UNIVERSIDADE FEDERAL DO RURAL DO SEMI-ÁRIDO
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA
PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA
DOCENTE: ÍTALO AUGUSTO SOUZA DE ASSIS

ERIKY ABREU VELOSO

PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

PAU DOS FERROS – RN
2025

QUESTÕES RESPONDIDAS

- ☑ QUESTÃO 01
- ☑ QUESTÃO 02
- ☑ QUESTÃO 03
- ☑ QUESTÃO 04
- ☑ QUESTÃO 05
- ☑ QUESTÃO 06
- ☑ QUESTÃO 07
- ☑ QUESTÃO 08
- ☑ QUESTÃO 09
- ☑ QUESTÃO 10
- ☑ QUESTÃO 11
- ☑ QUESTÃO 12
- ☑ QUESTÃO 13
- ☑ QUESTÃO 14
- ☑ QUESTÃO 15
- ☑ QUESTÃO 16
- ☑ QUESTÃO 17
- ☑ QUESTÃO 18
- ☑ QUESTÃO 19

POR QUE COMPUTAÇÃO PARALELA?

- QUESTÃO 01

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 privadas 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 elementos 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*.

RESPOSTA _____

Pseudocódigo 1. Divisão de trabalho entre os núcleos - Blocos

```C/C++
n = quantidade de valores;
p = quantidade de núcleos;
resto = n % p;
inteiro = parte inteira de n/p;
meu_desl = 1;

//ZONA PARALELA
i = índice do núcleo;
minha_soma = 0;
if(resto == 0){
       meu_pri_i = i*(inteiro);
       meu_ult_i = (i+1)*(inteiro );
}else{
       if(i < resto){
              meu_pri_i = i*(inteiro) + i;
```

```
            meu_ult_i = (i+1)*(inteiro) + i + 1;
      }else{
            meu_pri_i = i*(inteiro) + resto;
            meu_ult_i = (i+1)*(inteiro) + resto;
      }
for (meu_i = meu_pri_i; meu_i < meu_ult_i; meu_i+=meu_desl) {
      meu_x = Compute_prox_valor(. . .);
      minha_soma += meu_x;
}
```

- QUESTÃO 02

Suponha que precisamos calcular *n* valores e somá-los. Suponha também que tenhamos *p* núcleos e *p* seja muito menor que *n*. Então cada núcleo pode calcular uma soma parcial de aproximadamente *n/p* valores 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 privadas 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*. Comparado com a distribuição em blocos contíguos de iterações, sua fórmula deve reduzir a diferença do tempo de execução entre os núcleos.

RESPOSTA _____

No Pseudocódigo 1, apresentado na resolução da questão anterior, a operação de soma ocorria de maneira sequencial, ou seja, cada núcleo realiza a soma de uma determinada quantidade de valores que se configuram de maneira sequencial. Diferentemente do Pseudocódigo 2, que possui uma divisão dos elementos de maneira cíclica.

Pseudocódigo 2. Divisão de trabalho entre os núcleos - Cíclico

```C/C++
n = quantidade de valores;
p = quantidade de núcleos;

//ZONA PARALELA
i = índice do núcleo;
minha_soma = 0;
meu_pri_i = i;
meu_ult_i = n;
meu_desl = p;
for (meu_i = meu_pri_i; meu_i < meu_ult_i; meu_i+=meu_desl) {
       meu_x = Compute_prox_valor(. . .);
       minha_soma += meu_x;
}
```

Supondo uma situação hipotética, no qual se possui 16 números a serem somados e um total de 4 núcleos, pode-se observar, realizando a comparação entre os Quadros 1 e 2, a

diferença da divisão de trabalho entre os cores em ambos os cenários. Na situação descrita anteriormente, com o pseudocódigo da primeira questão, o core 0 vai executar as chamadas 0, 1, 2 e 3, o core 1 vai executar as chamadas 4, 5, 6 e 7, e assim por diante, dessa forma, o trabalho dos núcleos, na situação descrita na questão, vai ficar extremamente desbalanceado, como exposto no Quadro 1.

Quadro 1. Representação da situação aplicada ao primeiro pseudocódigo

Cores	Índice				Tempo Total
0	0	1	2	3	20
1	4	5	6	7	52
2	8	9	10	11	84
3	12	13	14	15	116

O pseudocódigo da segunda questão busca reduzir essa diferença de tempo total entre os cores, realizando uma distribuição do trabalho de forma mais eficiente, como exemplificado posteriormente no Quadro 2.

Quadro 2. Representação da situação aplicada ao segundo pseudocódigo

Cores	Índice				Tempo Total
0	0	4	8	12	56
1	1	5	9	13	64
2	2	6	10	14	72
3	3	7	11	15	80

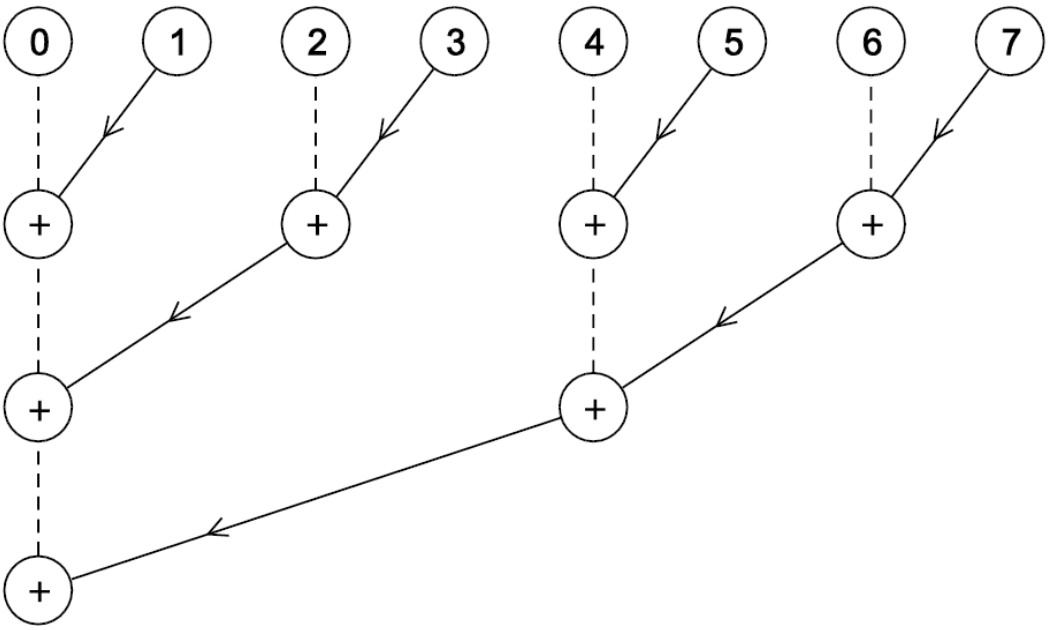
Analisando ambos os quadros, pode-se observar que a diferença do tempo de execução entre os núcleos reduziu significativamente com o segundo pseudocódigo.

