Programação com MPI

Ementa

- Estudo de caso: método do trapézio
- Estudo de caso: multiplicação de matriz
- Estudo de caso: número primos

Bibliografia

https://www.casadocodigo.com.br/products/livro-programacao-paralela

Google Colab

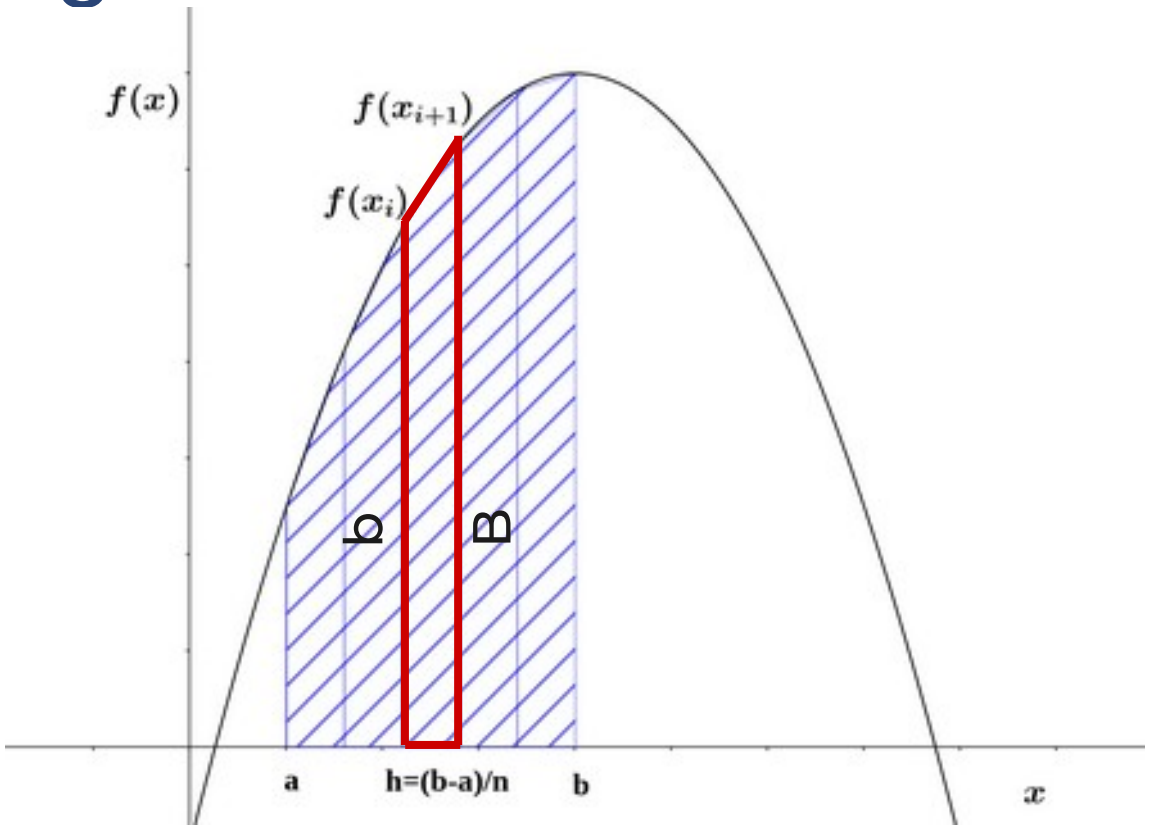
- Os exemplos, notebooks e slides do minicurso estão disponíveis no seguinte endereço:
- https://github.com/Programacao-Paralela-e-Distribuida/MPI/

