Bruno Victor Paiva Da Silva - 2020011045

Exercícios PCD:

Seção "1.11 EXERCISES": 1-6, 9; (7 questões)

1.1 Desenvolva fórmulas para as funções que calculam meu primeiro i e meu último i no exemplo da soma global. Lembre-se de que cada núcleo deve receber aproximadamente o mesmo número de elementos de cálculos no loop. Dica: considere primeiro o caso em que n é divisível por p.

```
/* Tamanho do bloco de interações para cada thread */
chuksize = n / p;

/* cálculo do resto */
resto = n % p;
```

```
/* Se o rank for menor que o resto */
se (my_rank < resto) {
my_n_count = chuksize + 1;
my_first_i = my_rank * my_n_count;
} senão {
my_n_count = chuksize;
my_first_i = my_rank * my_n_count + resto;
}
my_last_i = my_first_i + my_n_count;
```

1.2 Assumimos implicitamente que cada chamada para Compute next value requer aproximadamente a mesma quantidade de trabalho que as outras chamadas. Como você mudaria sua resposta para a pergunta anterior se a chamada i = k requer k + 1 vezes mais trabalho do que a chamada com i = 0? Portanto, se a primeira chamada (i = 0) requer 2 milissegundos, a segunda chamada (i = 1) requer 4, a terceira (i = 2) requer 6 e assim por diante.

A melhor abordagem seria uma atribuição cíclica utilizando o caminho reverso para as threads como no exemplo a seguir:
Exemplo: n = 8; e p = 3;

Núcleos	Tarefas			Tempo (milissegundos)
Core 0	8	3	2	32
Core 1	7	4	1	30
Core 2	6	5	0	28

E caso não fosse divisível, n = 8 e p = 3;

Núcleos	Tarefas			Tempo (milissegundos)
Core 0	7	4	1	28
Core 1	6	3	0	30
Core 2	5	2		14

1.3 Tente escrever um pseudocódigo para a soma global estruturada em árvore ilustrada na Figura 1.1. Suponha que o número de núcleos seja uma potência de dois (1, 2, 4, 8, . . .).

Dicas: Use um divisor de variável para determinar se um núcleo deve enviar sua soma ou receber e somar. O divisor deve começar com o valor 2 e ser dobrado após cada iteração. Use também uma diferença de núcleo variável para determinar qual núcleo deve ser associado ao núcleo atual. Deve começar com o valor 1 e também ser dobrado após cada iteração. Por exemplo, na primeira iteração 0 % divisor = 0 e 1 % divisor = 1, então 0 recebe e soma, enquanto 1 envia. Também na primeira iteração 0 + diferença de núcleo = 1 e 1 − diferença de núcleo = 0, então 0 e 1 são pareados na primeira iteração.

```
div = 2;
core_difference = 1;
sum = my_value;

while ( div <= number of cores ) {

if ( my_rank % divisor == 0 ) {
parceiro = my_rank + core_difference;
recebe o valor do core parceiro;
sum += recebe o valor;
} else {
Parceiro = my_rank - core_difference;
Enviar a soma para o código parceiro;
}
div *= 2;
core_difference *=2;
}
```

1.4 Como alternativa à abordagem descrita no problema anterior, podemos usar os operadores bit a bit de C para implementar a soma global estruturada em árvore. Para ver como isso funciona, é útil anotar a representação binária (base 2) de cada uma das classificações principais e observar os paresdurante

	Stages		
Cores	*1*	*2*	*3*
$0_{10} = 000_2$	$1_{10} = 001_2$	$2_{10} = 010_2$	$4_{10} = 100_2$
$1_{10} = 001_2$	$0_{10} = 000_2$	×	×
$2_{10} = 010_2$	$3_{10} = 011_2$	$0_{10} = 000_2$	×
$3_{10} = 011_2$	$2_{10} = 010_2$	×	×
$4_{10} = 100_2$	$5_{10} = 101_2$	$6_{10} = 110_2$	$0_{10} = 000_2$
$5_{10} = 101_2$	$4_{10} = 100_2$	×	×
$6_{10} = 110_2$	$7_{10} = 111_2$	$4_{10} = 100_2$	×
$7_{10} = 111_2$	$6_{10} = 110_2$	×	×

cada estágio:

