LISTA 1

José Augusto Agripino de Oliveira

DCA0123 - Programação Concorrente e Distribuída

Questões feitas: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12 (11/21)

1. Com suas próprias palavras, explique porque a programação paralela deixou de ser uma alternativa à computação sequencial convencional.

A computação paralela deixou de ser uma alternativa à computação sequencial convencional por ser, agora, uma necessidade para que possamos resolver problemas complexos apresentados no cotidiano, como a dobra de proteínas. Além disso, se não usássemos essa forma de computação, estaríamos desperdiçando parte do potencial computacional que temos hoje. Três fatores corroboram para escolha da computação paralela: resolver problemas de forma mais eficiente, resolver problemas maiores e ter uma resolução mais rápida.

2. Explique, na sua opinião, se as modificações feitas à máquina de Von Neuman, como cache e memória virtual, alteram a sua classificação na taxonomia de Flynn e porque. Estenda sua resposta a pipelines, múltiplas issues e multithreding de hardware.

Na minha opinião, para as modificações à máquina de Von Neuman, adicionando cache e memória virtual, não muda a taxonomia de Flynn, que é de SISD (Single Instruction - Single Data). Isso ocorre pela cache e pela memória virtual não possibilitarem o uso de múltiplas instruções e nem a aplicação de uma instrução em vários dados, mas sim aumenta apenas a velocidade de consulta aos dados, e faz com que a execução de programas não fique limitada à memória física, respectivamente.

Pipelines e múltiplas issues, que referem-se ao paralelismo de instruções, são, respectivamente, unidades funcionais sendo arranjadas em estágios para que instruções possam ser executadas ao mesmo tempo, e a réplica de unidades funcionais para tentar executar diferentes instruções de um programa ao mesmo tempo. Ao acrescentar essas modificações à máquina de Von Neuman, na minha opinião, a taxonomia de Flynn irá mudar de SISD para MISD (Multiple Instruction - Single Data), uma vez que poderá ocorrer o paralelismo de instruções.

Quanto à adição de multithreding, apenas vai fazer com que um processo possa fazer algumas coisas “ao mesmo tempo”, com o esquema tradicional de time-sharing. Desse modo, é apenas uma melhoria na máquina de Von Neuman.

3. Dê dois exemplos de arquiteturas paralelas, uma do tipo MIMD e outra do tipo SIMD, e explique suas principais diferenças no que diz respeito à forma de processamento paralelo.

Um exemplo de arquitetura paralela do tipo MIMD é a de multicore, já do tipo SIMD é uma arquitetura GPU.

Na arquitetura SIMD (Single Instruction - Multiple Data), existem algumas instruções específicas que utilizam registradores vetoriais, ou seja, uma única instrução processará um vetor inteiro e executará a instrução nesses múltiplos dados, ao mesmo tempo, caracterizando uma arquitetura SIMD.

Para uma arquitetura MIMD (Multiple Instruction - Multiple Data), também ocorre o uso de registradores vetoriais, porém, há a possibilidade do uso de várias instruções, já que existem mais de um núcleo de processamento.

4. Explique como o uso de uma variável compartilhada em um programa com múltiplos threads em uma arquitetura de memória compartilhada com múltiplas caches privadas pode deteriorar o desempenho paralelo do programa.

O uso de uma variável compartilhada em um programa com múltiplas threads em uma arquitetura de memória compartilhada com múltiplas caches privadas pode deteriorar o desempenho paralelo do programa, pois pode existir o problema de coerência de cache. Sempre que essa variável compartilhada for alterada em alguma thread, terá que ocorrer uma escrita, para atualizá-la, na memória principal, e em seguida várias leituras para as diferentes threads, para manter a variável coerente entre todas as threads. Esse processo de leitura e escrita é muito custoso, fazendo com que seja necessário que as threads esperem o dado ser atualizado para que a computação possa continuar.

Isso ocorre por existir apenas caches privadas, uma vez que cada cache possui apenas os valores de cada thread.

5. Explique os conceitos de localidade temporal e espacial no contexto de caches e memória virtual e exemplifique como o programador pode evitar armadilhas de desempenho com o conhecimento desses conceitos.

Localidade temporal diz respeito a quando a cache ou a memória virtual enche e é preciso escolher qual linha de cache ou página da memória virtual irá sair e dar lugar a novos dados. Para evitar armadilhas de desempenho, o programador deve ter conhecimento em mapeamento de cache, que é dividido em mapeamento associativo completo, mapeamento

associativo direto e mapeamento associativo de n-vias. Com esse conhecimento, é possível saber qual linha de cache ou página da memória virtual será retirada e escolher isso de forma a corroborar com o desempenho do programa em questão.

Localidade espacial refere-se à arquitetura assumir que os dados que serão utilizados pelo programa estão sequencialmente alocados na memória. Desse modo, sempre é pego uma linha da cache (64 bytes, se for uma arquitetura x86) ou uma página da memória virtual (4 a 16 kilobytes), apesar do dado requerido ocupar menos espaço, uma vez que isso é feito para poupar acessos repetidos à memória. Para evitar armadilhas de desempenho, o programador deve saber que deve alocar os dados necessários para o programa em sequência, evitando, assim, consultas desnecessárias à memória.

