Lista de Exercícios

1 Porque computação paralela?

1. Suponha que precisamos calcular *n* valores e somá-los. Suponha que também tenhamos *p* núcleos e *p* seja muito menor que *n*. Então cada núcleo pode calcular uma soma parcial de aproximadamente valores *n/p* da seguinte maneira:

```
minha_soma = 0;
meu_pri_i = . . . ;
meu_ult_i = . . . ;
        for (meu_i = meu_pri_i; meu_i < meu_ult_i; meu_i+=meu_desl) {
           meu_x = Compute_prox_valor(. . .);
           minha_soma += meu_x;
        }
```

Aqui o prefixo *meu_* indica que cada núcleo está usando suas próprias variáveis priva das e cada núcleo pode executar este bloco de código independentemente dos outros núcleos.

Supondo que cada chamada para *Compute_prox_valor* requer aproximadamente a mesma quantidade de trabalho, elabore fórmulas para calcular *meu_pri_i*, *meu_ult_i* e *meu_desl*. Lembre-se de que cada núcleo deve receber aproximadamente o mesmo número de ele mentos de computação no loop.

Dica: primeiro considere o caso em que *n* é divisível por *p*. A partir daí, elabore fórmulas para o caso em que *n* não é divisível por *p*.

Para n **divisível** por p e p sendo muito menor que n, temos:

- n/p = x
- meu_pri_i do núcleo p[i] = x * i
- meu_ult_i do núcleo p[i] = meu_pri_i [i] + x

Para n **não divisível** por p e p sendo muito menor que n, temos:

- Como n não é divisível por n, sobra um resto de divisão que não pode ser dividido igualmente entre os núcleos, então:
  - resto = n%p
  - A gente tá trabalhando com uma sequência, então, podemos usar essa sequência (valores de p), para dividir o trabalho restante
  - **se i < resto** :
    - meu_pri_i do núcleo p[i] = x*i
    - meu_ult_i do núcleo p[i] = meu_pri_i[i] + x
  - **se i = resto :**
    - meu_pri_i do núcleo p[i] = x* i
    - meu_ult_i do núcleo p[i] = meu_pri_i[i] + x + 1
  - **se i > resto:**
    - meu_pri_i do núcleo p[i] = (x*i) + 1
    - meu_ult_i do núcleo p[i] = meu_pri_i [i] + x + 1

Ex: 11/3 = 3 e resto = 11%3 = 2:

- núcleo 0 = (0,1,2)
  - meu_pri_0 = 0*3 = 0

    - meu_ult_0 = 0 + 3 = 3
- núcleo 1 = (3,4,5,6)
    - meu_pri_1 = 1*3 = 3
    - meu_ult_1 =3+3+1 = 7
- núcleo 2 = (7,8,9,10)
    - meu_pri_2 = 2*3+1 = 7
    - meu_ult_2 =7+3+1 = 11

2. Suponha que precisamos calcular *n* valores e somá-los. Suponha que também tenhamos *p* núcleos e *p* seja muito menor que *n*. Então cada núcleo pode calcular uma soma parcial de aproximadamente valores *n/p* da seguinte maneira:

```
minha_soma = 0;
meu_pri_i = . . . ;
meu_ult_i = . . . ;
for (meu_i = meu_pri_i; meu_i < meu_ult_i; meu_i+=meu_desl) {
   meu_x = Compute_prox_valor(. . .);
   minha_soma += meu_x;
}
```

Aqui o prefixo *meu_* indica que cada núcleo está usando suas próprias variáveis priva das e cada núcleo pode executar este bloco de código independentemente dos outros núcleos.

Considerando que a chamada com *i* = *k* requer *k*+1 vezes mais trabalho que a chamada com *i* = 0 (se a chamada com *i* = 0 requer 2 milissegundos, a chamada com *i* = 1 requer 4, a chamada com *i* = 2 requer 6...), elabore fórmulas para calcular *meu_pri_i*, *meu_ult_i* e *meu_desl*. Lembre-se de que cada núcleo deve receber aproximadamente o mesmo número de elementos de computação no loop.

Escalonamento circular: Os núcleos recebem trabalhos em turnos ao invés de sequência.