Na tabela, vemos que durante o primeiro estágio cada núcleo é emparelhado com o núcleo cuja classificação difere no primeiro bit ou mais à direita. Durante o segundo estágio, os núcleos que continuam são emparelhados com o núcleo cuja classificação difere no segundo bit e, durante o terceiro estágio, os núcleos são emparelhados com o núcleo cuja classificação difere no terceiro bit. Assim, se tivermos uma bitmask de valor binário que é 0012 para o primeiro estágio, 0102 para o segundo e 1002 para o terceiro, podemos obter a classificação do núcleo com o qual estamos emparelhados "invertendo" o bit em nossa classificação que é diferente de zero em bitmask. Isso pode ser feito usando o operador bit a bit exclusivo ou ∧.
Implemente esse algoritmo em pseudocódigo usando o bit a bit exclusivo ou e o operador de deslocamento à esquerda.

```
bitmask = 1;
div = 2;
sum = my_value;
while ( bitmask < number of cores ) {
parceiro = my_rank ^ bitmask;
if ( my_rank % div == 0 ) {
recebe o valor do código parceiro;
sum += recebe o valor;
} else {
enviar a soma para o código parceiro;
}
bitmask <<= 1;
```

```
div *= 2;
}
```

1.5 O que acontece se seu pseudocódigo no Exercício 1.3 ou no Exercício 1.4 for executado quando o número de núcleos não for uma potência de dois (por exemplo, 3, 5, 6, 7)? Você pode modificar o pseudocódigo para que funcione corretamente independentemente do número de núcleos?

Pode acontecer de alguns núcleos esperarem que núcleos inexistentes enviem valores, e isso provavelmente faria com que o código travasse ou falhasse. Podemos simplesmente adicionar uma condição,

```
if (parceiro  < número de cores) {
    recebe o valor
    sum += recebe o valor
}
```

Quando um núcleo tenta receber um valor de seu parceiro para garantir que o programa irá lidar com o caso em que o número de núcleos não é uma potência de 2.

1.6 Derive fórmulas para o número de recebimentos e adições que o núcleo 0 realizá usando

a. o pseudo-código original para uma soma global, e

b. a soma global estruturada em árvore.

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

a. O número de recebimentos é p − 1, e o número de adições é p − 1.

b. O número de recebimentos é log2(p) e o número de adições é log2(p)

(c)

p	Original	Tree-Structured
2	1	1
4	3	2
8	7	3
16	15	4
32	31	5
64	63	6
128	127	7
256	255	8
512	511	9
1024	1023	10

1.9 Escreva um ensaio descrevendo um problema de pesquisa em seu curso que se beneficiaria do uso da computação paralela. Forneça um esboço aproximado de como o paralelismo seria usado. Você usaria paralelismo de tarefas ou dados?

A área de Inteligência Artificial (IA) tem ganhado destaque nas últimas décadas devido ao seu potencial para revolucionar diversas áreas, incluindo a medicina. Uma aplicação significativa é o uso de redes neurais profundas para o reconhecimento de padrões em imagens médicas, auxiliando no diagnóstico precoce de doenças. No entanto, com o crescimento contínuo da complexidade dos modelos de redes neurais e o aumento das resoluções das imagens, os desafios computacionais associados a essa tarefa também têm aumentado substancialmente. É nesse contexto que a computação paralela pode desempenhar um papel crucial para acelerar o processo de treinamento e inferência de redes neurais, aprimorando a precisão e a eficiência desses sistemas.

