UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE – UFRN
CENTRO DE TECNOLOGIA – CT
DEPARTAMENTO DE COMPUTAÇÃO E AUTOMAÇÃO – DCA

PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA – 2023.2

LUIZ HENRIQUE ARAÚJO DANTAS – 20220044224

Questões realizadas:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14

13/21 questões

Natal/RN
Outubro/2023

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

A computação sequencial convencional durante muitos anos teve o proveito de processadores cada vez mais rápidos, até que houve o problema de limitação física no qual os transistores não conseguiam ser alocados no espaço destinado à eles por gerar o problema de refrigeração do calor emitido pelos transistores, com esse fato a taxa de melhoria de desempenho veio a diminuir, em contrapartida, os fabricantes de chips passaram a trabalhar com circuitos integrados multicore, que contém múltiplos processadores em um único chip.

Dessa forma, se os desenvolvedores quisessem continuar aumentando a velocidade no processamento algo teria que mudar, e assim a programação paralela deixou de ser uma alternativa à computação sequencial para se tornar "o protagonista" para se criar programas que desejam atingir grandes desempenhos.

Com toda essa revolução, as indústrias de produção de chip passaram a construir os processadores multicore (vários núcleos em um único chip), em detrimento aos processadores de único núcleo focados em velocidade e maior complexidade.

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.**

A taxonomia de Flynn classifica um sistema de acordo com a quantidade de fluxos de instruções que um sistema recebe e a quantidade de fluxo de dados que ele pode tratar.

A taxonomia de Flynn possui as seguintes classificações:

- SISD (Single Instruction, Single Data): única instrução por vez sobre um único dado, um sistema sequencial. A máquina de Von Neumann é um exemplo de um sistema SISD.
- SIMD (Single Instruction, Multiple Data): sistema que aplica uma instrução sobre vários dados, aplicando uma instrução em todos os dados e em seguida, a próxima instrução e assim por diante. Exemplos desse sistema são as GPUs e processadores vetoriais.
- MISD (Multiple Instruction, Single Data): sistemas de realizam instruções múltiplas em cima de um único dado.

- MIMD (Multiple Instruction, Multiple Data): sistemas de instruções múltiplas e dados múltiplos, esses sistemas executam vários fluxos de instruções independentes, cada um dos quais pode ter seu próprio fluxo de dados.

Como a arquitetura clássica de Von Neumann é composta pela: CPU, Memória principal e interconexão e nativamente SISD, o seu gargalo se encontra no barramento que limita a taxa com que as instruções podem ser executadas.

Memória cache e Memória virtual: a memória cache são locais da memória no qual podem ser acessados mais rapidamente, elas ficam localizadas entre a CPU e a memória principal e tem como objetivo reduzir os atrasos associados ao acesso à memória principal. Os dados armazenados nela utilizam o princípio da localidade, onde os dados que são mais utilizados estão mais próximos. assim linhas de dados e instruções são transferidos entre a memória e os caches, quando o item de dados ou instrução está na cache é considerado um hit, se não estiver na cache é considerada uma falha de cache. O funcionamento da memória virtual contém apenas partes restantes (as partes principais ficam armazenadas na memória principal) de instruções e dados de um programa armazenadas no espaço de troca (swap), ela opera com blocos de dados e instruções contíguas, chamadas de páginas, endereçando com endereços virtuais que são independentes dos endereços físicos reais, para corresponder os endereços físicos com os endereços virtuais é armazenada na memória uma tabela de páginas, que traz flexibilidade de armazenar dados e instruções de um programa em qualquer lugar da memória.

Pipelines: as pipelines se assemelham às linhas de montagem de uma fábrica, onde cada equipe é responsável por uma etapa da montagem, enquanto uma equipe realiza uma etapa, outra equipe pode executar operação que não dependa da operação anterior e outra equipe pode realizar uma etapa que depende das duas etapas anteriores. Trazendo ao contexto computacional, as unidades funcionais de um processador são ordenadas em sequência, com a saída de uma unidade sendo a entrada da próxima, digamos que um dado esteja sendo processado pela segunda unidade funcional, outro dado pode ser processado pela primeira.

Em geral, um pipeline com k estágios não obterá uma melhoria de k vezes no desempenho se o tempo das unidades forem diferentes, os estágios funcionarão na velocidade da unidade mais lenta, atrasos como espera por operandos podem fazer com que a pipeline pare.

Múltiplas issues: Se refere a capacidade de executar múltiplas instruções por ciclo de clock. Se as unidades funcionais foram escalonadas em tempo de compilação, o sistema de

multiple issues será do tipo estático, se as unidades funcionais forem agendadas em tempo de execução, o sistema de multiple issues será dinâmico. Um processador que suporta multiple issues dinâmico é considerado superescalar.