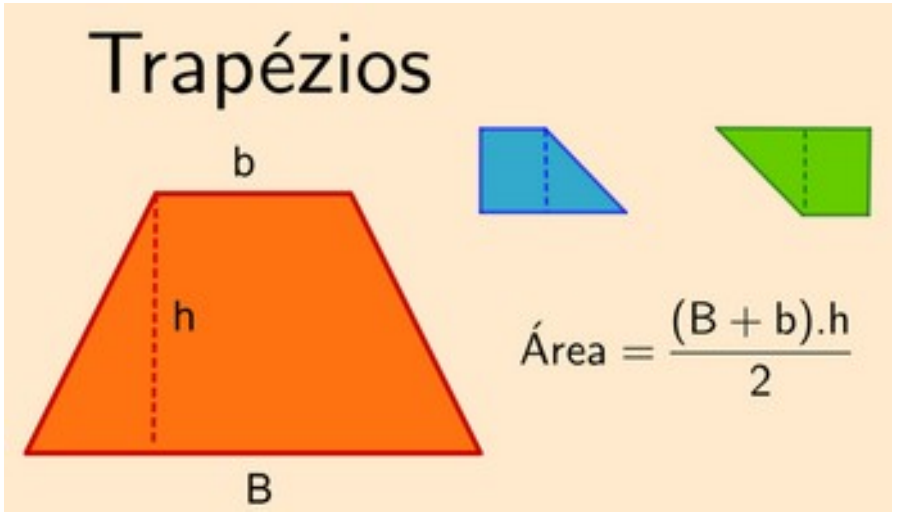
Método do Trapézio

Integral Definida - Método do Trapézio

- O valor de uma integral definida pode ser aproximado por vários métodos numéricos.
- Um dos métodos mais utilizados é o método do trapézio, onde o valor da integral de uma função, que é definido como a área sob a curva da função até o eixo x, é aproximado pela soma da área de diversos trapézios.
- Vejamos uma ilustração no slide a seguir.

Integral Definida - Método do Trapézio





Integral Definida - Método do Trapézio

- Vamos lembrar que o método do trapézio estima o valor de f(x) dividindo o intervalo [a; b] em n segmentos iguais e calculando a seguinte soma:

$$h * [\frac{f(x_o)}{2} + \frac{f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i)] \qquad (3.1)$$

$$h = \frac{(b-a)}{n} \text{ e } x_i = a + i * h, i = 1, ..., (n-1)$$

- Colocando f(x) em uma rotina, podemos escrever um programa para calcular uma integral utilizando o método do trapézio.

Versão Sequencial

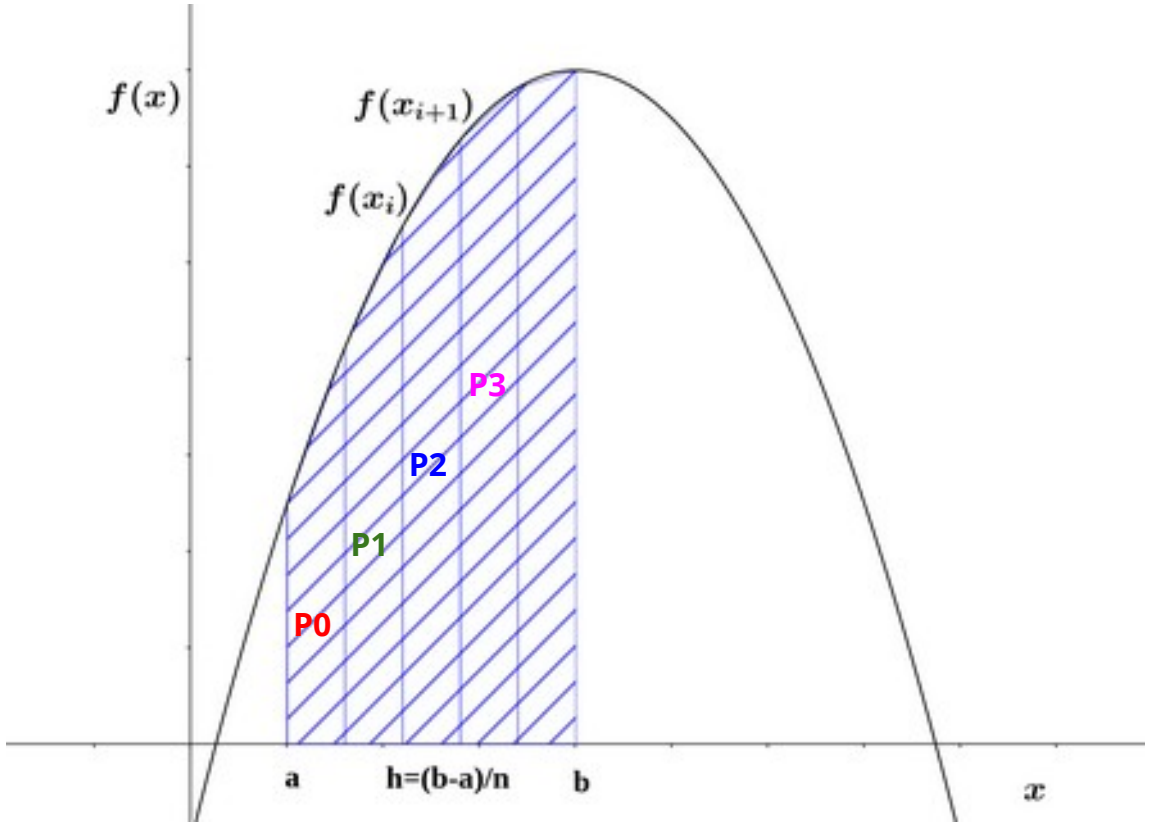
```c
#include <stdio.h>
#include <math.h>
float f(float x) {   /* Calcula f(x). */
float return_val;
   return_val=exp(x);
   return return_val;
}
int main(int argc, char *argv[]) {  /* trapezio_seq.c  */
float integral;   /* integral armazena resultado final */
float a, b;       /* a, b -  limite esquerdo e direito da função */
int   i,n;        /* n - número de trapezóides */
float x, h;       /* h - largura da base do trapezóide */

   printf("Entre a, b, e n \n");
   scanf("%f %f %d", &a, &b, &n);
   h = (b-a)/n;
   integral = (f(a) + f(b))/2.0;
   x = a;
   for (i = 1; i < n; i++) {
        x += h;
        integral += f(x);
   }
   integral *= h;
   printf("Com n = %d trapezóides, a estimativa \n", n);
   printf("da integral de %f até %f = %f\n", a, b, integral);
   return(0);
}
```