Imagine uma situação em que médicos estão tentando identificar sinais precoces de uma condição médica complexa a partir de imagens de ressonância magnética. Para atingir esse objetivo, eles utilizam redes neurais profundas, como convolucionais (CNNs), que possuem milhões de parâmetros e exigem a análise de um grande volume de dados.

Nesse cenário, a computação paralela pode ser altamente benéfica. A tarefa de treinar uma rede neural profunda é intensiva em termos de cálculos e requer a otimização de parâmetros. Paralelismo de tarefas e de dados podem ser empregados para otimizar a execução do treinamento e da inferência.

Escolheria o paralelismo de dados pois o treinamento de uma rede neural profunda geralmente envolve grandes conjuntos de dados,

Capítulo 2:

Questões 1-3, 5, 10, 15-17, 19-21, 24.

2.1. Quando estávamos discutindo a adição de ponto flutuante, fizemos a suposição simplificadora de que cada uma das unidades funcionais levava o mesmo tempo.

Suponha que buscar e armazenar leva 2 nanossegundos cada e as operações restantes levam 1 nanossegundo cada.

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

b. Quanto tempo levará uma adição sem pipeline de 1.000 pares de flutuadores com essas suposições?

c. Quanto tempo levará uma adição pipeline de 1.000 pares de floats com essas suposições?

d. O tempo necessário para buscar e armazenar 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 dois nanossegundos, enquanto uma busca de um cache de nível 2 leve cinco nanossegundos e uma busca da memória principal leve cinquenta nanossegundos. O que acontece com o pipeline quando há uma falta de cache de nível 1 em uma busca de um dos operandos? O que acontece quando há uma falha de nível 2?

R: a) 9 ns, na busca dos operandos levaria 2 ns, 1 para a comparação de expoentes, 1 para a etapa de deslocamento do operando, 1 ns para adição, 1 ns para normalização dos operandos, 1 ns para o arredondamento do resultado, e por último 2 ns para o armazenamento dos operandos
b) Se para cada adição de pares levaríamos 9 ns, em uma adição com 1000 pares, teríamos 9 x 1000 = 9000 ns para que seja realizada a operação de adição sem pipeline.
c) Considerando que a etapa de busca dos operandos e de armazenamento será basicamente executada de maneira sequencial e teremos então teremos a seguinte fórmula onde chegaremos ao nosso resultado: 9(999 * 2) = 2007. neste caso teríamos o número de times por execuções (+) o produto da multiplicação da quantidade de interações que seria 1000, mas teríamos até o número 999 pois as interações são iniciadas no número zero(0) multiplicado por 2(dois) que seria o intervalo de tempo entre o término de cada interação executada em pipeline.

d) Quando ocorre o miss cache de nível 1, a busca pelo operando é realizada no próximo nível de cache, que posteriormente é o nível 2, caso a busca feita no nível 2 também não esteja disponível a busca é feita no last level 3 que como o proprio

nome diz é o ultimo nivel do cache, e caso o miss cache persista a busca é feita na memória principal, o que realizaria uma perda de desempenho na máquina, pois levaria uma quantidade maior de tempo para realizar essa busca na memória.

2.2. Explique como uma fila, implementada em hardware na CPU, pode ser usada para melhorar o desempenho de um cache write-through.

R: A partir do princípio que o write-through que é uma estratégia de gerenciamento de cache em que sempre ocorre uma escrita em um bloco de cache onde essa escrita é feita tanto na cache como na memória principal simultaneamente. Esse processo garante a consistência entre o cache e a memória principal, mas pode introduzir atrasos significativos, uma vez que as operações de escrita na memória principal tendem a ser mais lentas, e o write-through, porém quando introduzirmos as queue(filas) podemos ter um maior ganho de desempenho, pois em vez de fazermos a atualização de maneira imediata, as linhas de cache são colocadas em uma fila, o que pode reduzir a quantidade de operações redundantes, ao utilizar a fila para agendar operações de escrita na memória principal, a CPU pode continuar a executar outras instruções enquanto as operações de escrita estão sendo tratadas em segundo plano.