- QUESTÃO 03

Escreva um pseudocódigo para uma soma global estruturada em árvore. Suponha que o número de núcleos seja uma potência de dois.

RESPOSTA _____

Figura 1. Representação da soma global estruturada em árvore



Realizando a análise do exposto na Figura 1, observou-se que para realizar a soma global estruturada em árvore, em cada iteração o núcleo necessita de um "parceiro" no qual é fornecido por a soma (em casos do núcleo receber e realizar a soma) ou subtração (em casos que o núcleo só envia os dados), do índice do referente núcleo por um deslocamento, sendo este uma potência de dois incrementado em cada estágio da soma. Essa lógica foi implementada no Pseudocódigo 3.

Pseudocódigo 3. Soma global estruturada em árvore - Sendo *p* potência de dois

```
C/C++
p = quantidade de núcleos;
meu_i = índice do núcleo;
duo_i = índice da dupla;
expoente = 2;
deslocamento = 1;
minha_soma = soma de cada núcleo;

while (deslocamento < p){
       if(meu_i % deslocamento == 0){
              if(meu_i % expoente==0){
```

```
                    duo_i = meu_i + deslocamento;
                    minha_soma += valor_duo(duo_i);
                    //Recebe o valor da dupla e soma ao seu valor

             }else{
                    duo_i = meu_i - deslocamento;
                    enviar_valor(duo_i);
                    //Envia o valor de minha_soma para a dupla
             }
      }
      expoente *= 2;
      deslocamento *= 2;
}
```

- QUESTÃO 04

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

RESPOSTA _____

Realizando uma análise sobre a soma global estruturada em árvore, observou-se que o parceiro de soma de cada etapa é fornecido por a soma (em casos do núcleo receber e realizar a soma) ou subtração (em casos que o núcleo só envia os dados), do seu índice por um deslocamento, sendo o mesmo uma potência de dois incrementado em cada estágio da soma. Realizando a análise da Tabela 1, é possível observar que ao se aplicar o operador OU EXCLUSIVO entre o fator do deslocamento e o índice do núcleo, é possível se obter o parceiro. Exemplo, considerando o núcleo 4, na segunda iteração, no qual o deslocamento é de 2, suas representações em binário são 0100 e 0010 respectivamente, ao se aplicar o XOR (`0100 ^ 0010`), o resultado obtido é igual a 0110, ou 6 em base decimal, como exposto na tabela a seguir.

Tabela 1. Parceiro de cada núcleo, baseado no deslocamento.

Núcleos	Deslocamento		
	$1_{10} = 0001_2$	$2_{10} = 0010_2$	$4_{10} = 0100_2$
$0_{10} = 0000_2$	$1_{10} = 0001_2$	$2_{10} = 0010_2$	$4_{10} = 0100_2$
$1_{10} = 0001_2$	$0_{10} = 0000_2$	-	-
$2_{10} = 0010_2$	$3_{10} = 0011_2$	$0_{10} = 0000_2$	-
$3_{10} = 0011_2$	$2_{10} = 0010_2$	-	-
$4_{10} = 0100_2$	$5_{10} = 0101_2$	$6_{10} = 0110_2$	$0_{10} = 0000_2$
$5_{10} = 0101_2$	$4_{10} = 0100_2$	-	-
$6_{10} = 0110_2$	$7_{10} = 0111_2$	$4_{10} = 0100_2$	-
$7_{10} = 0111_2$	$6_{10} = 0110_2$	-	-

O pseudocódigo 4, exposto a seguir, emprega a utilização do XOR, e, além disso, realiza o cálculo da potência de dois através do uso do operador *shift* para esquerda no fator de deslocamento.

Pseudocódigo 4. Soma global estruturada em árvore - XOR e o operador *shift* para esquerda

```
C/C++
p = quantidade de núcleos;
meu_i = índice do núcleo;
duo_i = índice da dupla;
deslocamento = 1;
minha_soma = soma de cada núcleo;

while (deslocamento < p){
	if(meu_i % deslocamento == 0){
		duo_i = meu_i ^ deslocamento;
		if(meu_i < duo_i){
			minha_soma += valor_duo(duo_i);
			//Recebe o valor da dupla e soma ao seu valor

		}else{
			enviar_valor(duo_i);
			//Envia o valor de minha_soma para a dupla
		}
	}
	deslocamento = deslocamento << 1;
}
```

- QUESTÃO 05

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

RESPOSTA _____

O Pseudocódigo 5 apresenta modificações do Pseudocódigo 3 para considerar o número de núcleos diferente de uma potência de dois.

Pseudocódigo 5. Soma global estruturada em árvore

```
C/C++
p = quantidade de núcleos;
meu_i = índice do núcleo;
duo_i = índice da dupla;
expoente = 2;
deslocamento = 1;
minha_soma = soma de cada núcleo;

while (deslocamento < p){
	if(meu_i % deslocamento == 0){
		if(meu_i % expoente==0){
			duo_i = meu_i + deslocamento;
			if(duo_i < p){
				minha_soma += valor_duo(duo_i);
				//Recebe o valor da dupla e soma ao seu valor
			}
		}else{
			duo_i = meu_i - deslocamento;
			enviar_valor(duo_i);
			//Envia o valor de minha_soma para a dupla
		}
	}
	expoente *= 2;
	deslocamento *= 2;
}
```

- QUESTÃO 06

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 núcleo 0 quando as duas somas são usadas com 2, 4, 8, · · · , 1024 núcleos.

RESPOSTA A _____

Quando apenas o núcleo 0 realiza as adições a fórmula obtida é:

$$n = p - 1$$

Onde *p* é o número de núcleo e *n* o número de recebimentos e adições.