- Todos os processos são colocados em uma fila circular.
- O primeiro processo na fila recebe a CPU e é executado por um período de tempo igual ao quantum.
- Após o quantum expirar, se o processo não terminar, ele é movido para o final da fila.
- O próximo processo na fila agora recebe a CPU e é executado pelo mesmo período de tempo.
- Este processo continua até que todos os processos tenham sido executados.

i = k + 1

2*((0+1)+(1+1)+(2+1)+(3+1)...k+1)

EX: n = 12 e p 3 -> 12/3 = 4 iterações para cada núcleo

- núcleo 0: 0, 5, 6, 11 = 52 ms
- núcleo 1: 1, 4, 7, 10 = 54 ms
- núcleo 2: 2, 3, 8, 9 = 56 ms

O escalonamento circular é justo porque cada processo recebe uma chance igual de usar a CPU. No entanto, pode ser ineficiente se o quantum for muito grande, pois pode haver muita espera entre as execuções dos processos. Se o quantum for muito pequeno, o

overhead de context switch (mudança de contexto) pode ser alto.

3. Escreva um pseudocódigo para uma soma global estruturada em árvore. Suponha que o número de núcleos seja uma potência de dois.
p = total de processadores
np = número do processador atual
divisor = 2 (potência de dois)

```
para cada (diferença_do_core=1; enquanto diferença_do_core< p; passo diferença_do_core *=2){
        se(np%divisor == 0){
                meuPar = np +1;
                recebeESoma(meuPar);
                }
        senao{meuPar = np -1;
                enviarParaSomar(meuPar);
                }
                divisor *=2;
        }
```

4. Podemos usar os operadores bit a bit de C para implementar uma soma global estru turada em árvore. Implemente este algoritmo em pseudocódigo usando o operador OU EXCLUSIVO e o operador *shift* para esquerda.

Considerando que temos 8 núcleos: bitmask -> serve para determinar qual bit vai ser trocado em uma operação XOR(exclusivo). Também indica o deslocamento entre núcleos de acordo com seu valor decimal.

primeiro estágio:

bitmask: 001
000 = 0 > 001
001 = 1 > 000
010 = 2 > 011
011 = 3 > 010
100 = 4 > 101
101 = 5 > 100
110 = 6 > 111
111 = 7 > 110

segundo estágio: bitmask: 010; desconsiderado = aaaa
000 = 0 > 001 > 010
001 = 1 > 000
010 = 2 > 011 > 000
011 = 3 > 010
100 = 4 > 101 > 110
101 = 5 > 100
110 = 6 > 111 > 100
111 = 7 > 110

terceiro estágio: bitmask: 100; desconsiderado = aaaa
000 = 0 > 001 > 010 > 100
001 = 1 > 000
010 = 2 > 011 > 000
011 = 3 > 010
100 = 4 > 101 > 110 > 000
101 = 5 > 100
110 = 6 > 111 > 100
111 = 7 > 110

PSEUDOCÓDIGO:

```
encontrarNucleosEmparelhados (id, estagio){
        nucleoEmparelhado = null;
                se id % (2^estágio) - 1 == 0 {
                        então bitMask = 1 <<(estágio - 1)
                        001 << (1-1) =001; 001<<(2-1) = 010; 001<<(3-1) = 100
                        bitValue = (id & bitmask) >> (estágio - 1)  /*
                        A operação "(id & bitmak) >> (estágio - 1)" pega um único bit na
```

posição (estágio - 1) do resultando do bitwise de (id & bitmask) e atribui o valor do bit ao bitValue (0 ou 1). Em teoria seria assim, mas ele aplica ao bitValue qualquer valor, em caso de ser 0, significa que o bit da posição é 0, se não, qualquer outro valor indica um bit de valor 1.

se bitValue == 0 então {

se a condição for verdadeira, significa que a operação id & bitmask resultou em um valor no qual o bit correspondente à bitmask é 0

nucleoEmparelhado = id | bitmask }

senão{

então nucleoEmparelhado = id & ~bitmask }

vai executar a mesma operação, mas o símbolo de negação em ~bitmask inverte todos os bits, (0 para 1 e 1 para 0)

return nucleoEmparelhado}

5. Escreva um pseudocódigo para uma soma global estruturada em árvore. Considere que o número de núcleos pode não ser uma potência de dois.

```
divisor = 2
para (diferença_dos_nucleos = 1; enquanto diferença_dos_nucleos < numero_processadores
;
        diferença_dos_nucleos *=2){
        se (nucleo%divisor == 0){
                se (nucleo = numero_processadores){
                        meuPar = nucleo - 1;
                        enviarParaSomar(meuPar);
                        quebra;
        senao {
                meuPar = nucleo+1;
                receberParaSomar(meuPar);}}
        senao {
                meuPar = nucleo-1;
                enviarParaSomar(meuPar);}
        divisor *=2;
        }
```

O código da questão anterior não precisa de modificação por estarmos trabalhando com a troca de bits de forma individual, então dessa forma o resultado produzido será o mesmo independente de estar ou não trabalhando com uma quantidade de núcleos que é uma potência de dois.