Método do Trapézio

- Este método divide o espaço de integração em **N** intervalos de tamanho **h**, onde **h = (b -a) /N**.
- Uma maneira simples de distribuir o trabalho entre os diversos processadores seria atribuir **N/P** intervalos em bloco para cada processo.
- Contudo, esse método tem dificuldade quando **N** não é múltiplo de **P,** e pode apresentar algum desbalanceamento de carga.
- Assim, uma melhor forma de distribuição é atribuir os intervalos alternadamente a cada processo, como ilustrado na figura seguinte.

Integral Definida - Método do Trapézio



Método do Trapézio

- Essa forma de distribuição pode ser realizada facilmente atribuindo o primeiro intervalo para cada processo **p** com a fórmula:

  $x_p = a + h * (ranque + 1)$

- E saltando **num_proc** intervalos **h** até que o valor de **x** seja maior do que **b**.
- Assim, mesmo que **N** não seja múltiplo de **P**, as iterações restantes são distribuídas o mais igualmente possível entre os processos, melhorando assim o balanceamento de carga.

Versão Paralela

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
/* h é o mesmo para todos os processos */
h = (b - a)/n;
/* O processo 0 calcula o valor de f(x)/2 em a e b */
if (meu_ranque == 0) {
    tempo_inicial = MPI_Wtime();
    integral = (f(a) + f(b))/2.0;
}
/* Cada processo calcula a integral sobre n/num_procs trapézios */
for (x = a+h*(meu_ranque+1); x < b ; x += num_procs*h) {
     integral += f(x);
}
integral = integral*h;
```

Versão Paralela