RESPOSTA B _____

Quando a soma global é estruturada em árvore, a fórmula obtida é:

$$n = log_2(p)$$

Onde *p* é o número de núcleo e *n* o número de recebimentos e adições.

Figura 1. Representação da soma global estruturada em árvore

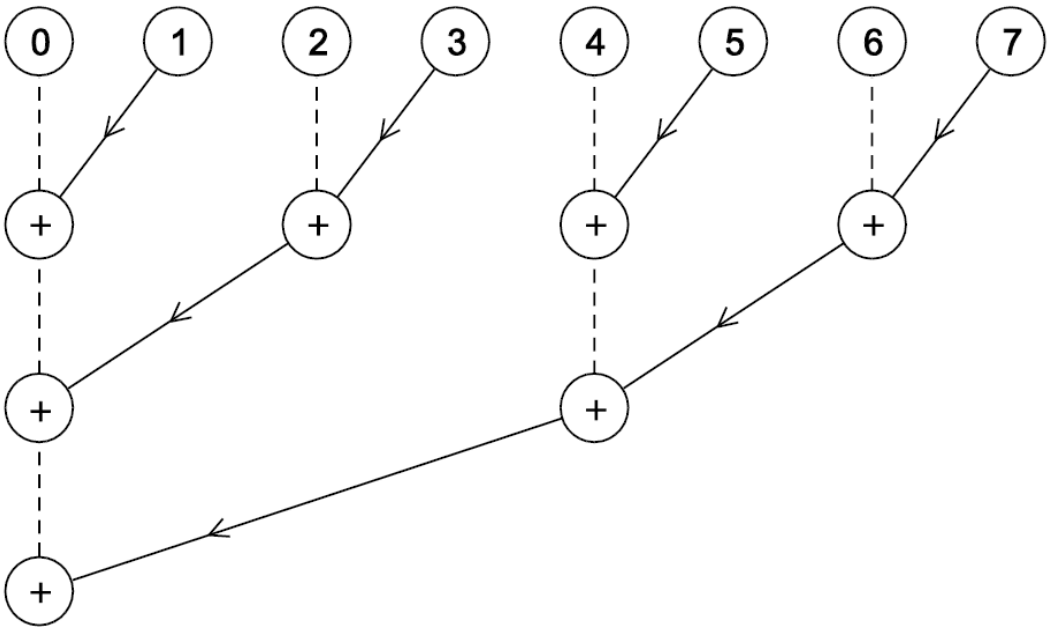


Tabela 2. Número de recebimentos e adições para ambos enunciados

p	A	B
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

- QUESTÃO 07

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?

RESPOSTA _____

Otimização do Treinamento e Inferência de Modelos de Deep Learning para Classificação Astronômica em Larga Escala, o problema de pesquisa foca na otimização do treinamento e inferência de modelos de Deep Learning, especificamente Redes Neurais Perceptron Multicamadas, para a classificação de objetos astronômicos em larga escala. A astronomia moderna, impulsionada pela evolução de equipamentos de observação como telescópios e espectrógrafos, gera uma quantidade massiva de dados. Classificar esses dados em categorias como estrelas, quasares e galáxias é crucial, mas a análise e classificação sequencial de milhões ou bilhões de observações espectrais se torna computacionalmente inviável e demorada. O desafio reside em processar eficientemente esse vasto volume de informações para treinar modelos complexos e, subsequentemente, utilizá-los para classificar novas descobertas de forma rápida e precisa.

O vasto conjunto de dados astronômicos seria particionado em conjuntos de dados reduzidos. Cada pequeno conjunto de dados seria então distribuído para diferentes unidades de processamento. Cada GPU executaria independentemente o cálculo da previsão do modelo e o cálculo de gradientes para as amostras de dados que recebeu.

Após o cálculo dos gradientes por cada unidade, eles seriam agregados para obter um gradiente global. Este gradiente agregado seria então usado por um otimizador para atualizar os pesos do modelo MLP, que seriam sincronizados entre todas as unidades de processamento antes da próxima iteração de treinamento.

Neste contexto, o paralelismo de dados seria a abordagem predominante e mais eficaz. A principal razão para isso é que o problema envolve a aplicação da mesma sequência de operações computacionais a grandes volumes de dados independentes. Ao dividir os dados em conjunto de dados menores e processar cada um em paralelo, podemos escalar significativamente o desempenho do sistema.

HARDWARE E SOFTWARE PARALELOS

- QUESTÃO 08

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?
(b) Quanto tempo levará uma adição sem pipeline de 1.000 pares de *floats* com essas suposições?
(c) Quanto tempo levará uma adição em pipeline de 1.000 pares de *floats* com essas suposições?
(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 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 falha de cache de nível 1 na busca de um dos operandos? O que acontece quando há uma falha de nível 2?

RESPOSTA A _____

Considerando as 7 operações necessárias para realizar uma adição de ponto flutuante: buscar (*Fetch operands*), comparar expoente (*Compare exponents*), deslocar (*Shift one operand*), adicionar (*Add*), normalizar (*Normalize result*), Arredondar (*Round result*) e armazenar (*Store result*). E sabendo que buscar e armazenar levem 2 nanossegundos cada e que as operações restantes levem 1 nanossegundo cada, para realizar uma única adição se é necessário 9 nanosegundos.

RESPOSTA B _____

Compreendendo, baseado no item anterior, que cada adição de pares de ponto flutuante necessitam de 9 nanosegundos, em uma adição com 1000 pares, realizada de forma sequencial, sem pipeline, o tempo final necessário pode ser obtido mediante: $9ns * 1000 = 9000\,ns$.

RESPOSTA C _____

Em uma arquitetura com pipeline, as diferentes etapas de várias operações podem se sobrepor no tempo. Considerando o exposto no Quadro 3, nota-se que o "gargalo" do pipeline é definido pela etapa mais longa, em outras palavras, o intervalo de tempo entre o término de cada iteração executada em pipeline é definido pela etapa mais longa, neste caso, buscar e armazenar levem 2 nanossegundos. Portanto, o tempo total necessário é dado pela seguinte fórmula:

$$Tt = Tp + (N - 1) \times Tc$$

Onde *Tt* é o tempo total, *Tp* é o tempo da primeira operação, *N* é o número de operações e *Tc* é o tempo da etapa mais longa. Portanto, com 1000 pares, considerando o "gargalo" como 2 ns e a tempo de uma iteração como 9 ns, obtemos o valor de 2007 ns.