6. Elabore fórmulas para o número de recebimentos e adições que o núcleo 0 realiza usando

(a) uma soma global onde apenas o núcleo 0 realiza as adições;

(b) uma soma global estruturada em árvore.

Faça uma tabela mostrando os números de recebimentos e adições realizados pelo nú cleo 0 quando as duas somas são usadas com 2, 4, 8, · · · , 1024 núcleos.

a) A quantidade de adições será um total de a quantidade total de núcleos - 1

TotalAdições = QntdNucleos - 1;

b) Para o código em árvore, baseado na complexidade de uma árvore AVL, será log2(n), sendo n o número de núcleos a serem emparelhados

Núcleos	código original	código em árvore
2	1	1
4	3	2
8	7	3
16	15	4
32	31	5
64	63	6
128	127	7
254	253	8
512	511	9
1024	1023	10

7. Descreva um problema de pesquisa em sua área (Tecnologia de Informação, Engenharia de Software...) que se beneficiaria com o uso da computação paralela. Forneça um esboço de como o paralelismo seria usado. Você usaria paralelismo de tarefas ou de dados? Por quê?

Processamento de consultas em bancos de dados. Tradicionalmente, essas consultas são processadas sequencialmente, o que pode ser ineficiente, especialmente quando lidamos com grandes volumes de dados.

A computação paralela pode ser uma solução eficaz para este problema.

Cada consulta poderia ser dividida em subconsultas menores que poderiam ser processadas simultaneamente por diferentes núcleos de CPU. Além disso, a computação paralela também poderia ser usada para distribuir o processamento de consultas em vários servidores.

O paralelismo de tarefas parece ser a abordagem mais adequada para este problema, pois permite que as consultas sejam processadas simultaneamente, melhorando a eficiência e a velocidade do processamento de consultas.

2 *Hardware* e *software* paralelos

8. Quando discutimos a adição de ponto flutuante, partimos da suposição simplificadora de que cada uma das unidades funcionais levava o mesmo tempo. Suponha que buscar (*fetch*) e armazenar (*store*) levem 2 nanossegundos cada e que as operações restantes levem 1 nanossegundo cada.

(a) Quanto tempo leva uma adição de ponto flutuante com essas suposições?

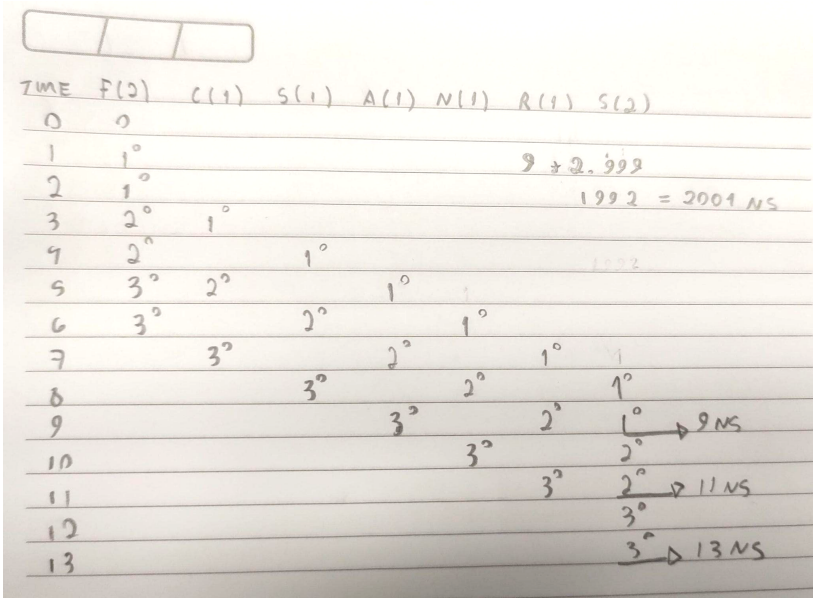
fetch (2), compare(1), shift(1), add(1), normalize(1),round(1),Store(2) = 2*2+5 = 9

(b) Quanto tempo levará uma adição sem pipeline de 1.000 pares de *floats* com essas suposições?

Uma adição sem pipeline de 1000 pares floats custará 9000 nanossegundos

(c) Quanto tempo levará uma adição em pipeline de 1.000 pares de *floats* com essas suposições?

A primeira adição, mais todo o restante da operação vezes o tempo necessário pra segunda terminar em comparação com a primeira. 9+2*999=2001 NS



(d) O tempo necessário para busca e armazenamento pode variar consideravelmente se os operandos/resultados forem armazenados em diferentes níveis da hierarquia de memória. Suponha que uma busca de um cache de nível 1 leve 2 nanossegundos, enquanto uma busca de um cache de nível 2 leve 5 nanossegundos e uma busca da memória principal leve 50 nanossegundos. O que acontece com o pipeline quando há uma falha de cache de nível 1 na busca de um dos operandos? O que acontece quando há uma falha de nível 2?

