Computação Concorrente (MAB-117)
Cap. I: Introdução e histórico da programação concorrente

Prof. Silvana Rossetto

[1]Departamento de Ciência da Computação (DCC)
Instituto de Matemática (IM)
Universidade Federal do Rio de Janeiro (UFRJ)
Março de 2012

Prefácio A disciplina *Computação Concorrente* passou a compor o Currículo do curso de Ciência da Computação do DCC/IM/UFRJ em 2009. Os objetivos da disciplina são:

- Introduzir o paradigma de programação concorrente capacitando o aluno a construir programas que fazem uso da execução concorrente (simultânea) de várias tarefas computacionais interativas, as quais podem ser implementadas como processos separados ou como um conjunto de threads criadas dentro de um único processo.
- Apresentar as questões relacionadas com a interação e a comunicação correta entre as diferentes tarefas e com a coordenação do acesso concorrente aos recursos computacionais.
- Discutir e mostrar exemplos de modelagem e implementação de problemas computacionais que são concorrentes por natureza.
- Apresentar bibliotecas e mecanismos oferecidos pelas linguagens de programação para o desenvolvimento de aplicações concorrentes, paralelas e distribuídas e fazer uso dessas ferramentas em atividades práticas da disciplina.

1. Caracterização da programação concorrente

A **programação concorrente** está relacionada com a atividade de construir programas de computador que incluem **linhas de controle distintas**, as quais podem executar simultaneamente. Dessa forma, um programa dito *concorrente* se diferencia de um programa dito *sequencial* por conter mais de um contexto de execução ativo ao mesmo tempo. As diferentes linhas de controle de um programa concorrente cooperam para a execução de uma tarefa única (finalidade do programa). Para isso, na maioria dos casos é necessário usar mecanismos que permitam a **comunicação** entre as linhas de controle e a **sincronização** das suas ações de forma a garantir a correta execução da atividade fim.

Concorrência não é um conceito novo (a linguagem Algol68 já incluia características de programação concorrente). O crescente interesse por concorrência, entretanto, é um evento mais recente e se deve a maior disponibilidade de multiprocessadores de custo mais baixo e à proliferação de aplicações gráficas, multimídia e Web, todas naturalmente representadas por várias linhas de controle concorrentes.

A concorrência aparece em qualquer sistema de computação, por exemplo: (i) no nível de *lógica digital*, boa parte dos eventos ocorre em paralelo (mais de um circuito opera simultaneamente no processamento de uma instrução de máquina); (ii) no nível mais alto, a técnica de *pipelining* é usada para explorar o paralelismo entre instruções.

No nosso estudo, iremos abordar a concorrência no *nível de aplicação*, representada por construções que são semanticamente visíveis para o programador.

Dentro de um programa concorrente usaremos o termo "thread" para referenciar um contexto de execução distinto. As threads de um programa poderão ser implementadas dentro de um ou mais processos gerenciados pelo Sistema Operacional. O termo "tarefa" será usado para referenciar uma unidade de trabalho bem definida que deve ser executada por alguma thread.

Usaremos o termo "paralelismo real" para caracterizar programas concorrentes no qual a execução acontece, de fato, em mais de um contexto ao mesmo tempo, e o termo "paralelismo aparente" para caracterizar programas concorrentes que executam em sistemas preemptivos com um único processador. Paralelismo real requer hardware paralelo (máquinas multiprocessadores ou multicomputadores). Semanticamente, entretanto, não há diferença entre "paralelismo real" e "paralelismo aparente" pois as trocas de contextos de execução ocorrem em instantes de tempo imprevisíveis, então as mesmas técnicas de programação se aplicam nos dois casos [1].

2. Motivação para a programação concorrente

Os avanços mais recentes em termos de Arquitetura de Computadores têm implicado muito mais no incremento do paralelismo das máquinas (arquiteturas multiprocessadores ou *multicores*) do que no aumento significativo da velocidade de *clock*. Como explorar esse paralelismo é um dos desafios da Ciência da Computação moderna e está implicitamente relacionado com a programação concorrente [2].

De outro lado, os avanços nas tecnologias de comunicação possibilitam o compartilhamento de recursos de hardware (ex., impressora, servidores) e/ou de software (ex., bases de dados) localizados em máquinas distintas e remotas. Novamente as questões básicas de concorrência aparecem como fundamentos para a construção de aplicações distribuídas capazes de aproveitar de forma eficiente e correta a infraestrutura de comunicação provida pelas redes de computadores.