```
/* O processo 0 soma as integrais parciais recebidas */
if (meu_ranque == 0) {
    total = integral;
    for (origem = 1; origem < num_procs; origem++) {
        MPI_Recv(&integral, 1, MPI_DOUBLE, origem, etiq,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total += integral;
    }
} else {
 /* As integrais parciais são enviadas para o processo 0 */
    MPI_Send(&integral, 1, MPI_DOUBLE, destino, etiq,
MPI_COMM_WORLD);
 }
 /* Imprime o resultado */
 if (meu_ranque == 0) {
    tempo_final = MPI_Wtime();
    printf("Foram gastos %3.1f segundos\n",tempo_final-
tempo_inicial);
    printf("Com n = %ld trapezoides, a estimativa\n", n);
    printf("da integral de %lf até %lf = %lf \n", a, b, total);
 }
 MPI_Finalize();
 return(0);
```

Otimizando o programa

- Lembre-se também que o MPI oferece uma operação coletiva de redução, que pode ser utilizada para simplificar o código e otimizar a execução em paralelo.
- Assim, podemos reescrever as últimas linhas do programa substituindo o laço de envio dos resultados parciais pela rotina de redução, que é uma forma muito mais eficiente para o cálculo do valor final da integral.

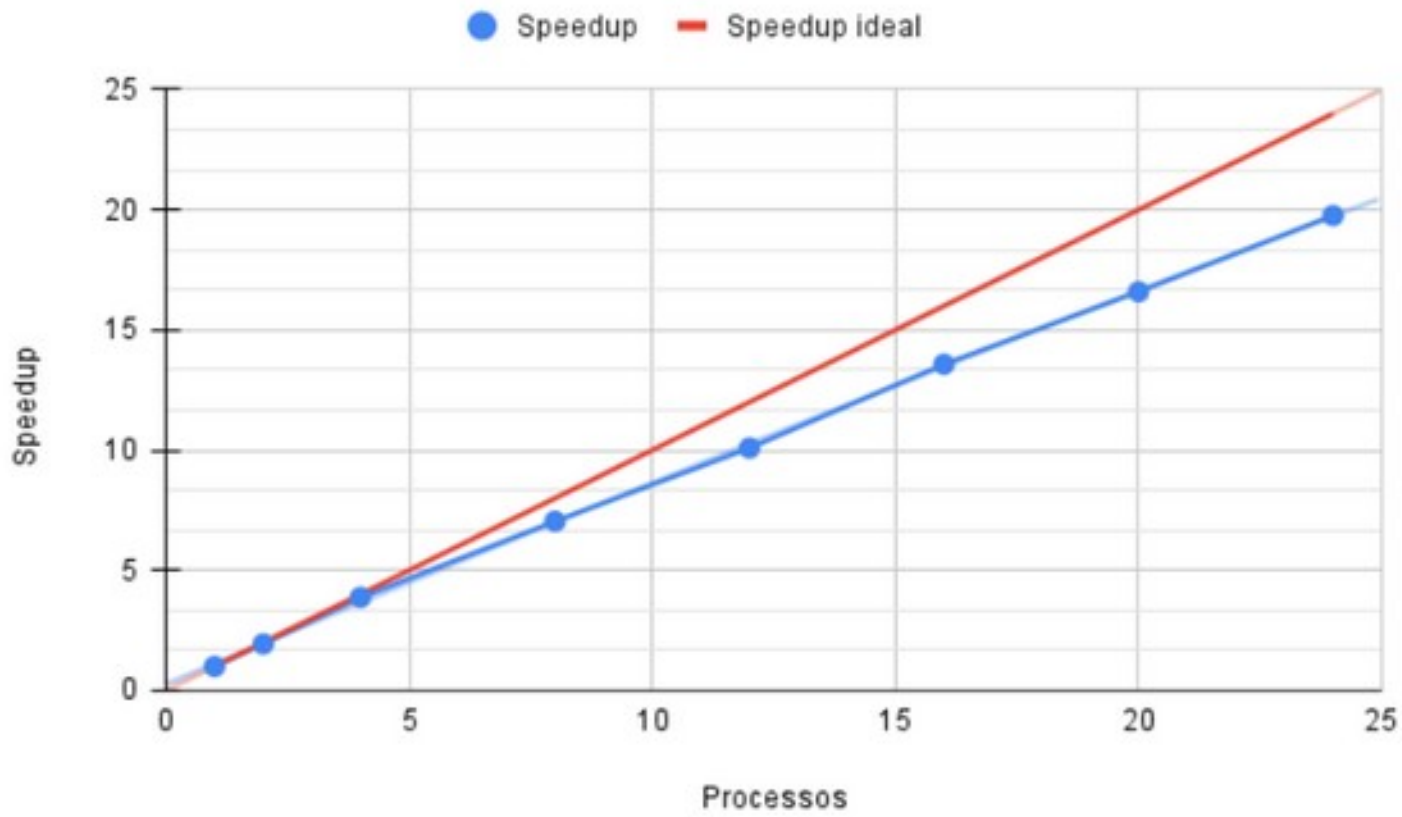
Versão Paralela Otimizada