Quando há uma falha de cache de nível 1, a operação precisará ser refeita após recuperar os dados do cache de nível 2 ou da memória principal. Portanto, o tempo necessário é 5 (buscar do cache de nível 2) + 5 (tempo para a operação de adição) + 2 (tempo para armazenar o resultado) = 12 nanossegundos.

Quando há uma falha de nível 2, a operação precisará ser refeita após recuperar os dados da memória principal. Portanto, o tempo necessário é 50 (tempo para buscar da memória principal) + 5 (tempo para a operação de adição) + 2 (tempo para armazenar o resultado) = 57 nanossegundos.

9. Explique como uma fila implementada em hardware na CPU poderia ser usada para melhorar o desempenho de um *cache write-through*.

Uma fila implementada em hardware na CPU pode gerenciar as operações de escrita direta de forma eficiente ao escrever na memória principal. Só que quando há uma operação de escrita, os dados são atualizados na memória cache e na principal, porém, devido a diferença de velocidade de acesso às memórias pode ocorrer um atraso nessa abordagem.

Com a implementação de uma fila em hardware na CPU, as operações podem ser enfileiradas antes de serem enviadas para a memória principal, e enquanto essas informações são escritas/processadas para a memória principal, a memória cache está livre para realizar outras operações e processamentos.

De modo resumido, a fila colabora como uma espécie de buffer, melhorando o desempenho

do sistema.