Multithreading de hardware: permite que uma única CPU consiga executar várias threads simultaneamente. Tentativa de manter o processador o mais ocupado possível, trocando rapidamente entre threads. Fornece um meio para que os sistemas possam realizar um trabalho útil quando a tarefa que está sendo executada no momento estiver paralisada (até que um acesso à memória seja concluído, por exemplo) antes de poder executar uma instrução.

A adição de estratégias como memória cache e memória virtual não alteram sua arquitetura SISD uma vez que não altera a forma como as instruções serão executadas, a adição da memória cache e virtual melhoram o desempenho da máquina de Von Neumann.

A adição do pipeline, onde a saída de uma unidade funcional será a entrada da próxima unidade, muda a taxonomia de Flynn, pois agora há a possibilidade do paralelismo de instruções, o fluxo de instruções será múltiplo, os dados continuam sendo de fluxo único e as unidades funcionais, agora organizadas em estágios, tentarão executar diferentes instruções de um programa ao mesmo tempo, transformando assim para um sistema MISD.

Para o multiple issues, a adição dessa estratégia na máquina de Von Neumann também alteraria sua taxonomia pois pode acontecer o paralelismo de instruções, enquanto aos dados continuarão sendo carregados de forma sequencial, executando agora como múltipla instruções e um fluxo de dado (MISD).

No multithreading de hardware irá melhorar o desempenho do sistema onde o tempo de espera entre threads será menor, enquanto as especificações da arquitetura SISD continuam imutáveis, executando apenas uma instrução e dados por vez.

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.**

O SIMD se refere ao Single Instruction Multiple Data, onde a primeira instrução é aplicada em todo o conjunto de dados, a segunda instrução da mesma forma e assim por diante, esse tipo de estrutura é aplicada em programas paralelo de dados, onde os dados são divididos entre os processadores e cada item de dados é submetido mais ou menos à mesma sequência de instruções. Um exemplo de SIMD são as GPUs e os processadores vetoriais. Os

sistemas SIMD são ideais para paralelizar loops simples que operam em grandes quantidades de dados.

Já o MIMD se refere ao Multiple Instruction Multiple Data, o funcionamento do MIMD é baseado na execução de múltiplos fluxos de instruções diferentes operando em múltiplos fluxos de dados, que consiste de uma coleção de unidades de processamento ou núcleos totalmente independentes, onde cada um possui a própria unidade de controle e ULA. Os sistemas MIMD possuem processadores que podem funcionar no seu próprio ritmo, sendo assim assíncronos. Um exemplo de MIMD é um sistema de memória distribuída, onde cada processador possui sua própria memória privada e o processador-memória se comunica a partir de uma interconexão, onde os processadores explicitamente se comunicam a partir de troca de mensagens e funções especiais que fornecem acesso à memória de outro processador.

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.**

Como as caches de uma CPU são gerenciadas por hardware do sistema, os programadores não possuem controle direto sobre elas. O uso de uma variáveis compartilhadas em um sistema de memória compartilhada gera um problema chamado de coerência de cache. A mesma variável pode ser armazenada na cache de dois núcleos diferentes e, se um núcleo atualizar o valor da variável, o outro núcleo não terá conhecimento da alteração. Com isso, existem duas estratégias possíveis para que a coerência de cache seja solucionada:

- Snooping cache: seu funcionamento é baseado em “bisbilhotar” as linhas de caches dos cores, quando um core atualiza a cópia de um valor x armazenada em sua cache e transmite essa informação pela interconexão, o core que está observando verá que o valor x foi atualizado e irá invalidar sua cópia de x. A transmissão informa aos outros núcleos que a linha de cache contendo x foi atualizada e não que o valor x foi atualizado.
- Directory-based: o funcionamento da coerência de cache baseada em diretórios utiliza uma estrutura de dados chamada diretório, no qual armazena o status de cada linha de cache, se trata de uma estrutura distribuída onde cada par core/memória pode conter uma parte da estrutura que especifica o status das linhas de cache em sua memória local. Assim, quando uma variável de cache é atualizada, apenas os núcleos

que contém linhas de cache que armazenam essa variável precisam ser contatados.

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: um dado que foi acessado recentemente, é provável que ele seja acessado novamente em um breve espaço de tempo. Muitas vezes um programa realiza operações repetitivas que acessam um mesmo dado, caso esse dado esteja na memória cache ou na memória virtual, pode melhorar o desempenho do sistema.