```
/* Soma as integrais calculadas por cada processo */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD)
    ;
    /* Imprime o resultado */
    if (meu_ranque == 0) {
        tempo_final = MPI_Wtime();
        printf("Foram gastos %3.1f segundos\n",tempo_final-
   tempo_inicial);
        printf("Com n = %ld trapezoides, a estimativa\n", n);
        printf("da integral de %lf até %lf = %lf \n", a, b, total);
    }
    MPI_Finalize();
    return(0);
}
```

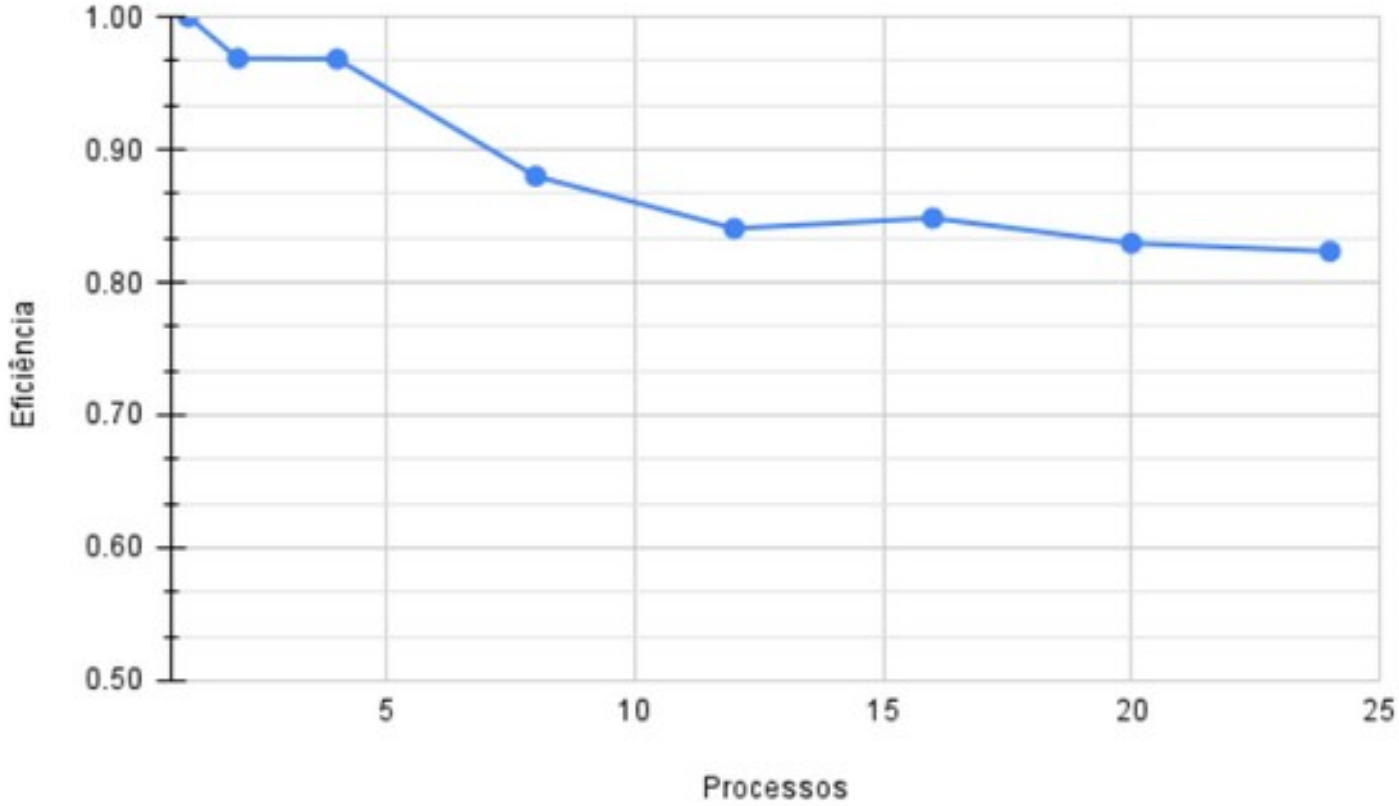
Método do trapézio