10. Dado o exemplo a seguir envolvendo leituras de cache de um array bidimensional.

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
// Initializa A e x; y = 0
. . .
// Primeiro par de loops/
for (i = 0; i < MAX ; i++)
    for (j = 0; j < MAX; j++)
        y [i] += A[i][j]x[j];
. . .
// y = 0
. . .
// Segundo par de loops
for (j = 0; j < MAX ; j++)
    for (i = 0; i < MAX; i++)
        y [i] += A[i][j]x[j];
```

(a) Como o aumento do tamanho da matriz afetaria o desempenho dos dois pares de loops aninhados?

Os arrays são armazenados em ordem de linha, sempre que aumenta o i ou o j no par de loops, a linha/coluna é pulada para a próxima na memória, e os valores anteriores não estarão mais na cache para a próxima iteração. Isso resulta em falha de cache e atrasa o resultado. Já que a cache só consegue armazenar duas linhas da matriz (ou oito elementos da matriz de modo sequencial), só conseguimos armazenar 2 elementos por vez de uma coluna.

(b) Como o aumento do tamanho da cache afetaria o desempenho dos dois pares de loops aninhados?

Uma cache maior pode conter mais dados da matriz A na memória, o que reduziria o número de falhas de cache e melhoraria o desempenho. No entanto, se o tamanho da cache for muito grande em relação ao tamanho da matriz, muitos dados da matriz podem não caber na cache, resultando em mais falhas de cache. No primeiro par de loops o aumento da cache iria diminuir ou tornar nula a quantidade de falhas, mas na segunda, como estamos lendo os dados por coluna e a cache usa localidade espacial, independente do tamanho da cache ainda irá ocorrer a mesma quantidade de falhas.

(c) Quantas falhas ocorrem nas leituras de *A* no primeiro par de loops aninhados?

A quantidade de falhas do primeiro par de loops será igual a sua quantidade de linhas, dado que a memória cache usa localidade espacial e quando buscamos o primeiro elemento de uma linha, trazemos todos os outros elementos daquela linha até o tamanho máximo da linha de cache e ao acessar de modo sequencial só teremos um novo cache miss ao acessar uma nova linha.

(d) Quantas falhas ocorrem no segundo par?

Para o segundo par de loops, seria gerado uma para cada novo elemento que fossemos

ler. Apesar do cache utilizar a localidade espacial que ao procurar um elemento ele também carrega todos os outros elementos da linha de forma subsequente, já que estamos lendo por colunas, sempre que um novo elemento for lido a linha da cache ficará cheia.

Dessa forma ao pedir um novo elemento iremos apagar linhas anteriores e ao procurar por elas de novo em uma nova busca de elemento por coluna, que resulta em falha.

11. Suponha que uma linha de cache consiga armazenar quatro elementos de *A*.

(a) A adição de cache e memória virtual a um sistema von Neumann altera sua designação como sistema SISD?

A Arquitetura de von Neumann é uma arquitetura de computador que se caracteriza pela possibilidade de uma máquina digital armazenar seus programas no mesmo espaço de memória que os dados, podendo assim manipular tais programas.

Nesse caso, não vai alterar em nada já que ela trabalha apenas com uma instrução e unidade de dados de cada vez, adicionar esses dois componentes apenas iria tornar isso mais rápido, provendo maior desempenho.

A adição de cache e memória virtual melhora a eficiência do sistema, mas não muda a natureza fundamental do sistema, que é SISD (Single Instruction, Single Data).

(b) E quanto à adição de pipeline?

Pipeline é um método usado em processadores para executar múltiplas instruções simultaneamente. A técnica melhora o desempenho do sistema e funciona a partir da divisão de uma tarefa em partes menores, que podem ser processadas em conjunto (Single Instruction, Multiple Data).

Adicionar um pipeline ao sistema altera a classificação SISD. Pipelining é uma técnica usada para melhorar a eficiência do sistema ao permitir que várias instruções sejam executadas simultaneamente.

(c) Multiple issues?

O sistema de problemas múltiplos permite que a CPU execute várias instruções simultaneamente, o que pode aumentar significativamente o desempenho, especialmente em cenários onde uma instrução precisa esperar por um recurso, como a leitura de dados da memória.

Neste caso, ela se tornaria uma máquina MISD, devido ao fato de que o multithreading permite trabalhar com várias instruções, tendo em mente que quando uma instrução fica parada ele executa outra tarefa para calcular outra instrução desejada.

(d) Hardware multithreading?

A arquitetura de Von Neumann é uma arquitetura de computador que separa a memória de dados e a memória de instruções. Nesta arquitetura, a unidade central de processamento (CPU) lê instruções de uma memória de instruções e executa operações sobre dados armazenados em uma memória de dados separada.

No entanto, a introdução de multithreading na CPU pode mudar a forma como as instruções são executadas. Com multithreading, a CPU pode executar várias threads de execução simultaneamente, cada uma com seu próprio fluxo de controle e estado. Isso permite que a CPU execute várias instruções de maneira simultânea, melhorando o desempenho,

especialmente em aplicações que podem tirar proveito do paralelismo, e sim, altera.

12. Suponha que um programa deva executar 10^{12} instruções para resolver um problema específico. Suponha ainda que um sistema de processador único possa resolver o problema em 10^6 segundos (cerca de 11, 6 dias). Portanto, em média, o sistema de processador único executa 10^6 instruções por segundo. Agora suponha que o programa tenha sido paralelizado para execução em um sistema de memória distribuída. Suponha também que o programa paralelo usa p processadores e, assim, cada processador executará instruções $10^{12}/p$. Além disso, cada processador deverá enviar $10^9(p - 1)$ mensagens. Por fim, suponha que não haja *overhead* adicional na execução do programa paralelo. Ou seja, o programa será concluído depois que cada processador tiver executado todas as suas instruções e enviado todas as suas mensagens e não haverá atrasos devido a coisas como espera por mensagens.

(a) Suponha que demore 10^{-9} segundos para enviar uma mensagem. Quanto tempo levará para o programa ser executado com 1000 processadores se cada processador for tão rápido quanto o processador único no qual o programa serial foi executado?

O tempo de execução das instruções é de 10^6, cada processador executa 10^12 instruções, com 1000 processadores fica: 10^6/1000 = 10^3 10^12/1000 = 10^9.

Temos que cada processador (dos 1000) consegue executar 10^9 instruções cada em 1000(10^3) segundos.

Cada processador vai executar 10^9 instruções e deverá enviar 10^9*999 (10^12/1000 = 10^9 10^9(1000-1) = 10^9 * 999) mensagens. Se cada mensagem leva 10^-9 segundos, então temos que o tempo total para enviar todas as mensagens é de 10^9*999*10^-9 = 999. Somando os tempos das operações temos: 10^3+999 segundos.

(b) Suponha que demore 10^{-3} segundos para enviar uma mensagem. Quanto tempo levará para o programa rodar com 1000 processadores?

Apenas alterando o tempo que as instruções são enviadas, temos que cada processador ainda vai executar 10^9 instruções e vai enviar 10^9*999 mensagens.

Se cada mensagem agora leva 10^-3 em cada uma das instruções, então temos que o tempo total para enviar todas as mensagens é de 10^9*999*10^-3 = 999000000 segundos.

Somando os tempos das operações temos: 10^3+999000000 segundos.

13.

(a) Suponha que um sistema de memória compartilhada use coerência de cache (*snooping*) e caches *write-back*. Suponha também que o núcleo 0 tenha a variável x em seu cache e execute a atribuição $x = 5$. Finalmente, suponha que o núcleo 1 não tenha x em seu cache e, após a atualização do núcleo 0 para x, o núcleo 1 tente executar $y = x$. Qual valor será atribuído a y? Por que?

O valor para y será 5. Quando o núcleo 0 altera o valor de x para 5 o bloco do cache do núcleo 1 que contém x é marcado como "dirty"para indicar que o valor em cache é diferente do valor na memória principal. Quando o núcleo tenta executar y = x, já que ele não tem acesso a x em seu cache o "snooping" no barramento detecta que o valor de x foi modificado pelo núcleo 0 e o cache de x para 1 está desatualizado e nisso entra a coerência de cache,

onde o núcleo 1 precisará buscar o valor de x no cache do núcleo 0.

(b) Suponha que o sistema de memória compartilhada da parte anterior utilize um pro tocolo baseado em diretório. Qual valor será atribuído a *y*? Por que?

O valor continuará sendo 5. Quando o núcleo 0 altera o valor de x para 5, ele envia o status do bloco onde se encontra a variável x para o diretório. Quando o núcleo 1 fizer y = x, ele irá consultar no diretório e verificar onde se encontra o valor mais atualizado da variável x, sendo no núcleo 0. Ele obtém o valor de x que é 5 e y será 5.

(c) Como os problemas encontrados nas duas primeiras partes podem ser resolvidos?

Os problemas encontrados nas partes (a) e (b) são principalmente relacionados à consistência dos dados em sistemas de memória compartilhada. Eles podem ser resolvidos através de várias técnicas de sincronização e controle de concorrência, como:

Coerência de Cache: Utilizar protocolos de coerência de cache, como snooping e write-through/write-back, para garantir que todos os caches estejam atualizados com os valores mais recentes das variáveis.

Protocolos de Diretório: Implementar protocolos baseados em diretório que mantêm registros de quem possui uma cópia válida de cada bloco de memória e permitem que os núcleos solicitem o valor mais recente de uma variável quando necessário.

Barramentos de Memória: Utilizar barramentos de memória dedicados para comunicação entre núcleos, que podem ser configurados para transmitir apenas informações críticas de sincronização e evitar a transmissão de dados desnecessários.

Atomicidade e Ordem de Execução: Garantir operações atômicas e a ordem de execução correta das instruções para evitar condições de corrida e inconsistências de dados.

Mecanismos de Bloqueio: Implementar mecanismos de bloqueio, como locks e semáforos, para controlar o acesso a recursos compartilhados e prevenir conflitos.

Cache Coherence Protocols: Utilizar protocolos avançados de coerência de cache, como MESI (Modified, Exclusive, Shared, Invalid) ou MOESI, que ajudam a manter a consistência dos dados em sistemas multiprocessadores.

Memória Virtual: Utilizar memória virtual para abstrair a localização física dos dados e permitir que os sistemas operacionais gerenciam a consistência dos dados de forma transparente aos aplicativos.

14. Sendo *n* o tamanho do problema e *p* o número de *threads* (ou processos):

(a) Suponha que o tempo de execução de um programa sequencial seja dado por $T_{sequencial} = n^2$, onde as unidades do tempo de execução estão em microssegundos. Suponha que uma paralelização deste programa tenha tempo de execução $T_{paralelo} = n^2/p + log_2(p)$. Escreva um programa que encontre as acelerações e efi ciências deste programa para vários valores de *n* e *p*. Execute seu programa com $n = 10, 20, 40, \cdots, 320$ e $p = 1, 2, 4, \cdots, 128$.

• O que acontece com os *speedups* e eficiências à medida que *p* aumenta e *n* é mantido fixo?

• O que acontece quando *p* é fixo e *n* é aumentado?