Quadro 3. Representação de uma adição em pipeline

Tempo(ns)	Busca	Compara	Desloca	Adiciona	Normaliza	Arredonda	Armazena
1	1						
2	1						
3	2	1					
4	2		1				
5	3	2		1			
6	3		2		1		
7	4	3		2		1	
8	4		3		2		1
9	5	4		3		2	1
10	5		4		3		2
11	6	5		4		3	2
12	6		5		4		3
13	7	6		5		4	3
14	7		6		5		4
15		7		6		5	4
…	…	…	…	…	…	…	…
2005						1000	999
2006							1000
2007							1000

RESPOSTA D _____

Quando ocorre uma falha de cache de nível 1 durante a busca de um operando, o pipeline precisa interromper seu avanço normal. Nesse caso, o processador é obrigado a buscar o operando no cache de nível 2, que possui um tempo de acesso maior. Por exemplo, se a busca no cache L1 levaria normalmente dois nanossegundos, ao falhar e precisar acessar o cache L2, esse tempo aumenta para sete nanossegundos.

Se, além disso, houver uma falha também no cache de nível 2, o pipeline precisa buscar o dado diretamente na memória principal, cujo tempo de acesso é muito mais elevado, chegando a cinquenta nanossegundos. Logo, o tempo total de busca custará 57 ns.

- QUESTÃO 09

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

RESPOSTA _____

Uma fila implementada em hardware dentro da CPU pode ser utilizada para melhorar significativamente o desempenho de um cache write-through, uma vez que, nesse tipo de cache, toda vez que ocorre uma escrita, a atualização precisa ser imediatamente propagada à memória principal. Embora isso garanta maior consistência, também pode introduzir atrasos, pois cada operação de escrita depende diretamente do tempo de acesso à memória.

Ao adicionar uma fila de escrita implementada em hardware, a CPU consegue armazenar temporariamente os dados a serem escritos na memória. Assim, logo após a CPU realizar a operação de escrita no cache, ela pode colocar o valor na fila e seguir para a próxima instrução sem precisar esperar o término do acesso à memória principal. A fila, por sua vez, gerencia essas escritas em segundo plano, enviando-as à memória principal de forma assíncrona.

- QUESTÃO 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?
(b) Como o aumento do tamanho da cache afetaria o desempenho dos dois pares de loops aninhados?
(c) Quantas falhas ocorrem nas leituras de *A* no primeiro par de loops aninhados?
(d) Quantas falhas ocorrem no segundo par?

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

RESPOSTA A _____

Quando aumentamos o tamanho da matriz, o desempenho dos dois pares de loops aninhados tende a piorar, principalmente devido ao maior número de acessos à memória e do aumento das falhas de cache. No primeiro par de loops, o acesso à matriz A ocorre percorrendo a linha inteira antes de mudar para a próxima, esse acesso "linear" tende a aproveitar melhor a localidade espacial, reduzindo falhas de cache. Já no segundo par de loops, os índices percorrem trocando a ordem, primeiro percorrem as linhas (i) para cada coluna (j), isso faz com que o acesso ocorra de forma "vertical" em memória, prejudicando a localidade e aumentando o número de falhas de cache à medida que a matriz cresce.

RESPOSTA B _____

Considerando que, com o aumento da memória cache, o valor suportado para armazenamento seja menor ou igual a MAX, o desempenho do primeiro loop melhora consideravelmente, pois ele passa a aproveitar melhor a localidade de linha, reduzindo significativamente o número de falhas de cache. No entanto, para o segundo loop, esse aumento de memória tem efeito insignificante, mantendo praticamente o mesmo número de cache misses, já que o padrão de acesso por colunas continua desfavorável. Por outro lado, se

a memória cache for suficientemente grande a ponto de suportar valores superiores a MAX, ou até mesmo armazenar toda a matriz, ambos os laços passam a obter ganhos expressivos de desempenho, com uma redução muito significativa nas falhas de cache.

RESPOSTA C

No primeiro par de loops, cada linha de cache armazena quatro elementos consecutivos de A, como exposto no enunciado. Como no primeiro loop o acesso à matriz A ocorre percorrendo a linha inteira antes de mudar para a próxima, a cada quatro elementos acessados em A[i][j], ocorre uma falha de cache, logo, em uma única linha ocorrem $\frac{MAX}{4}$ falhas. Como existem MAX linhas, a fórmula obtida para calcular a quantidade de falhas de cache será:

$$\frac{MAX}{4} * MAX = \frac{MAX^2}{4}$$

RESPOSTA D

No segundo par de loops, percorremos primeiro sobre as linhas para cada elemento das colunas, ou seja, acessos verticais. Para cada coluna j, ao acessar A[i][j], cada elemento A[0][j], A[1][j], ..., A[MAX−1][j] está em uma linha diferente de A na memória. Cada elemento individual A[i][j] pertence a uma linha separada em cache, e não há reutilização. Assim, para cada elemento, ocorre uma falha de cache, resultando em MAX falhas por coluna. Como existem MAX colunas, o total de falhas será:

$$MAX * MAX = MAX^2$$

- QUESTÃO 11

Responda às questões abaixo. Explique o porquê de cada resposta.

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

(b) E quanto à adição de pipeline?

(c) Multiple issue?

(d) Hardware multithreading?

RESPOSTA A _____

Não, a adição de cache e memória virtual não altera a classificação do sistema como SISD. Mesmo com a adição de cache, para acelerar acessos à memória, ou memória virtual, para gerenciar memória de forma mais flexível, o sistema ainda opera com apenas uma instrução e um dado por vez em sua unidade de processamento. Esses recursos são melhorias de desempenho, mas não mudam a natureza sequencial e única da execução.

RESPOSTA B _____

Não, a adição de pipeline também não altera a classificação para além de SISD. Apesar de parecer paralelismo, o pipeline não cria múltiplos fluxos independentes de instrução ou dados. O processador ainda processa instruções individuais, uma após a outra, só que com maior eficiência.