2.3. Lembre-se do exemplo envolvendo leituras de cache de um array bidimensional (página 22). Como uma matriz maior e um cache maior afetam o desempenho dos dois pares de loops aninhados? O que acontece se MAX = 8 e o cache puder armazenar quatro linhas? Quantas faltas ocorrem nas leituras de A no primeiro par de loops aninhados? Quantas falhas ocorrem no segundo par?

```
Unset
for (int i = 0; i < MAX; i++) {
    for (int j = 0; j < MAX; j++) {
        // Acessar A[i][j]
    }
}

for (int j = 0; j < MAX; j++) {
    for (int i = 0; i < MAX; i++) {
        y[i] += A[i][j] * x[j];
    }
```

```
}
```

Não importa o tamanho da matriz ou da cache, o desempenho do segundo loop não vai melhorar, ele só tende a piorar quanto maior for a matriz para leitura na memória principal. O número de cache miss cresce conforme o tamanho da matriz cresce, quadraticamente.
Agora em relação ao primeiro loop é diferente, quanto maior for a matriz, maior deve ser a linha de cache - onde uma linha inteira da matriz cabe na linha da cache - para um melhor desempenho, pois caso a linha da matriz seja maior que a linha de cache, provavelmente o cache miss irá aumentar.
Levando como base que o MAX = 8 e há 4 linhas na cache, temos que:
No primeiro par de loop (linha a linha) - Inicialmente a CPU irá a cache procurar A[0][0] no cache, não irá achar (cache miss) e vai buscar a linha que tem A[0][0] na memória principal, vai ler a linha inteira (junto ao A[0][0] vem também até A[0][3]). Ainda falta terminar a leitura da linha A[0][*], então ele procura na cache e não acha (cache miss), então novamente faz a leitura na memória principal. Esse processo se repete mais 14 vezes (mais 14 cache miss para finalizar a leitura). Ao todo, será 8 cache miss, pois serão necessárias 16 leituras na cache.
No segundo par de loop (coluna a coluna) - A CPU irá fazer a leitura A[0][0] na cache, não irá achar (cache miss), faz a leitura A[0][0] e junto traz a linha da matriz toda da memória principal, mas só faz leitura do A[0][0], depois vai fazer a leitura do A[1][0], não vai achar na cache (cache miss), e repete o processo a cada busca, fazendo a leitura de apenas um dado enquanto leitura traz uma linha inteira. A cada interação é feita apenas a leitura de um dado, então se a matriz é 8 x 8, haverá 64 cache miss.

2.5. A adição de cache e memória virtual a um sistema von Neumann altera sua designação como sistema SISD? E quanto à adição de pipelining? Problema múltiplo? Multithreading de hardware?

R: A adição de cache a memória virtual a um sistema von Neumann não irá sua designação como sistema SISD, pois a técnica de cache e de memória virtual acaba sendo em se uma técnica para melhorar o desempenho do sistema e não a quantidade de fluxo de dados de acordo com a classificação da taxonomia de Flynn que se pode ser processado.

Podemos dizer assim que na etapa de pipelining não se tem uma designação do sistema SISD, pois o pipeline divide as tarefas e aumenta o desempenho porém a execução do pipeline continua sendo uma única operação de apenas um fluxo de dados por vez.

No caso do Multiple Issue, várias instruções são executadas em paralelo, operando em diferentes fluxos de dados. Isso proporciona um aumento significativo no desempenho do sistema, pois ele pode realizar várias tarefas de forma simultânea.

Com o multithreading de hardware, o processador pode alternar rapidamente entre diferentes threads de execução, permitindo que várias tarefas sejam executadas concorrentemente. Essa técnica aumenta a utilização da CPU e melhora o desempenho geral do sistema.