```
int n = 10, p;
float Tsequencial;
float TParalelo;
float speed;
float efficience;
```

```
for(p = 1; p <= 128; p *= 2 ){
   for(n = 10; n <= 320; n*= 2){
      Tsequencial = pow(n,2);
      TParalelo = pow(n,2)/p+log2(p);
      speed = Tsequencial/TParalelo;
      efficience = speed/p;
      printf("n = %i, p = %i, Tsequencial = %.2f, TParalelo = %.2f, speed = %.2f, efficience =
%.2f \n",n,p,Tsequencial, TParalelo, speed, efficience);
   }
}
```

No primeiro caso, quando p está aumentando e n é fixo, inicialmente o speedup aumenta até que p seja igual a 128 e então começa a diminuir. Já a eficiência do programa começa a diminuir cada vez mais. Já quando p se mantém fixo e o n vai aumentando, o speed começa em um valor e vai aumentando com base nas alterações de n e o mesmo serve para a eficiência.

(b) Suponha que $T_{paralelo} = T_{sequencial}/p + T_{overhead}$. Suponha também que mantemos p fixo e aumentamos o tamanho do problema.

• Mostre que, se $T_{overhead}$ crescer mais lentamente que $T_{sequencial}$, a eficiência pa ralela aumentará à medida que aumentarmos o tamanho do problema.