Localidade espacial: itens que estão próximos de itens acessados recentemente têm maior probabilidade de serem acessados em um futuro próximo, pois os dados estão armazenados em blocos contíguos ou linhas de cache, onde as instruções e dados são transferidos entre a memória principal e a cache.

O programador pode evitar armadilhas nas quais, para a localidade temporal a busca de dados na memória principal pode ser minimizada caso o acesso aos dados seja otimizado, utilizando variáveis locais, enquanto para a localidade espacial a organização dos dados deve ser otimizada para matrizes de tamanhos ideais e não utilizar algoritmos que realizem saltos de 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 ou mais programas estão utilizando dados de uma mesma linha de cache, quando ocorre alguma alteração, a linha de cache será marcada como suja, assim quando outra thread fizer o acesso à linha de cache o acesso será negado, pois ela está sendo apontada como suja devido à inconsistência dos dados, isso faz com que o desempenho do programa seja prejudicado, causando mais acessos à memória principal do que o necessário.

Podemos reduzir seu efeito utilizando armazenamento temporário local, sendo uma cache privada para armazenar os resultados do programa, em seguida são copiados os dados do armazenamento temporário para o armazenamento compartilhado assim que o procedimento estiver finalizado. Em caso de falso compartilhamento, a memória principal será acessada uma única vez e não sempre que um dado seja modificado.

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

A lei de Amdahl, afirma que, de grosso modo, a menos que todo o programa serial seja paralelizado, a possível aceleração será muito limitada independentemente do número de núcleos disponíveis. De forma geral, se uma fração r do nosso programa permanecer sem ser paralelo, a lei de Amdahl diz que não podemos obter uma aceleração maior que 1/r independente da quantidade de núcleos utilizados. Porém Amdahl não levou em consideração o tamanho do problema e para sua maioria, à medida em que aumentamos o tamanho do problema a fração "inerentemente serial" do programa diminui de tamanho, com isso surge a lei de Gustafson, que começa a considerar a parte esquecida de Amdahl. Enquanto Amdahl leva em consideração o tempo de execução sequencial para estimar o speedup máximo com múltiplos processadores, Gustafson parte do tempo de execução paralela para estimar o speedup máximo comparado com a execução sequencial.

8. **Como se calcula a eficácia de um programa paralelo?**

Considerando S como o speedup (aceleração resultante da transformação de um programa serial em um programa paralelo) e p seja o número de processadores utilizados, temos:

$$E = \frac{S}{p} = \frac{(\frac{Tserial}{Tparalelo})}{p} = \frac{T_{serial}}{p \cdot T_{paralelo}}$$

A eficiência é calculada a partir da fórmula acima, onde T_{serial} é o tempo de processamento serial, $T_{paralelo}$ seja o tempo de processamento paralelo. A 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 não 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 central de uma amostra de valores, para um programa paralelo a mediana retornará o valor que menos teve interferência com os picos de processamento (outliers) decorrentes da execução do programa e de outros processos.

Mas para threads ou processos o ideal é que o tempo a ser medido seja o tempo mínimo de execução da thread, pois é improvável que algum evento externo possa realmente fazer nosso programa rodar mais rápido do que o melhor tempo de execução possível.

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

Podemos considerar a escalabilidade de um programa paralelo quando aumentamos o número de processadores por um fator k e necessitamos achar um valor x para aumentar o tamanho do problema para que a eficiência permaneça inalterada. Assim podemos classificar em três tipos de escalabilidade:

- Fracamente escalável: um programa é dito fracamente escalável quando aumentamos o número de processadores e o tamanho do problema e a eficiência se mantiver fixa.
- Escalável: um programa é dito escalável quando a eficiência permanece inalterada quando aumentamos o número de processadores e o tamanho do problema na mesma proporção.
- Fortemente escalável: um programa é dito fortemente escalável quando aumentando o número de processadores e conseguimos manter a eficiência sem aumentar o tamanho do problema. (Speedup linear).

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?

A diretiva critical é responsável por garantir que uma parte do código seja executada somente por uma thread por vez para evitar que outras threads sobreponham a escrita do dado, a diretiva bloqueia a seção para outras threads até que a thread que está no bloco critical finalize a sua tarefa.

A condição de corrida é a situação quando uma ou mais threads disputam a ordem de gravação de um dado na variável compartilhada. Ao retirar a diretiva critical, a condição de corrida será mais evidente a partir de 4 threads, onde haverá uma disputa entre as threads para

escrever o resultado obtido na variável, trazendo assim, um valor inconsistente ao valor que deveria ser retornado de 21.333.

A tabela abaixo mostra os resultados obtidos com o uso de 1, 2, 4, 8 e 16 threads para 64, 128, 256 e 512 trapézios. Para 3 threads utilizamos os valores 63, 126, 252 e 504 (valores divisíveis por 3).