2.10 Suponha que um programa deva executar 10^{12} instruções para resolver um determinado problema. Suponha ainda que um sistema com um único processador possa resolver o problema em 10^{6} segundos (cerca de 11,6 dias). Assim, em média, o sistema de processador único executa 10^{6} ou um milhão de instruções por segundo. Agora suponha que o programa tenha sido paralelizado para execução em um sistema de memória distribuída.

Suponha também que, se o programa paralelo usar 'p' processadores, cada processador executará $10^{12} / p$ instruções e cada processador deverá enviar $10^{9}(p - 1)$ mensagens. Finalmente, suponha que não haja sobrecarga 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 esperar por mensagens.

a. Suponha que demore 10^{-9} segundos para enviar uma mensagem. Quanto tempo levará para o programa rodar com 1.000 processadores, se cada processador for tão rápido quanto o único processador no qual o programa serial foi executado?

R:O problema/desafio é composto por 10^{12} instruções que precisam ser distribuídas entre diversos processadores para sua resolução:

$$10^{12} / 1000 = 10^{9},\ 10^{12} / 10^{3} = 10^{9}$$

cada processador executa 10^{9} instruções. Com base em suas especificações, um processador é capaz de executar 10^{6} instruções por segundo.

$$10^{9} / 10^{6} = 10^{3}$$

Será necessário 1000 segundos para executar todas as instruções.

Além disso, cada processador precisa enviar um número determinado de mensagens. Então,

$$10^{9}(1000 - 1) = 10^{9} * 999$$

cada processador enviará por volta de $10^9 * 999$ mensagem. Tomando com base no tempo de envio de uma mensagem, cada processador

$$10^{-9} * 10^9 * 999 = 999$$

levará 999 segundos para enviar todas suas mensagens.

Já que os processadores irão demorar 1000 segundos (16,6 minutos) para executar as instruções e 999 segundos para enviar as mensagens, ao todo, o programa custará 1999 segundos.

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

R:Considerando alguns cálculos feitos na questão anterior. Temos:

Cada processador executa 10^9 instruções e isso irá custar 1000 segundos;

Cada processador deverá enviar $10^9 * 999$ mensagem;

Assim, sabendo que cada mensagem leva média 10^{-3} para ser enviada. Temos que,

$$10^{-3} * 10^9 * 999 = 10^6 * 999 = 999000000$$

custará por volta de 999000000 ou $10^6 * 999$ para enviar todas as mensagens.

Com isso, para rodar todo o programa necessitará de $(10^3 + 10^6 * 999)$ segundos, ou equivalente a 31,6 anos.

2.15 a. Suponha que um sistema de memória compartilhada use coerência de cache por sondagem (snooping) e 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. Que valor será atribuído a y? Por quê?

R:O protocolo de coerência de cache por sondagem envolve de fato a comunicação entre núcleos para garantir a consistência dos dados em caches compartilhados, então de fato o valor de y será 5.

b. Suponha que o sistema de memória compartilhada da parte anterior use um protocolo baseado em diretório. Que valor será atribuído a y? Por quê?

R:No contexto de um protocolo baseado em diretório, o valor atribuído a y será 5. O núcleo 1 verificará com o diretório, que apontará para o núcleo 0 como possuidor da cópia válida e modificada de x. O núcleo 1 então obterá o valor atualizado de x do núcleo 0.

c. Você pode sugerir como os problemas encontrados nas duas primeiras partes podem ser resolvidos?

R:Caso fosse utilizado escrita direta (Write-Through) ao invés de escrita adiada (Write-Back) resolveria os problemas passados pelos métodos de coerência de cache.
Por sondagem: Não seria necessária uma comunicação extra na interconexão, pois o dado quando alterado já seria alterado na memória principal. Por diretório:

2.16 a) Suponha que o tempo de execução de um programa serial seja dado por $T_{serial} = n^2$, onde as unidades do tempo de execução estão em microssegundos. Suponha que uma paralelização desse programa tenha tempo de execução $T_{parallel} = n^2/p + log2(p)$. Escreva um programa que encontre os aumentos de velocidade e eficiência desse programa para vários valores de 'n' e 'p'. Execute seu programa com n = 10, 20, 40, … , 320 e p = 1, 2, 4, … , 128. O que acontece com os aumentos de velocidade e eficiência quando 'p' é aumentado e 'n' é mantido fixo? O que acontece quando 'p' é fixo e 'n' é aumentado?