RESPOSTA C _____

Não, a técnica de Multiple Issue se refere à capacidade de um processador de iniciar a execução de múltiplas instruções em um único ciclo de clock. Embora um processador superescalar possa iniciar várias instruções no mesmo ciclo de clock, essas instruções ainda fazem parte de um único fluxo de instruções, ou seja, da mesma thread ou programa, e operam sobre o fluxo de dados dessa mesma thread.

RESPOSTA D _____

Hardware Multithreading é uma técnica que permite que um único núcleo de processador execute instruções de múltiplas threads simultaneamente, logo, altera a designação do sistema SISD. Com o Hardware Multithreading, um único núcleo do processador consegue executar instruções de múltiplas threads simultaneamente, cada thread tem seu próprio fluxo de instruções e opera em seu próprio fluxo de dados.

- QUESTÃO 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 $10^{12}/p$ instruções. 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?

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

RESPOSTA A _____

Considerando uma quantidade de 1000 processadores ($p = 1000$), logo, cada processador realizará 10^{9} instruções, como calculado a seguir.

$$\frac{10^{12}}{10^{3}} = 10^{9}$$

Além disso, cada processador deverá enviar, $999 * 10^{9}$ mensagens.

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

O tempo necessário para realizar todas as instruções, pode ser obtido da seguinte maneira:

1 segundo	—	10^{6} instruções
x segundos	—	10^{9} instruções

$$x * 10^{6} = 10^{9} * 1$$

$$x = \frac{10^9}{10^6} = 10^3 \, segundos$$

O tempo necessário para realizar o envio das mensagens pode ser obtido através da multiplicação da quantidade de mensagens pelo tempo necessário para realizar o envio de uma mensagem:

$$999 * 10^9 * 10^{-9} = 999 \, segundos$$

Logo o tempo total necessário, supondo que não haja overhead adicional na execução do programa paralelo, será:

$$999 + 10^3 = 1999 \, segundos$$

Aproximadamente 33,32 minutos.

RESPOSTA B _____

Considerando uma quantidade de 1000 processadores ($p = 1000$), logo, cada processador realizará 10^9 instruções, como calculado a seguir.

$$\frac{10^{12}}{10^3} = 10^9$$

Além disso, cada processador deverá enviar, $999 * 10^9$ mensagens.

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

O tempo necessário para realizar todas as instruções, pode ser obtido da seguinte maneira:

1 segundo	—	10^6 instruções
x segundos	—	10^9 instruções

$$x * 10^6 = 10^9 * 1$$
$$x = \frac{10^9}{10^6} = 10^3 \, segundos$$

O tempo necessário para realizar o envio das mensagens pode ser obtido através da multiplicação da quantidade de mensagens pelo tempo necessário para realizar o envio de uma mensagem:

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

Logo o tempo total necessário, supondo que não haja overhead adicional na execução do programa paralelo, será:

$$\begin{aligned} 999 * 10^{6} + 10^{3} &= 10^{3} * (999 * 10^{3} + 1) \\ &= 1000 * (999 * 1000 + 1) \\ &= 1000 * (999000 + 1) \\ &= 1000 * 999001 \\ &= 999001000\ segundos \end{aligned}$$

Aproximadamente 31,68 anos.

- QUESTÃO 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?
(b) Suponha que o sistema de memória compartilhada da parte anterior utilize um protocolo baseado em diretório. Qual valor será atribuído a y? Por que?
(c) Como os problemas encontrados nas duas primeiras partes podem ser resolvidos?

RESPOSTA A _____

O valor atribuído a y será o valor antigo de x. Em um sistema com coerência de cache baseada em snooping e caches do tipo write-back, todos os núcleos compartilham um mesmo barramento. Isso significa que qualquer alteração transmitida nesse barramento pode ser "observada" por todos os núcleos conectados. Quando um núcleo atualiza a variável x em seu próprio cache, ele também envia essa modificação pelo barramento. Caso outro núcleo esteja "espionando" esse barramento e possua x em seu cache, ele detecta a alteração e invalida sua cópia local. No entanto, se esse outro núcleo não tiver a variável x armazenada em seu cache no momento da modificação, a mensagem de atualização será simplesmente ignorada. Portanto, considerando o exemplo citado, o núcleo 1 não possui x em seu cache quando o núcleo 0 realiza a atualização. Assim, ao acessar x, o núcleo 1 buscará diretamente na memória principal, obtendo o valor anterior à modificação feita por núcleo 0.

RESPOSTA B _____

Utilizando um protocolo baseado em diretório, o valor atribuído a y também será o valor antigo de x. Nesse tipo de coerência de cache, uma estrutura chamada diretório mantém o controle do estado de cada linha de cache, registrando quais núcleos possuem cópias de determinadas variáveis. Quando uma variável é modificada, o diretório é consultado para identificar quais caches possuem essa variável, e esses núcleos são então notificados para invalidarem suas cópias. No entanto, no exemplo citado, o núcleo 1 não possui a variável x em seu cache no momento em que o núcleo 0 realiza a modificação. Como resultado, o diretório não envia nenhuma notificação ao núcleo 1, e este acaba acessando o valor antigo de x diretamente da memória principal ao atribuí-lo à variável y.

RESPOSTA C _____

Utilizando Write-Through em vez de Write-Back, a consistência dos dados na memória principal seria garantida imediatamente após cada escrita, pois toda modificação feita no cache é automaticamente refletida na memória principal. Isso elimina o risco de leitura de valores desatualizados por outros núcleos, uma vez que a memória principal sempre conterá o dado mais recente. No entanto, essa abordagem pode aumentar o tráfego no

barramento e impactar o desempenho, especialmente em sistemas com muitas operações de escrita.

- QUESTÃO 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 eficiências deste 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 speedups e eficiências à medida que p aumenta e n é mantido fixo?
- O que acontece quando p é fixo e n é aumentado?

(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 paralela aumentará à medida que aumentarmos o tamanho do problema.
- 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.

RESPOSTA A _____

Baseado nos cálculos expostos no material de apoio da disciplina, os speedups foram obtidos da seguinte maneira:

$$Speedup = \frac{T_{Serial}}{T_{Parallel}}$$

Para o cálculo da eficiência utilizou-se:

$$Efficiency = \frac{T_{Serial}}{p * T_{Parallel}}$$

Ambas as equações foram implementadas no Código 1 a seguir.

