versão sequencial.

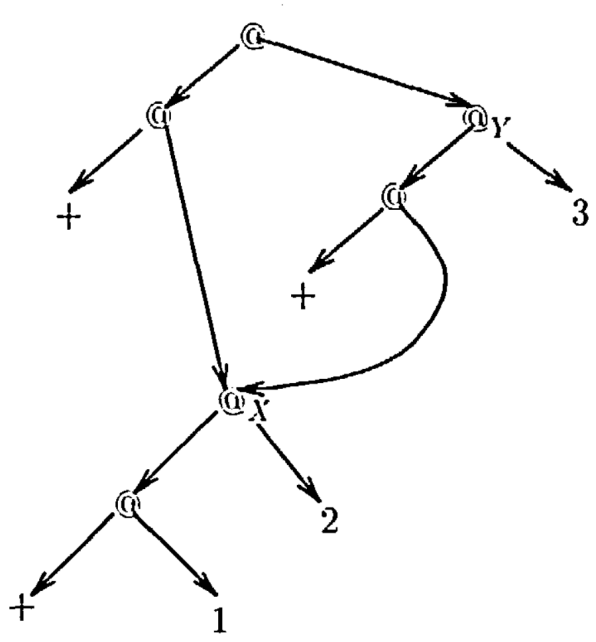

Figura 4.1: *Grafo representando a expressão (4.1). Imagem retirada de (2).*

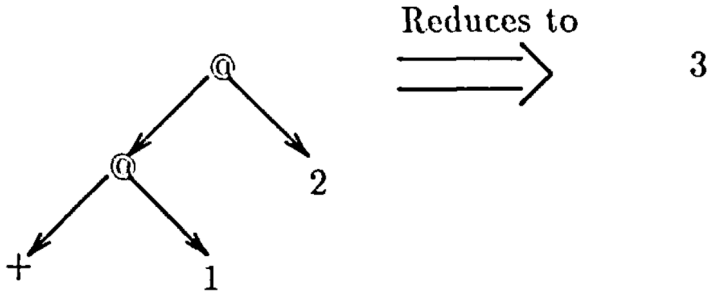

Figura 4.2: *Redução de uma expressão. Imagem retirada de (2).*

4.1.5 Independência de arquitetura

Boas abstrações de paralelismo (que abstraiem as questões de mais baixo nível) encorajam a portabilidade. Em casos extremos essa abstração pode esconder detalhes da implementação do paralelismo levando a um modelo de paralelismo implícito. Como as questões de baixo nível frequentemente dependem das propriedades de uma arquitetura específica, uma abordagem de mais alto nível é significativamente menos dependente de arquitetura do que as abordagens de mais baixo nível. O custo da idependência de arquitetura é elaborar processores da linguagem: o compilador ou o sistema em tempo real, ou a combinação de ambos, precisam se adaptar ao paralelismo de alto nível para entender a arquitetura. Quando o paralelismo é tratado no nível de execução, existem padrões como o PVM e o MPI que permitem que linguagens consigam abstrair as características da arquitetura. Mas, diferentemente das linguagens imperativas, as linguagens funcionais permitem um grau maior de abstração do que esses padrões por meio de funções de grau superior e de polimorfismo.

5 Conclusões

As linguagens funcionais possuem várias características que favorecem a paralelização, como o maior controle ou a ausência dos efeitos colaterais, o maior nível de abstração e, consequentemente, a maior modularização do programa. Essas facilidades permitem que o programador se foque no algoritmo paralelo sem ter que se preocupar com deadlock, portabilidade de arquiteturas, entre outras questões de baixo nível relacionadas a paralelismo. Acreditamos que essas grandes facilidades tornam a programação paralela bastante natural em linguagens funcionais e compensam o fato de as linguagens funcionais não expressarem certos algoritmos paralelos não-determinísticos (pois programas funcionais paralelos são determinísticos) e serem ligeiramente menos efiecientes que algumas linguagens imperativas.

Referências Bibliográficas

[1] E. Belikov, H.-W. Loidl, and G. Michaelson, "Characterisation of Parallel Functional Applications," in *Draft Proceedings of the 2014 Symposium on Trends in Functional Programming*, 2014. 1

[2] P. Roe, *Parallel programming using functional languages*. PhD thesis, University of Glasgow, 1991. 1, 7, 9

[3] "Functional programming," June 2015. https://en.wikipedia.org/w/index.php?title=Functional_programming. 2

[4] "Functional programming," 2014. https://wiki.haskell.org/Functional_programming. 2

[5] K. Hammond, "Why Parallel Functional Programming Matters: Panel Statement," in *Reliable Software Technologies - Ada-Europe 2011* (A. Romanovsky and T. Vardanega, eds.), no. 6652 in Lecture Notes in Computer Science, pp. 201–205, Springer Berlin Heidelberg, 2011. 5

[6] S. Marlow, "Parallelism \= concurrency," 2009. https://ghcmutterings.wordpress.com/2009/10/06/parallelism-concurrency/. 5

[7] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, J. Rebón, and P. W. Trinder, "Comparing Parallel Functional Languages: Programming and Performance," *Higher Order Symbol. Comput.*, vol. 16, pp. 203–251, Sept. 2003. 7