Assim, podemos aferir que a quantidade mínima de threads que ativa a condição de corrida é 3, enquanto à quantidade mínima de trapézios que inicia a condição de corrida são 252 trapézios com 3 threads.

n° threads/ n° trapézios	1	2	3	4	8	16
64	21.333	21.333	63 ⇒ 21.333	19.001	16.959	11.522
128	21.333	21.333	126 ⇒ 21.333	15.000	10.375	10.120
256	21.333	21.333	252 ⇒ 6.321	12.666	7.083	15.119
512	21.333	21.333	504 ⇒ 20.543	21.000	15.208	20.197

13. Em nossa primeira tentativa de paralelizar o programa para estimar π, nosso programa estava incorreto. Na verdade, usamos o resultado do programa quando ele foi executado com uma única thread como evidência de que o programa rodado com duas threads estava incorreto. Explique por que nós poderíamos 'confiar' no resultado do programa quando ele foi executado com uma única thread.

```
1         double factor = 1.0;
2         double sum = 0.0;
3  #      pragma omp parallel for num_threads(thread_count) \
4            reduction(+:sum)
5         for (k = 0; k < n; k++) {
6            sum += factor/(2*k+1);
7            factor = -factor;
8         }
9         pi_approx = 4.0*sum;
```

O programa acima apresenta uma instância de dependência de loop-carried, onde o valor de k pode ser adotado por uma thread e a iteração k+1 ser adotada por outra thread, assim não há garantias de que o valor de *factor* seja o correto, gerando assim uma condição de corrida no qual uma das threads podem atualizar a variável *factor* e resultar em um resultado incorreto.

Com base nisso, podemos confiar no resultado da execução com apenas uma thread pois não há a possibilidade de acontecer a dependência carregada de loop, pois há apenas uma thread durante a execução do programa. A diretiva *parallel for* não irá surtir efeito (somente uma thread) e o programa seria executado como um programa serial, visto que há somente uma thread.

14. Considere o loop

```
a[0] = 0;
for (i = 1; i < n; i++)
   a[i] = a[i-1] + i;
```

Há claramente uma dependência que persiste ao longo do loop, já que o valor de a[i] não pode ser calculado sem o valor de a[i-1]. Você consegue ver uma maneira de eliminar essa dependência e paralelizar o loop?

Para essa questão, devemos primeiramente entender a dependência carregada no loop, podemos ver que o valor de a[i] depende sempre do valor anterior a[i-1] somado com o valor atual da variável i, sendo assim, temos a seguinte situação:

$i = 1;\ a[1] = a[0] + 1 = 1;\ a[1] = 1$

$i = 2;\ a[2] = a[1] + 2 = 3;\ a[2] = 3$

$i = 3;\ a[3] = a[2] + 3 = 6;\ a[3] = 6$

$i = 4;\ a[4] = a[3] + 4 = 10;\ a[4] = 10$

E assim por diante, com isso, podemos ver que o loop é uma sequência conhecida como fórmula da soma aritmética, e que deve ser substituída por:

$$a[i] = \frac{i(i+1)}{2}$$

O código paralelizado utilizando as diretivas do parallel for, default, private e shared

```
#pragma omp parallel for num_threads(thread_count) default(none) \
private(i) shared(a, n)
   for (i = 1; i < n; i++)
      a[i] = i * (i + 1) / 2;
```

A diretiva parallel for é usada para distribuir as iterações do laço for entre as threads.

A diretiva default é para desativar a suposição padrão e declarar explicitamente quem são as variáveis privadas e compartilhadas.

A diretiva private é para que cada thread tenha uma cópia privada da variável i.

A diretiva shared é para que todas as threads possam acessar a mesma cópia das variáveis a e n.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main(int argc, char *argv[])
{
   int thread_count = 2;
   int n = 10;
   int a[n];
   int i;

   a[0] = 0;

#pragma omp parallel for num_threads(thread_count) default(none) private(i) shared(a, n)
   for (i = 1; i < n; i++)
   {
      a[i] = i * (i + 1) / 2;
      printf("a[%d] = %d\n", i, a[i]);
   }

} /* main */
```

```
luizdts@Luiz:~/pcd$ gcc -g -Wall -fopenmp -o q14 q14.c
luizdts@Luiz:~/pcd$ ./q14
a[6] = 21
a[7] = 28
a[8] = 36
a[9] = 45
a[1] = 1
a[2] = 3
a[3] = 6
a[4] = 10
a[5] = 15
luizdts@Luiz:~/pcd$
```