```
int n , p = 2;
float Tsequencial;
float TParalelo;
float speed;
float efficience;
float Toverhead;
//for(p = 1; p <= 512; p *= 2 ){
   for(n = 10; n <= 320; n*= 2){
      Toverhead = n*2;
      Tsequencial = pow(n,2);
      TParalelo = Tsequencial/p+Toverhead;
      speed = Tsequencial/TParalelo;
      efficience = speed/p;
      printf("n = %i, p = %i, Tsequencial = %.2f, TParalelo = %.2f, Toverhead = %.2f, speed
= %.2f, efficience = %.2f \n",n,p,Tsequencial, TParalelo, Toverhead, speed, efficience);
   }
//}
```

saida no terminal para quando aumentamos o tamanho do problema (n) e p = 1

```
n = 10, p = 1, Tsequencial = 100.00, TParalelo = 120.00, Toverhead = 20.00, speed = 0.83,
efficience = 0.83
n = 20, p = 1, Tsequencial = 400.00, TParalelo = 440.00, Toverhead = 40.00, speed = 0.91,
efficience = 0.91
n = 40, p = 1, Tsequencial = 1600.00, TParalelo = 1680.00, Toverhead = 80.00, speed =
0.95, efficience = 0.95
n = 80, p = 1, Tsequencial = 6400.00, TParalelo = 6560.00, Toverhead = 160.00, speed =
0.98, efficience = 0.98
n = 160, p = 1, Tsequencial = 25600.00, TParalelo = 25920.00, Toverhead = 320.00, speed =
0.99, efficience = 0.99
n = 320, p = 1, Tsequencial = 102400.00, TParalelo = 103040.00, Toverhead = 640.00,
```

speed = 0.99, efficience = 0.99

saida no terminal para quando aumentamos o tamanho do problema (n) e p = 2

n = 10, p = 2, Tsequencial = 100.00, TParalelo = 70.00, Toverhead = 20.00, speed = 1.43, efficience = 0.71
n = 20, p = 2, Tsequencial = 400.00, TParalelo = 240.00, Toverhead = 40.00, speed = 1.67, efficience = 0.83
n = 40, p = 2, Tsequencial = 1600.00, TParalelo = 880.00, Toverhead = 80.00, speed = 1.82, efficience = 0.91
n = 80, p = 2, Tsequencial = 6400.00, TParalelo = 3360.00, Toverhead = 160.00, speed = 1.90, efficience = 0.95
n = 160, p = 2, Tsequencial = 25600.00, TParalelo = 13120.00, Toverhead = 320.00, speed = 1.95, efficience = 0.98
n = 320, p = 2, Tsequencial = 102400.00, TParalelo = 51840.00, Toverhead = 640.00, speed = 1.98, efficience = 0.99

• Mostre que se, por outro lado, $T_{overhead}$ crescer mais rápido que $T_{sequencial}$, a eficiência paralela diminuirá à medida que aumentamos o tamanho do problema.