R:Quando 'n' é fixo e 'p' é aumentado: A velocidade de execução aumenta bastante a cada adição de novos processadores, no entanto, a cada adição é provável que a eficiência do programa caia pouco ou muito conforme o tamanho do problema, se o tamanho for pequeno e haver muitos processadores, é provável que a eficiência não seja das melhores.

Quando 'p' é fixo e 'n' é aumentado: A velocidade de execução aumenta, mas chega num limite rapidamente, já que o número de processos é fixo, elas possuem uma capacidade máxima. A eficiência tente a ficar constante após um tempo, pois os recursos/processos estarão sendo 100% utilizados.

b) Suponha que $T_{parallel} = (T_{serial} / p) + T_{overhead}$. Suponha também que fixamos 'p' e aumentamos o tamanho do problema.
- Mostre que se $T_{overhead}$ cresce mais lentamente que T_{serial}, a eficiência paralela aumentará à medida que aumentamos o tamanho do problema.

R: Consideramos inicialmente T_{serial} igual a 25, p igual a 2 e $T_{overhead}$ igual a 3. A taxa de crescimento de T_{serial} é 2x e $T_{overhead}$ é 1,5x.

T_{serial} (x2)	$T_{overhead}$ (x1,5)	$T_{parallel}$ ((T_{serial} / p) + $T_{overhead}$)	Speed	Efficiency
25	3	15,5	1,61	0,805
50	4,5	29,5	1,69	0,847
100	6,75	56,75	1,76	0,881

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

Consideramos inicialmente T_{serial} igual a 25, p igual a 2 e $T_{overhead}$ igual a 3. A taxa de crescimento de T_{serial} é 1,5x e $T_{overhead}$ é 2x.

T_{serial} (1,5x)	$T_{overhead}$ (2x)	$T_{parallel}$ ((T_{serial} / p) + $T_{overhead}$)	Speed	Efficiency
25	3	15,5	1,61	0,805
37,5	6	24,75	1,51	0,757
56,25	12	40,125	1,40	0,7

Com base na análise das tabelas, é possível observar que quando o aumento $T_{overhead}$ tem um aumento maior que T_{serial}, a velocidade de execução do programa diminui e consequentemente sua eficiência também.

2.17 Um programa paralelo que obtém um speedup maior que ‘p’ - o número de processos ou threads - às vezes é dito ter um aumento de superlinear speedup. No entanto, muitos autores não contam programas que superam “limitações de recursos” como tendo superlinear speedup. 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 colocar 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 acelerações maiores que ‘p’.

R: Uma situação em que o speedup é maior que ‘p’, ou seja, um superlinear speedup é quando um problema que requer número muito grande de dados que exceder o armazenamento da memória cache de um único processador é dividido para ‘p’ processadores diferentes. Assim, o problema que requer um número grande de dados é dividido entre as memórias caches dos processadores, evitando assim

um número de cache miss exagerado do caso de um único processador, o que acaba melhorando bastante o desempenho.

2.19 Suponha que T_{serial} = n e $T_{parallel}$ = $n/p + log2(p)$, onde os tempos estão em microssegundos. Se aumentarmos 'p' por um fator de 'k', encontre uma fórmula para quanto precisaremos aumentar 'n' para manter a eficiência constante. Em quanto devemos aumentar 'n' se dobrarmos o número de processos de 8 para 16? O programa paralelo é escalável?