Outros motivos para a programação concorrente surgem como requisitos das próprias aplicações, entre os quais se destacam [1]:

1. **Necessidade de capturar a estrutura lógica de um problema**. Exemplos típicos incluem servidores Web e aplicações gráficas onde é preciso manter trilhas de execução independentes executando diferentes tarefas. A maneira mais simples e lógica de estruturar esses programas é representar cada tarefa como uma linha de controle distinta.
2. **Necessidade de lidar com dispositivos independentes**. Um exemplo clássico são os Sistemas Operacionais (SOs). Durante a operação de uma uma máquina, o SO pode ser interrompido por diferentes dispositivos de hardware a qualquer tempo. Para isso o SO precisa dispor de um contexto de execução para representar/guardar a tarefa que estava executando antes da interrupção acontecer e outro contexto de execução para tratar a própria interrupção. Outro exemplo é o roteamento de mensagens na Internet. Em linhas gerais, trata-se de um programa concorrente executando em vários servidores/roteadores ao redor do mundo.
3. **Necessidade de aumentar o desempenho das aplicações**: Nessa categoria o exemplo mais comum são as aplicações de computação científica que requerem

grande quantidade de processamento. A paralelização dessas aplicações é a alternativa para alcançar um tempo de processamento factível.

3. Histórico da programação concorrente

Os primeiros computadores eram máquinas **monousuário**, usadas em modo *stand-alone*, i.e., cada usuário tinha uso exclusivo sobre os recursos da máquina. Enquanto o usuário examinava a saída de um programa ou tentava descobrir *bugs* do código, o computador era sub-utilizado [1].

Para otimizar o uso dos computadores, criou-se um modo de operação no qual os usuários criavam **jobs** (sequências de programas e seus respectivos valores de entrada) e então os submetia a um operador para execução, chamado **processamento em lote**. O operador mantinha um *batch* (lote) de *jobs* enfileirados em cartões ou fitas magnéticas para entrada no sistema. Ao final da operação, o programa transferia o controle de volta para um **programa monitor** (forma primitiva de Sistema Operacional), o qual lia o próximo programa na memória (do *job* atual ou do seguinte) para execução, sem intervenção do operador.

Esse modo de operação ainda deixava o processador ocioso (*idle*) por muito tempo, especialmente no caso de aplicações com pouco processamento e muitas operações de entrada e saída (E/S). Para executar operações de E/S (ex., ler dados da memória, escrever resultados na impressora) em um sistema em lote (*batch*) simples, o processador envia um comando para o dispositivo E/S e fica em **espera ocupada** (*busy-wait*) pelo término da operação, normalmente testando uma variável que o dispositivo modifica quando finaliza a sua tarefa.

Para explorar os ciclos de CPU perdidos com a espera ocupada, foi desenvolvida a técnica de *sobreposição de E/S e computação*, e propostas as idéias de: (i) **E/S dirigido a interrupção** (elimina a necessidade de espera ocupada); e (ii) **multiprogramação** (permite mais de um programa na memória principal ao mesmo tempo). As duas inovações precisaram de suporte de hardware: no primeiro caso para implementar o mecanismo de `interrupção`, e no segundo caso para implementar mecanismos de `proteção de memória`. Quando o programa corrente precisa esperar por um dispositivo de E/S, outro programa pode fazer uso do processador.

3.1. Multiprogramação e E/S dirigido a interrupção

Em um **sistema em lote multiprogramado**, o Sistema Operacional precisa manter informações sobre quais programas estão esperando por E/S e quais estão prontos para executar. Para ler ou escrever dados, o programa corrente passa o controle para o Sistema Operacional. O Sistema Operacional, por sua vez, envia o comando para o dispositivo apropriado, o qual inicia a operação solicitada, e depois transfere o controle para outro programa executar. Quando o dispositivo termina a operação, ele gera uma interrupção, o que faz o processador transferir o controle de volta para o Sistema Operacional. O Sistema Operacional identifica qual programa estava bloqueado esperando por essa interrupção e o desbloqueia, permitindo que ele volte a concorrer pelo uso do processador. O Sistema Opercacional escolhe um dos programas apto a executar e transfere o controle para esse programa. Nesse modo de operação, o processador fica ocioso apenas se todos os programas carregados na memória estiverem bloqueados esperando por E/S.

O modelo de E/S dirigido a interrupção introduziu o **conceito de concorrência dentro dos sistemas operacionais**. Como uma interrupção pode acontecer a qualquer momento, incluindo quando o controle já está com o Sistema Operacional, os **tratadores de interrupção** e o **código principal do Sistema Operacional** são *linhas de controle concorrentes*. Se uma interrupção ocorre enquanto o Sistema Operacional está modificando uma estrutura de dados que pode também ser usada pelo tratador da interrupção, então é possível que o tratador veja a estrutura de dados em um estado inconsistente.