```
int n , p = 2;
float Tsequencial;
float TParalelo;
float speed;
float efficience;
float Toverhead;
//for(p = 1; p <= 512; p *= 2 ){
   for(n = 10; n <= 320; n*= 2){
      Toverhead = pow(n,2);
      Tsequencial = n*2;
      TParalelo = Tsequencial/p+Toverhead;
      speed = Tsequencial/TParalelo;
      efficience = speed/p;
      printf("n = %i, p = %i, Tsequencial = %.2f, TParalelo = %.2f, Toverhead = %.2f, speed = %.2f, efficience = %.2f \n",n,p,Tsequencial, TParalelo, Toverhead, speed, efficience);
   }
//}
```

saida no terminal para quando aumentamos o tamanho do problema (n) e p = 1

n = 10, p = 1, Tsequencial = 20.00, TParalelo = 120.00, Toverhead = 100.00, speed = 0.17, efficience = 0.17
n = 20, p = 1, Tsequencial = 40.00, TParalelo = 440.00, Toverhead = 400.00, speed = 0.09, efficience = 0.09
n = 40, p = 1, Tsequencial = 80.00, TParalelo = 1680.00, Toverhead = 1600.00, speed = 0.05, efficience = 0.05
n = 80, p = 1, Tsequencial = 160.00, TParalelo = 6560.00, Toverhead = 6400.00, speed = 0.02, efficience = 0.02
n = 160, p = 1, Tsequencial = 320.00, TParalelo = 25920.00, Toverhead = 25600.00, speed = 0.01, efficience = 0.01
n = 320, p = 1, Tsequencial = 640.00, TParalelo = 103040.00, Toverhead = 102400.00, speed = 0.01, efficience = 0.01

saida no terminal para quando aumentamos o tamanho do problema (n) e p = 2
n = 10, p = 2, Tsequencial = 20.00, TParalelo = 110.00, Toverhead = 100.00, speed = 0.18, efficience = 0.09
n = 20, p = 2, Tsequencial = 40.00, TParalelo = 420.00, Toverhead = 400.00, speed = 0.10, efficience = 0.05
n = 40, p = 2, Tsequencial = 80.00, TParalelo = 1640.00, Toverhead = 1600.00, speed = 0.05, efficience = 0.02
n = 80, p = 2, Tsequencial = 160.00, TParalelo = 6480.00, Toverhead = 6400.00, speed = 0.02, efficience = 0.01
n = 160, p = 2, Tsequencial = 320.00, TParalelo = 25760.00, Toverhead = 25600.00, speed = 0.01, efficience = 0.01
n = 320, p = 2, Tsequencial = 640.00, TParalelo = 102720.00, Toverhead = 102400.00, speed = 0.01, efficience = 0.00

A eficiência S/p diminui com base no aumento de p (processadores) devido ao overhead (sobrecarga), porque a coordenação e a troca de informações entre os processadores ficarão maiores e aumenta também a latência, gerando sobrecargas no sistema paralelo, a disputa por memória entre a quantidade de processadores também afeta a eficiência.

15. Às vezes, diz-se que um programa paralelo que obtém um *speedup* maior que *p* (o nú mero de processos ou *threads*) tem *speedup* superlinear. No entanto, muitos autores não consideram os programas que superam as "limitações de recursos"como tendo acelera ção superlinear. Por exemplo, um programa que deve usar armazenamento secundário para seus dados quando é executado em um sistema de processador único pode ser capaz de acomodar todos os seus dados na memória principal quando executado em um grande sistema de memória distribuída. Dê outro exemplo de como um programa pode superar uma limitação de recursos e obter *speedups* maiores que *p*.

Um exemplo de como um programa pode superar uma limitação de recursos é com o uso eficiente de recursos em diferentes ambientes de execução. Por exemplo, um programa que enfrenta limitação de recursos como armazenamento secundário, ao ser executado em um sistema de processador único.

Nesse caso, o programa vai ser obrigado a armazenar parte dos dados em armazenamento secundário, resultando em grande perda de desempenho. Porém, se executarmos o mesmo programa em um grande sistema de memória distribuída, onde há mais memória disponível, a limitação de armazenamento secundário pode ser superada. Neste caso, o programa obteve um speedup maior que p ao migrar para um ambiente onde uma limitação de recursos foi superada.

16. Sendo n o tamanho do problema e p o número de *threads* (ou processos), suponha $T_{serial} = n$ e $T_{parallel} = n/p + log_2(p)$, onde os tempos estão em microssegundos. Se aumentarmos p por um fator de k, encontre uma fórmula para quanto precisaremos au mentar n para manter a eficiência constante.

(a) Quanto devemos aumentar n se dobrarmos o número de processos de 8 para 16? (b) O programa paralelo é escalável?

17. Um programa que obtém *speedup* linear é fortemente escalável? Explique sua

resposta.

Sim, um programa com speedup linear é fortemente escalável devido ao fato de que ele consegue ter um aumento de processadores e distribuir o trabalho entre os núcleos de maneira eficiente.

18. Bob tem um programa que deseja cronometrar com dois conjuntos de dados, *input_data1* e *input_data2*. Para ter uma ideia do que esperar antes de adicionar funções de tempo rização ao código de seu interesse, ele executa o programa com os dois conjuntos de dados e o comando shell Unix *time*:

```
$ time ./bobs_prog < input_data1
real 0m0.001 s
user 0m0.001 s
sys 0m0.000 s

$ time ./bobs_prog < input_data2
real 1m1.234 s
user 1m0.001 s
sys 0m0.111 s
```

A função de temporização que Bob está usando tem resolução de milissegundos.

(a) Bob deveria usá-la para cronometrar seu programa com o primeiro conjunto de dados? Por que?

(b) E quanto ao segundo conjunto de dados? Por que?

19. Escreva um pseudocódigo para a soma global estruturada em árvore para somar vetores.

(a) Primeiro considere como isso pode ser feito em um ambiente de memória compartilhada.
• Quais variáveis são compartilhadas e quais são privadas?
(b) Depois considere como isso pode ser feito em um ambiente de memória distribuída.