Código 1. Programa para cálculo de acelerações e eficiências

```
C/C++
#include <stdio.h>
#include <math.h>

int main() {
    int n[6] = {10, 20, 40, 80, 160, 320};
    int p[8] = {1, 2, 4, 8, 16, 32, 64, 128};
    double t_seqencial, t_paralelo, speedup, efficiency;
```

```
    for(int x = 0 ; x < 6; x++){
        printf("------------- TAMANHO DO PROBLEMA: %d -------------\n",
n[x]);
        t_seqencial = pow(n[x], 2);
        printf("-- TEMPO SEQUENCIAL: %.2f \n", t_seqencial);
        printf("-- PARALELO: \n");
        for(int y = 0; y < 8; y++){
            t_paralelo = (t_seqencial / p[y]) + log2(p[y]);
            printf("- PARA %d THREADS: \n", p[y]);
            printf("TEMPO: %.2f \n", t_paralelo);
            speedup = t_seqencial / t_paralelo;
            printf("SPEEDUP: %.2f \n", speedup);
            efficiency = t_seqencial / (p[y]*t_paralelo);
            printf("EFICIENCIA: %.2f \n", efficiency);
        }

    }
    return 0;
}
```

Baseado nas saídas apresentadas na Tabela 3, ao aumentar o número de threads e manter fixo o tamanho do problema, vemos que os speedups inicialmente crescem, mas depois tendem a saturar ou até reduzir. Esse fenômeno ocorre mais rápido e evidentemente em problemas pequenos, pois o ganho adicional com mais threads é limitado, logo, quando considerado o overhead, o ganho real é reduzido.

Tabela 3. Speedups baseado no tamanho do problema e no número de threads

Número de Threads	Tamanho do Problema					
	10	20	40	80	160	320
1	1,00	1,00	1,00	1,00	1,00	1,00
2	1,96	1,99	2,00	2,00	2,00	2,00
4	3,70	3,92	3,98	4,00	4,00	4,00
8	6,45	7,55	7,88	7,97	7,99	8,00
16	9,76	13,79	15,38	15,84	15,96	15,99
32	12,31	22,86	29,09	31,22	31,80	31,95
64	13,22	32,65	51,61	60,38	63,05	63,76
128	12,85	39,51	82,05	112,28	123,67	126,89

Entretanto, quando mantemos o número de threads fixo e aumentamos o tamanho do problema, os speedups aumentam de forma consistente. Isso ocorre porque, com problemas maiores, o trabalho computacional se torna mais dominante em relação ao tempo de overhead. Assim, as threads conseguem ser melhor aproveitados, alcançando speedups muito próximos do ideal.

Quando aumentamos o número de threads e mantemos fixo o tamanho do problema, a eficiência tende a cair. Como exposto na Tabela 4, esse fenômeno fica mais evidente em problemas menores, pois existindo pouco trabalho para dividir entre as threads, a sobrecarga de comunicação, sincronização e gerenciamento se torna proporcionalmente maior, desperdiçando capacidade de processamento.

Entretanto, quando mantemos o número de threads fixo e aumentamos o tamanho do problema, a eficiência aumenta e tende a se aproximar de 1. Isso ocorre porque, com mais trabalho disponível, as threads conseguem operar por mais tempo em atividades úteis, diluindo o impacto do overhead.

Tabela 4. Eficiência baseada no tamanho do problema e no número de threads

Número de Threads	Tamanho do Problema					
	10	20	40	80	160	320
1	1,00	1,00	1,00	1,00	1,00	1,00
2	0,98	1,00	1,00	1,00	1,00	1,00
4	0,93	0,98	1,00	1,00	1,00	1,00
8	0,81	0,94	0,99	1,00	1,00	1,00
16	0,61	0,86	0,96	0,99	1,00	1,00
32	0,38	0,71	0,91	0,98	0,99	1,00
64	0,21	0,51	0,81	0,94	0,99	1,00
128	0,10	0,31	0,64	0,88	0,97	0,99

RESPOSTA B

Realizando modificações no Código 1 para atender ao solicitado no item B, fixamos o número de threads em 32, e no primeiro tópico, no qual solicita que $T_{overhead}$ cresça mais lentamente que $T_{sequencial}$, adota-se `T_overhead = pow(n[x], 1.5)` e `T_sequencial = pow(n[x], 2)`, como exposto no Código 2.

Código 2. Programa para cálculo de acelerações e eficiências

```
C/C++
#include <stdio.h>
```

```c
#include <math.h>

int main() {
    int n[6] = {10, 20, 40, 80, 160, 320};
    int p = 32;
    double t_seqencial, t_paralelo, t_overhead, speedup, efficiency;


    for(int x = 0 ; x < 6; x++){
        printf("------------- TAMANHO DO PROBLEMA: %d -------------\n",
n[x]);
        t_seqencial = pow(n[x], 2);
        t_overhead = pow(n[x], 1.5);
        printf("-- TEMPO SEQUENCIAL: %.2f \n", t_seqencial);
        printf("-- TEMPO OVERHEAD: %.2f \n", t_overhead);
        printf("-- PARALELO: \n");
        t_paralelo = (t_seqencial / p) + t_overhead;
        printf("- PARA %d THREADS: \n", p);
        printf("TEMPO: %.2f \n", t_paralelo);
        speedup = t_seqencial / t_paralelo;
        printf("SPEEDUP: %.2f \n", speedup);
        efficiency = t_seqencial / (p*t_paralelo);
        printf("EFICIENCIA: %.2f \n", efficiency);

    }

    return 0;
}
```

Pode-se observar na Tabela 5 que a eficiência paralela sofre um aumento à medida que aumentamos o tamanho do problema, quando $T_{overhead}$ crescer mais lentamente que $T_{sequencial}$, esse fenômeno reflete a realidade de programas que não exigem muita comunicação entre as threads, no qual a medida que aumentarmos o tamanho do problema, o tempo necessário de comunicação passa a ser menos significativo no tempo total.

Tabela 5. Eficiência baseada no tamanho do problema

Número de Threads	Tamanho do Problema					
	10	20	40	80	160	320
32	0,09	0,12	0,17	0,22	0,28	0,36

Almejando atender o solicitado no segundo tópico do Item B, foi realizada uma pequena modificação no Código 2, como exposta no Código 3.