Esse problema é um exemplo de **condição de corrida**: a linha de execução do Sistema Operacional e a linha de execução do tratador de interrupção estão "correndo" para um ponto no código no qual ambas acessam um **objeto comum**, e o comportamento do sistema dependerá de qual linha de execução chega primeiro. Para garantir o funcionamento correto é necessário **sincronizar** as ações das linhas de execução, i.e., seguir passos explícitos para controlar a ordem na qual suas ações ocorrem.

Importante notar que nem toda `condição de corrida` é "ruim", i.e., algumas vezes qualquer saída do programa é aceitável. A sincronização é usada para resolver condições de corrida "ruins": *aquelas que fazem o programa produzir resultados incorretos*.

3.2. Tempo compartilhado e distribuição

Com o incremento do espaço de memória principal e com o desenvolvimento da memória virtual, tornou-se possível construir **sistemas com um número arbitrário de programas carregados simultaneamente na memória**. Ao invés de submeter *jobs* em modo *offline*, o usuário pode interagir com o computador diretamente. Para prover **respostas interativas**, o Sistema Operacional precisou implementar uma nova idéia, chamada **preempção**: ao invés de alterar o programa corrente apenas quando ocorre E/S, define-se **fatias de tempo** e muda-se o programa em execução sempre que a sua fatia completar. Esses sistemas ficaram conhecidos como **sistemas de tempo compartilhado** (ou *timesharing*), e já eram comuns nos anos 70.

Quando acrescidos de mecanismos que permitiram `compartilhamento de dados` e outras formas de comunicação entre os programas em execução, a **concorrência foi introduzida nas aplicações no nível do usuário**.

Mais tarde, com o surgimento das Redes de Computadores, introduziu-se o paralelismo de verdade na forma de **sistemas distribuídos**: programas que executam em máquinas separadas fisicamente e se comunicam via troca de mensagens.

Parte dos Sistemas Distribuídos atuais refletem a primeira razão para a concorrência: responder às características da própria aplicação (ex., aplicações Web). Outra parte dos Sistemas Distribuídos refletem a segunda razão para a concorrência: a necessidade de lidar com vários dispositivos. E, por fim, alguns Sistemas Distribuídos, caracterizados pelas aplicações paralelas, refletem a terceira razão para a concorrência: a necessidade de distribuir o processamento entre vários processadores para reduzir o tempo final da aplicação.

4. Desafios para a programação concorrente

Para escrever um programa concorrente é necessário definir quais contextos de execução criar e de que forma eles deverão interagir dentro da aplicação. As decisões devem levar

em conta as especificidades da aplicação e o hardware disponível.

Uma visão ideal da programação concorrente (proporcionada pela semântica que algumas linguagens de programação são capazes de oferecer, como é o caso das linguagens declarativas) é a *concorrência implícita*: partes independentes do programa podem executar em paralelo, e sempre que uma parte depende dos resultados produzidos por outra, a comunicação e a sincronização ocorrem do forma implícita.

Entretanto, nas linguagens de programação imperativas (ou na implementação das linguagens declarativas em máquinas *von Neumann*), é necessário escrever **programas concorrentes imperativos**, nos quais a *concorrência*, a *comunicação* e a *sincronização* são **explícitas**. Assim, dentro de uma programa concorrente é preciso criar explicitamente vários **processos (ou threads)**, os quais (processos ou threads) devem se comunicar e sincronizar suas ações **lendo/escrevendo em variáveis compartilhadas** ou **enviando/recebendo mensagens**.

Aprender a criar programas concorrentes é como apreender um novo "jogo": é preciso conhecer as regras, as ferramentas disponíveis e as estratégias mais usadas. No caso da programação concorrente: (i) as *regras* são os modelos formais que ajudam a entender e desenvolver programas corretos; (ii) as *ferramentas* são os mecanismos oferecidos pelas linguagens de programação para descrever computações concorrentes; e (iii) as *estratégias* são os paradigmas de programação adequados para as aplicações tratadas [3].

A programação concorrente é desafiadora porque os sistemas de computação modernos são inerentemente **assíncronos**: as atividades podem ser suspensas ou retardadas sem qualquer previsão devido a eventos de: interrupção, preempção, atualização da memória cache, atraso de comunicação, e outros. Em função disso, o desenvolvimento de aplicações divididas em partes, com contextos de execução separados, deve ser visto por dois ângulos: (1) **computabilidade**, conjunto de princípios que abordam o que pode ser computado em um ambiente concorrente e assíncrono; (2) **desempenho**, avaliação prática do ganho em desempenho (tempo de processamento, capacidade de interatividade, tolerância a falhas, etc.) das soluções implementadas [2].