```
$ mpicc -o teste mpi_trapezio.c
$ mpirun -np 4 ./bin/mpi_trapezio
Foram gastos 1.45700 segundos
Com n = 500000000 trapezoides, a estimativa
da integral de 0.000000 até 1.000000 = 1.718282
```

Speedup


Speedup
Speedup ideal
25
20
15
10
5
0
Speedup
0
5
10
15
20
25
Processos

Eficiência

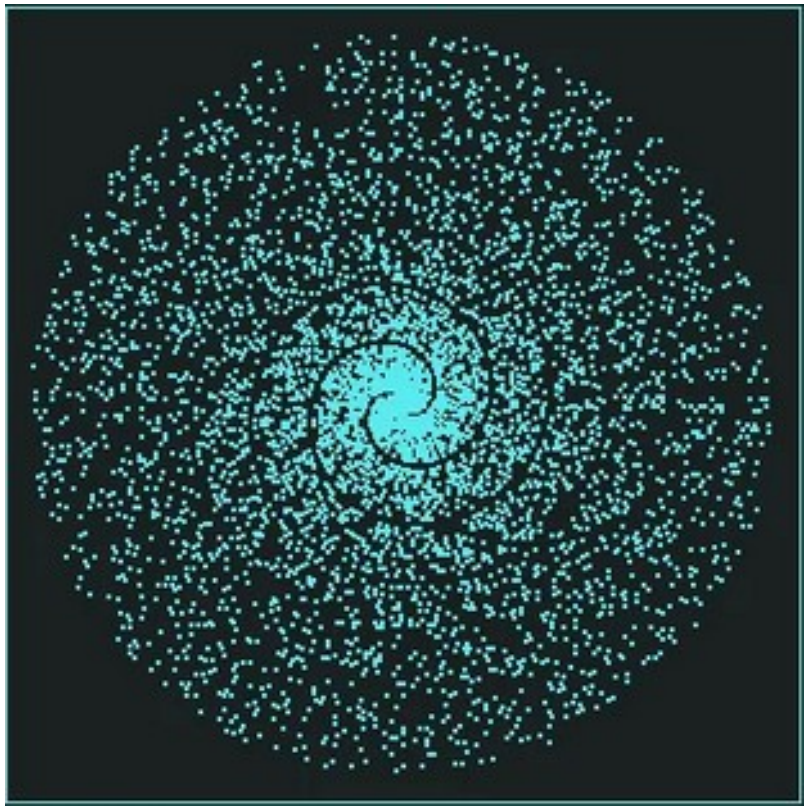


Números Primos

Números primos

- O programa em questão serve para determinar a quantidade de números primos entre 0 e um determinado valor inteiro N.
- Embora possa parecer um programa trivial a princípio, ele tem algumas particularidades que o tornam um problema interessante.
- Na matemática, o Teorema do Número Primo (TNP) descreve a distribuição assintótica dos números primos entre os inteiros positivos.
- Ele formaliza a ideia intuitiva de que os números primos tornam-se menos comuns à medida que N aumenta, quantificando precisamente a taxa em que isso ocorre.

Distribuição de primos - coordenadas polares



Números primos

- O que isso implica na programação paralela?
- Bom, a nossa primeira tentativa de paralelização seria dividir o total de N números igualmente entre os P processadores disponíveis, ou seja, N/P valores para cada uma das *threads*.
- No entanto, há uma implicação importante: a distribuição dos números primos não é uniforme entre os inteiros.
- Isso torna essa abordagem de divisão direta ineficaz, pois as *threads* que receberem intervalos com uma maior concentração de números primos terão uma carga de trabalho significativamente maior.
- Isso pode levar a um desequilíbrio no desempenho, com algumas *threads* concluindo suas tarefas mais rapidamente, enquanto outras permanecem ocupadas, resultando em subutilização dos recursos disponíveis.

Números primos

- Mas antes de avançarmos nessas questões, vamos ver como se comporta o algoritmo sequencial de busca de primos.
- Essa solução descarta todos os números pares de início, pois obviamente eles não são primos.
- Em seguida, é verificado se N é divisível por algum número ímpar entre 0 e a raiz quadrada de N. Por que isso?
- Se um número composto N pode ser fatorado como N = a * b, então pelo menos um dos fatores (a ou b) deve ser menor ou igual à raiz quadrada de N.
- Se ambos a e b forem maiores que a raiz quadrada de N, então ao multiplicá-los, o resultado seria maior que N. Perfeito?

Números primos Sequencial

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

int primo (long int n) {
  for (long int i = 3; i < (int)(sqrt(n) + 1); i+=2)
       if (n%i == 0)  return 0;
  return 1;
}

int main(int argc, char *argv[]) { /* primos_seq.c  */
  int total = 0;
  long int i, n=10000;
  if (argc > 1) n = strtol(argv[1], (char **) NULL, 10);
  for (i = 3; i <= n; i += 2)
      if (primo(i) == 1) total++;
  total += 1;     /* Acrescenta o dois, que também é primo */
  printf("Quant. de primos entre 1 e n: %d \n", total);
  return(0);
}
```

Números primos

- Uma estratégia mais simples de paralelização seria distribuir as tarefas entre os processos em lote, passando uma faixa contínua de valores para cada processo verificar a quantidade de números primos no intervalo recebido.
- Mas como já vimos, essa não é uma boa abordagem, pois a distribuição de números primos não é uniforme ao longo do espaço de números inteiros.
- Um alternativa possível é cada processo verificar alternadamente se cada número ímpar é primo, o que na prática significa que cada processo comece verificando um número ímpar diferente e saltando **num_proc** números ímpares entre cada número ímpar verificado.