6. Falso compartilhamento pode ser tão prejudicial ao desempenho de um programa paralelo quanto o que foi exposto na questão 4. Como isso ocorre e como pode ser evitado?

O falso compartilhamento ocorre quando dois programas ou mais usam dados diferentes de uma mesma linha de cache, e quando um desses programas altera o dado e tem que escrevê-lo na linha de cache, ela é marcada como suja. Quando um outro programa tenta acessar a mesma linha, decorrente de uma mudança em um outro dado, esse acesso é negado, já que a linha está marcada como suja. Sendo assim, é um falso compartilhamento pelos programas entenderem que são o mesmo dado por estarem na mesma linha de cache, entretanto são outros. Ao acontecer isso, é preciso que o programa acesse a memória principal para escrever no dado, sendo uma operação muito custosa, como já sabemos.

Uma alternativa para mitigar esse custo, é fazer com que cada programa tenha uma cache privada, e os resultados alterados sejam postos nela, e quando os processos estiverem terminados, os dados são compartilhados no armazenamento compartilhado. Mesmo que ocorra o falso compartilhamento, será preciso ir na memória principal somente uma vez, e não sempre que cada dado for modificado.

7. O que é a lei de Amdahl e como ela se relaciona com a lei de Gustafson?

A lei de Amdahl fornece o speedup máximo que um programa paralelo pode obter. O limite está condicionado à parte não paralelizável do código. Assim, supondo que exista uma fração “r” do código que não é paralelizável, a lei de Amdahl diz que não podemos obter um speedup melhor que 1/r, independente do número de processadores utilizados. Já a lei de Gustafson leva em conta algo esquecido pela lei de Amdahl, que é o tamanho do problema. À

medida que aumentamos o tamanho do problema, a fração “inerentemente serial” diminui de tamanho. Essa lei também nos fornece o speedup máximo. Entretanto, enquanto a lei de Amdahl parte do tempo de execução sequencial para estimar o speedup máximo conseguido com o uso de múltiplos processadores, a lei de Gustafson faz precisamente o contrário, ou seja, parte do tempo de execução em paralelo para estimar o speedup máximo comparado com a execução sequencial.

8. Como se calcula a eficiência de um programa paralelo?

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}$$

A eficiência é calculada a partir da fórmula acima, onde T_{serial} é o tempo de processamento serial, $T_{parallel}$ é o tempo de processamento paralelo e “p” é o número de processadores utilizados. Essa eficiência está ligada ao tamanho do programa, como previsto na lei de Gustafson.

9. Quando se mede intervalos de tempo de execução de um programa paralelo, é aconselhável realizar mais de uma medição. É comum se pensar na média como cálculo de agregação desses valores, mas há argumentos em favor do uso da mediana. Quais são esses argumentos? Por outro lado, quando se mede os tempos de uma região paralela do programa, cada thread ou processo tem sua própria medição e, neste caso, nem a média nem a mediana são adequadas, por quê? O que é mais adequado?

A mediana indica o valor do meio dentro de uma amostra. Sendo assim, fazendo várias medições do tempo de execução de um programa paralelo, o valor da mediana retorna um tempo de execução com pouca interferência de picos de altos e baixos de processamento, decorrente de estar ou não rodando outros processos.

Quando se mede o tempo de threads ou processos não é adequado o uso da média e nem da mediana, pois é improvável que algum evento externo ao programa faça com que a execução da thread seja mais rápida do que o seu melhor tempo. O mais adequado é capturar o tempo mínimo de execução da thread.

10. Como é possível determinar a escalabilidade de um programa paralelo? O que faz um programa ser escalável, fracamente escalável ou fortemente escalável?

É possível determinar a escalabilidade de um programa paralelo observando a eficiência do programa ao mudarmos o tamanho do problema e o número de processadores.

Um programa é escalável quando aumenta-se o número de processadores e é possível encontrar uma taxa de incremento do tamanho do problema que faça com que a eficiência "E" seja igual à eficiência original.

Para um programa ser fracamente escalável, é preciso que a eficiência mantenha-se igual aumentando na mesma proporção o tamanho do problema e o número de processadores.

Já se aumentarmos apenas a quantidade de processadores e a eficiência continuar a mesma, é um programa fortemente escalável.

12. Baixe omp_trap1.c do site do livro, e apague a critical directive. Agora compile e execute o programa com mais e mais threads e valores cada vez maiores de n. Quantas threads e quantos trapézios são necessários antes que o resultado seja incorreto?

Conforme testes realizados no supercomputador com o programa omp_trap1.c, a partir de 4 threads e 80 trapézios, o resultado começa a ser diferente do esperado. Isso ocorre pela condição de corrida, que é quando as threads compartilham a mesma variável, e que sem a critical directive, elas não esperam que uma outra thread termine de gravar o valor modificado da variável compartilhada. Assim, isso faz com que as threads disputem qual delas grava por último o valor da variável, sobrepondo o valor da thread que começou a escrever o valor primeiro. Isso gera incoerências nos resultados. Essa parte do código é chamada de seção crítica.