No mundo ideal, quando migramos uma aplicação que executa em uma máquina monoprocessador para uma máquina n-processador espera-se que seja obtido um incremento computacional de n-vezes, i.e., se a aplicação inicialmente consumia t unidades de tempo para concluir a sua tarefa, na máquina n-processada esse tempo será de t/n unidades de tempo. Na prática isso nunca acontece.

Um bom exemplo é o caso de 5 amigos que decidem pintar 5 cômodos de uma casa [2]. Se todos os cômodos são do mesmo tamanho, então faz sentido que cada um deles fique responsável por pintar um dos cômodos. Considerando que todos pintam com a mesma velocidade, alcançaremos o incremento de 5-vezes no desempenho final, comparado à atribuição da tarefa inteira para uma única pessoa. A estória fica mais complicada se um dos cômodos tiver tamanho diferente dos demais, por exemplo se um deles tiver o dobro do tamanho. Nesse caso não teremos mais o ganho anterior porque o tempo necessário para completar a tarefa inteira corresponderá ao tempo necessário para pintar o cômodo maior.

Esse tipo de análise é muito importante para a computação concorrente. O ganho de tempo na execução de uma tarefa complexa é limitado pela quantidade de código que

deve continuar sendo executado sequencialmente. Uma maneira de medir o ganho obtido é calcular a razão entre o tempo que um processador gasta para completar uma tarefa e o tempo que n-processadores concorrentes demandam para completar a mesma tarefa.

Seja p a fração da tarefa que pode ser executada em paralelo. Vamos assumir por simplicidade que o tempo para um processador completar a tarefa é 1. Com n-processadores concorrentes a parte paralela gastará p/n e a parte sequencial $1 - p$, somando $(1 - p + p/n)$. Então, a razão entre o tempo gasto com um processador e o tempo com n-processadores concorrentes será: $S = 1/(1 - p + p/n)$. Assumindo que cada cômodo é uma unidade de execução e que o cômodo maior são duas unidades, e atribuindo um pintor por cômodo, 5 das 6 unidades de execução podem ser realizadas (pintadas) em paralelo, então termos: $p = 5/6$ e $1 - p = 1 - 5/6 = 1/6$. Daí, $S = 1/(1/6 + (5/6)/5) = 1/(1/6 + 1/6) = 1/(2/6) =$**3**. Ou seja, ao invés de incrementar o tempo final de 5-vezes, conseguimos incrementar apenas de 3-vezes.

A solução nesse caso parece ser flexibilizar mais a paralelização: assim que um pintor termina de pintar um cômodo ele deve ajudar outro pintor que ainda não terminou. A questão é que a pintura compartilhada requer coordenação entre os pintores, acrescentando um custo extra de comunicação e coordenação entre as partes.

Alguns problemas são facilmente paralelizáveis, i.e., eles podem ser divididos em componentes que executam concorrentemente (ex., computação científica, aplicações gráficas). Em geral, entretanto, para um dado problema e uma máquina com 10 processadores, mesmo que consigamos paralelizar 90% da solução, o ganho de tempo será de apenas 5-vezes e não de 10 como seria esperado. O esforço principal da programação concorrente está exatamente em conseguir paralelizar o máximo possível esses 10% restantes. Essa tarefa não é trivial pois exige o incremento do código com mecanismos de comunicação e sincronização entre atividades.

O foco do curso de Computação Concorrente está justamente no estudo das ferramentas e técnicas que permitem aos programadores codificar as partes do código que requerem coordenação (comunicação e sincronização) entre as linhas de execução distintas, pois os ganhos alcançados com essa paralelização podem trazer um grande impacto no desempenho final da aplicação.

Exercícios

1. Diferencie programa sequencial e programa concorrente.
2. Argumente as razões para a computação concorrente (por que escrever programas concorrentes? o que tem causado o aumento do interesse pela programação concorrente nos últimos anos?).
3. Descreva a evolução da operação dos computadores: do modo *stand-alone* para processamento em lote, multiprogramação e sistemas de tempo compartilhado.
4. O que é E/S dirigida à interrupção? O que isso tem a ver com concorrência?
5. O que é condição de corrida?
6. O que é preempção?
7. Se uma tarefa é composta por 8 atividades das quais 6 podem ser executadas em paralelo, qual será o ganho em executá-la em uma máquina com 4 processadores ao invés de uma máquina monoprocessador?

Referências

[1] M. L. Scott. *Programming Language Pragmatics*. Morgan-Kaufmann, 2 edition, 2006.

[2] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan-Kaufmann, 2008.

[3] G. Andrews. *Concurrent Programming — Principles and Practice*. Addison-Wesley, 1991.