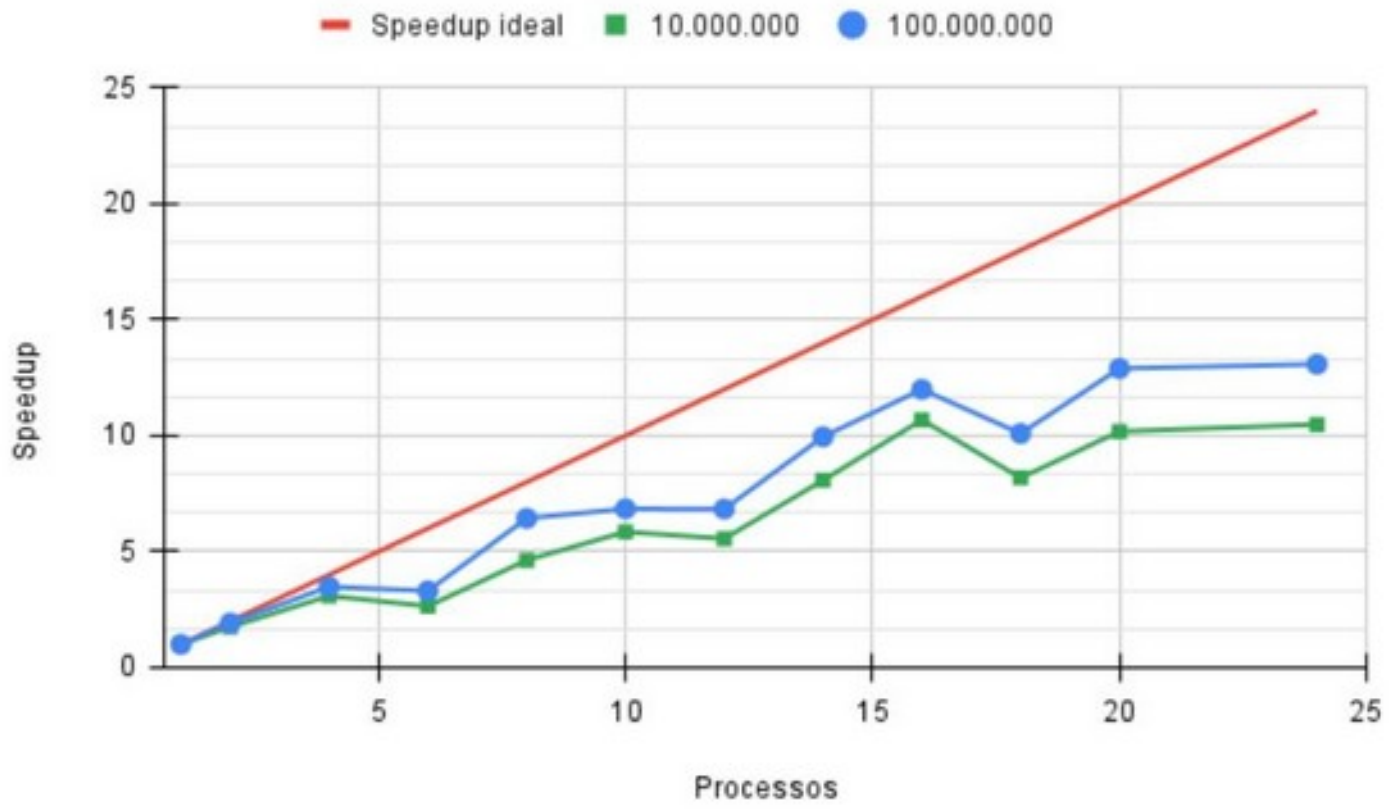
Números primos (paralelo)

```
inicio = 3 + meu_ranque*2;
salto = num_procs*2;
for (i = inicio; i <= n; i += salto) {
            if(primo(i) == 1) cont++;
}
MPI_Reduce(&cont, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
if (meu_ranque == 0) {
total += 1;    /* Acrescenta o dois, que também é primo */
          printf("Quant. de primos entre 1 e n: %d \n", total);
}
MPI_Finalize();
return(0);
```