R: Já que manter a eficiência constante, então devemos igualar a fórmula de eficiência quando temos 8 e 16 processadores, levando em consideração o aumento que devemos fazer quando temos 16 processadores. Então, temos o seguinte:

$$\frac{n}{\frac{n/8 + log2(8)}{8}} = \frac{xn}{\frac{xn/16 + log2(16)}{16}}$$

$$\frac{n}{\frac{n/8 + 3}{8}} = \frac{xn}{\frac{xn/16 + 4}{16}}$$

$$\frac{n}{n + 24} = \frac{xn}{xn + 64}$$

$$xn^2 + 64n = xn^2 + 24xn$$

$$64n = 24xn$$

$$24xn = 64n$$

$$x = \frac{64n}{24n}$$

$$x = \frac{8}{3} = 2,67$$

Diante do resultado, quando duplicamos o número de processadores, devemos então multiplicar o tamanho do trabalho por **2,67**, assim, o programa não é nem fracamente escalável porque quando aumentamos o número de trabalho na mesma taxa que aumentamos o números de processadores.

2.20 Um programa que obtém uma aceleração linear é fortemente escalável? Explique sua resposta.

R: Não, já que para um programa ser considerado fortemente escalável, é necessário que quando o número de trabalho (n) se mantenha fixa e o número de

processos aumentar, a eficiência se mantenha inalterada ou maior. Em uma aceleração linear, a eficiência se mantém quase fixa ou menor.

2.21 Bob tem um programa que ele quer cronometrar com dois conjuntos de dados, input_data1 e input_data2. Para ter uma ideia do que esperar antes de adicionar funções de cronometragem ao código em que ele está interessado, ele executa o programa com os dois conjuntos de dados e o comando de shell Unix time:

```
$ time ./bobs_prog < input_data1

real 0m0.001s
user 0m0.001s
sys  0m0.000s

$ time ./bobs_prog < input_data2

real 1m1.234s
user 1m0.001s
sys  0m0.111s
```

A função de temporização que Bob vai usar em seu programa tem uma resolução de milissegundos. Bob deve usá-la para cronometrar seu programa com o primeiro conjunto de dados? E o segundo conjunto de dados? Por quê ou por que não?

R: Diante de que o programa utiliza uma resolução de milissegundos, Bob não deve utilizá-lo para cronometrar o primeiro conjunto de dados devido à imprecisão do tempo, o ideal seria utilizar uma resolução de nanosegundos.
Agora para o segundo conjunto de dados, a precisão/resolução é suficiente para que o tempo seja visto corretamente por Bob.

2.24 Se você ainda não fez isso no Capítulo 1, tente escrever um pseudocódigo para nossa soma global em estrutura de árvore, que soma os elementos de loc_bin_cts. Primeiro, considere como isso pode ser feito em um ambiente MIMD de memória compartilhada. Em seguida, considere como isso pode ser feito em um ambiente MIMD de memória distribuída. No ambiente de memória compartilhada, quais variáveis são compartilhadas e quais são privadas?

R:
div = 2;
core_difference = 1;
bin_count[];

while (div <= number of cores) {

if (my_rank % divisor == 0) {
parceiro = my_rank + core_difference;
recebe o valor do core parceiro;
loc_bin_cout[] += valor[];
sum += recebe o valor;

```
} else {
Parceiro = my_rank - core_difference;
eviar(loc_bin_counts[])
Enviar a soma para o código parceiro;
}
div *= 2;
core_difference *=2;
}

if(my_rank == 0){
        bin_counts[] = loc_bin_counts[];
}
```

Num sistema de memória compartilhada teriamos um compartilhamento da memória entre os cores, já na distribuida teriamos essa memória privada e seu acesso se daria apartir de uma rede.

variáveis compartilhadas:

core_difference;
div;
number of cores(que poderia ser interpretado como a variavel p);
bin_counts[];

privadas:

loc_bin_counts;
my_rank;
valor[];