Código 3. Programa para cálculo de acelerações e eficiências

```C/C++
#include <stdio.h>
#include <math.h>

int main() {
    int n[6] = {10, 20, 40, 80, 160, 320};
    int p = 2;
    double t_seqencial, t_paralelo, t_overhead, speedup, efficiency;


    for(int x = 0 ; x < 6; x++){
        printf("------------- TAMANHO DO PROBLEMA: %d -------------\n",
n[x]);
        t_seqencial = pow(n[x], 2);
        t_overhead = pow(n[x], 2.5);
        printf("-- TEMPO SEQUENCIAL: %.2f \n", t_seqencial);
        printf("-- TEMPO OVERHEAD: %.2f \n", t_overhead);
        printf("-- PARALELO: \n");
        t_paralelo = (t_seqencial / p) + t_overhead;
        printf("- PARA %d THREADS: \n", p);
        printf("TEMPO: %.2f \n", t_paralelo);
        speedup = t_seqencial / t_paralelo;
        printf("SPEEDUP: %.2f \n", speedup);
        efficiency = t_seqencial / (p*t_paralelo);
        printf("EFICIENCIA: %.2f \n", efficiency);

    }

    return 0;
}
```

Pode-se observar na Tabela 6 que a eficiência paralela sofre uma redução à medida que aumentamos o tamanho do problema, quando $T_{overhead}$ crescer mais rápido que $T_{sequencial}$, esse fenômeno reflete a realidade de programas que exigem uma dependência alta entre as tarefas realizadas nas threads, no qual a medida que aumentamos o tamanho do problema, o número necessário de comunicação aumenta.

Tabela 6. Eficiência baseada no tamanho do problema

Número de Threads	Tamanho do Problema					
	10	20	40	80	160	320
2	0,14	0,10	0,07	0,05	0,04	0,03

- QUESTÃO 15

Às vezes, diz-se que um programa paralelo que obtém um *speedup* maior que *p* (o número de processos ou *threads*) tem speedup superlinear. No entanto, muitos autores não consideram os programas que superam as "limitações de recursos" como tendo aceleração 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*.

RESPOSTA _____

Um exemplo relevante de aceleração superlinear ocorre quando o volume de dados processado por um único processador é suficientemente grande para exceder a capacidade de sua memória cache. Nesse cenário, a execução sequencial sofre penalizações de desempenho significativas devido à alta taxa de falhas de cache (cache misses), o que resulta em acessos frequentes à memória principal, mais lenta.

Ao paralelizar a aplicação entre *p* processadores, o volume total de dados é dividido aproximadamente por *p*, reduzindo a carga de dados atribuída a cada processador. Essa divisão pode permitir que os dados de cada processador se ajustem integralmente à sua memória cache, minimizando as falhas de cache e maximizando o aproveitamento do acesso rápido à memória.

Como consequência, o desempenho individual de cada processador pode ser significativamente superior ao do processador único, possibilitando a obtenção de um *speedup* maior que *p*. Esse ganho adicional decorre não apenas da divisão do trabalho, mas também da eliminação de gargalos relacionados ao acesso à memória.

- QUESTÃO 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 aumentar 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?

RESPOSTA A _____

Sabendo que para realizar o cálculo da eficiência deve-se utilizar a seguinte fórmula:

$$Efficiency = \frac{T_{Serial}}{p * T_{Parallel}}$$

Logo, substituindo T_{Serial} e $T_{Parallel}$ pelas equações fornecidas na questão, obtemos:

$$\begin{aligned} Efficiency &= \frac{n}{p*(\frac{n}{p}+log_2(p))} \\ &= \frac{n}{\frac{p*n}{p}+p*log_2(p)} \\ &= \frac{n}{n+p*log_2(p)} \end{aligned}$$

Visando manter a eficiência constante caso aumentarmos p por um fator de k:

$$\frac{n}{n+p*log_2(p)} = \frac{n_k}{n_k+k*p*log_2(p*k)}$$

$$n_k = \frac{n*(n_k+k*p*log_2(p*k))}{n+p*log_2(p)}$$

Sabendo que $n_k = n * x$, onde x é o fator que se deseja descobrir, obtemos:

$$n * x = \frac{n*(n*x+k*p*log_2(p*k))}{n+p*log_2(p)}$$

$$\frac{n*x}{n} = \frac{n*x+k*p*log_2(p*k)}{n+p*log_2(p)}$$

$$x = \frac{n*x+k*p*log_2(p*k)}{n+p*log_2(p)}$$

$$x = \frac{n*x}{n+p*log_2(p)} + \frac{k*p*log_2(p*k)}{n+p*log_2(p)}$$

$$x - \frac{n*x}{n+p*log_2(p)} = \frac{k*p*log_2(p*k)}{n+p*log_2(p)}$$

$$x * \left(\frac{1}{1} - \frac{n}{n+p*log_2(p)}\right) = \frac{k*p*log_2(p*k)}{n+p*log_2(p)}$$

$$x * \left(\frac{n+p*log_2(p) - n}{n+p*log_2(p)}\right) = \frac{k*p*log_2(p*k)}{n+p*log_2(p)}$$

$$x = \frac{(k*p*log_2(p*k)) * (n+p*log_2(p))}{(n+p*log_2(p))*(p*log_2(p))}$$

$$x = \frac{k*p*log_2(p*k)}{p*log_2(p)}$$

$$x = \frac{k*log_2(p*k)}{log_2(p)}$$

Substituindo pelos valores fornecidos no item A, obtemos:

$$x = \frac{2*log_2(16)}{log_2(8)}$$

$$x = \frac{2*4}{3}$$

$$x = \frac{8}{3}$$

Logo, para manter sua eficiência ao dobrarmos o número de processos, basta aumentar *n* em $\frac{8}{3}$ vezes.

RESPOSTA B _____

Não é escalável, pois se aumentarmos a quantidade de processos, é necessário aumentar a quantidade de problemas, para o programa manter a eficiência.

- QUESTÃO 17

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

RESPOSTA _____

Um programa que possui um *speedup* linear, pode ter seu tempo calculado através da seguinte equação:

$$T_{parallel} = \frac{T_{serial}}{p}$$

Logo, o valor da eficiência pode ser obtido através da seguinte fórmula:

$$Efficiency = \frac{T_{Serial}}{p * T_{Parallel}} = \frac{T_{Serial}}{p * \frac{T_{Serial}}{p}} = \frac{T_{Serial}}{T_{Serial}} = 1$$

Como a eficiência permanecerá constante quando aumentarmos o número de processos / *threads,* sem aumentar o tamanho do problema, o programa se caracteriza como fortemente escalável, com a eficiência igual a 1.

- QUESTÃO 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 temporização 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?

RESPOSTA A _____

Bob não deverá utilizar essa função de temporização, pois o tempo de execução com o `input_data1` foi de 1 milissegundo, real 0m0.001s, o que está no limite da resolução da função de temporização que está sendo utilizada. Como a resolução é de milissegundos, esse valor pode ser impreciso ou até sujeito a flutuações causadas por ruído do sistema operacional.

RESPOSTA B _____

Bob pode utilizar a função de temporização com o segundo conjunto de dados, pois os resultados são estáveis, o tempo de execução foi de mais de um minuto (real 1m1.234s), o que está muito acima da resolução mínima da função de milissegundos.

- QUESTÃO 19

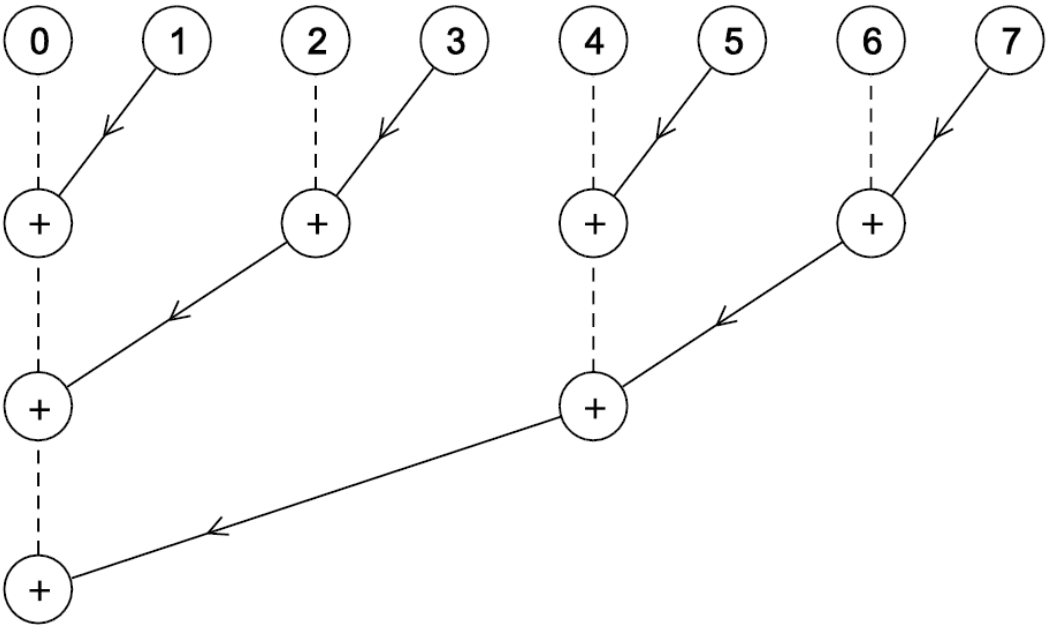
Escreva um pseudocódigo para a soma global estruturada em árvore para somar vetores.

(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.

RESPOSTA A _____

Figura 2. Representação da soma global estruturada em árvore



Realizando a análise do exposto na Figura 2, observou-se que para realizar a soma global estruturada em árvore, em cada iteração o núcleo necessita de um "parceiro" no qual é fornecido por a soma (em casos do núcleo receber e realizar a soma) ou subtração (em casos que o núcleo só envia os dados), do índice do referente núcleo por um deslocamento, sendo este uma potência de dois incrementado em cada estágio da soma. Essa lógica foi implementada no Pseudocódigo 3.

Pseudocódigo 6. Soma global estruturada em árvore

```
C/C++
p = quantidade de núcleos; //Compartilhada
tamanho_vetor = definir o tamanho do vetor; //Compartilhada
vetores[p][tamanho_vetor] = vetor dos vetores; //Compartilhada
resultado[tamanho_vetor] = vetor final com a soma global; //Compartilhada
```

```
meu_i = índice do núcleo; //Privada
duo_i = índice da dupla; //Privada
expoente = 2; //Privada
deslocamento = 1; //Privada
minha_soma[] = soma de cada núcleo; //Privada

while (deslocamento < p){
       if(meu_i % deslocamento == 0){
              if(meu_i % expoente==0){
                     duo_i = meu_i + deslocamento;
                     if(duo_i < p){
                            for (j = 0; j < tamanho_vetor; j++){
                                   minha_soma[j] += valor_duo[duo_i][j];
                            }
                            //Recebe o vetor da dupla e soma ao seu vetor
                     }
              }else{
                     duo_i = meu_i - deslocamento;
                     vetores[meu_i] = minha_soma;
                     //Envia o vetor "minha_soma" para a dupla
              }
       }
       expoente *= 2;
       deslocamento *= 2;
}

if (meu_i == 0) {
    resultado = minha_soma
}
```

RESPOSTA B _____

No ambiente de memória distribuída, cada processo possui sua própria memória local, e a comunicação entre eles ocorre explicitamente via envio e recebimento de mensagens. Logo, todas as variáveis são locais, como exposto no Pseudocódigo 7.

Pseudocódigo 7. Soma global estruturada em árvore

```
C/C++
p = quantidade de processos;
tamanho_vetor = definir o tamanho do vetor;
```

```
meu_i = identificador do processo;
duo_i = identificador da dupla;
expoente = 2;
deslocamento = 1;
meu_vetor[tamanho_vetor] = vetor local;
buffer[] = vetor temporário do duo;

while (deslocamento < p){
       if(meu_i % deslocamento == 0){
              if(meu_i % expoente==0){
                     duo_i = meu_i + deslocamento;
                     if(duo_i < p){
                            Receber(buffer[]);
                            //Receber o vetor da dupla e armazenar em buffer
                            for (j = 0; j < tamanho_vetor; j++){
                                   meu_vetor[j] += buffer[j];
                            }
                     }
              }else{
                     duo_i = meu_i - deslocamento;
                     Enviar(meu_vetor[]);
                     //Enviar o vetor para a dupla
              }
       }
       expoente *= 2;
       deslocamento *= 2;
}

if (meu_i == 0) {
    imprimir(meu_vetor);
}
```