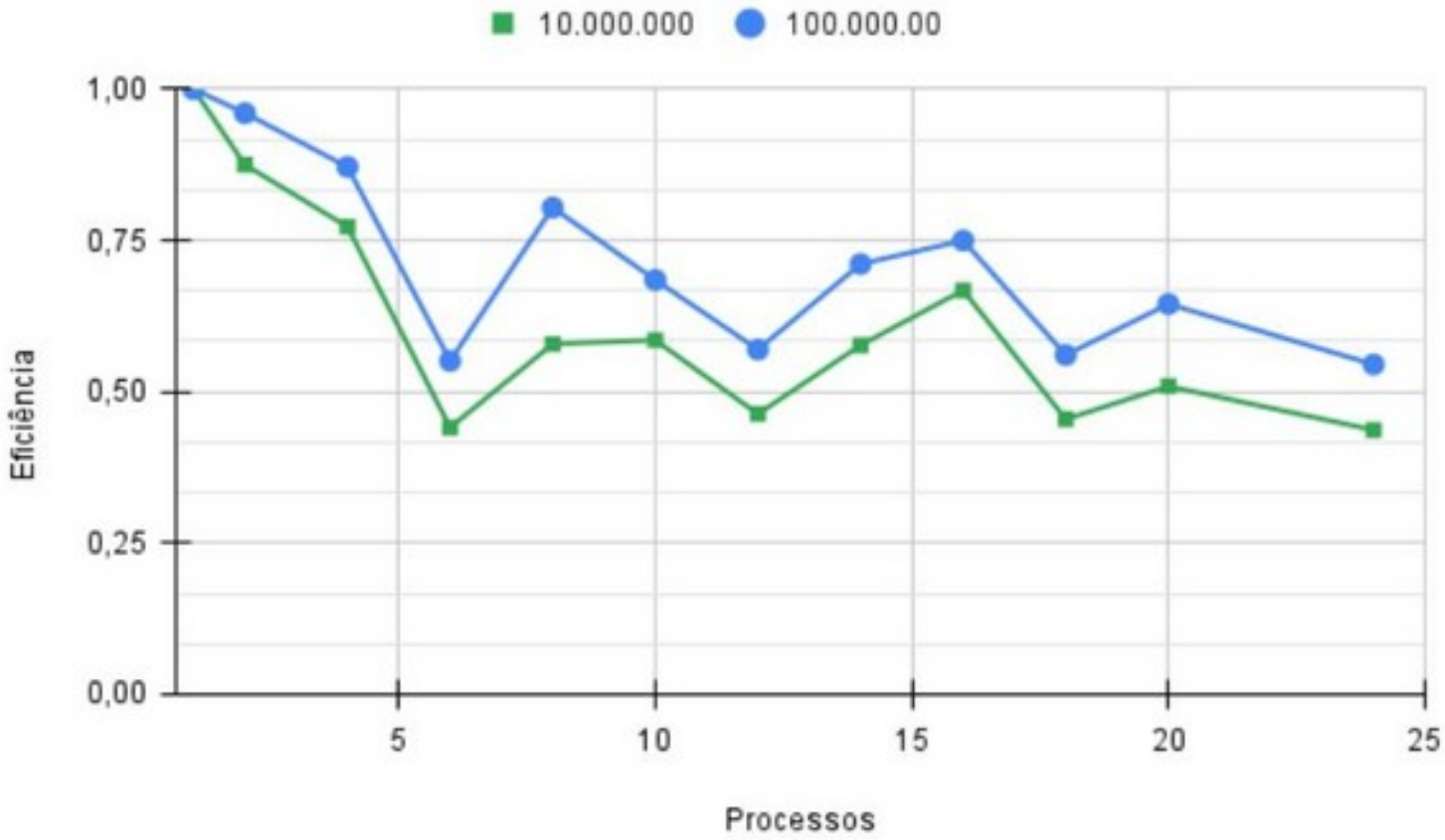
Números primos (paralelo)

```
$ mpicc -o teste mpi_primos.c -lm
$ mpirun -np 4 ./teste
    Quant. de primos entre 1 e n: 664579
    Tempo de execucao: 3.418
```

Speedup


Speedup ideal
10.000.000
100.000.000
25
20
15
10
5
0
Speedup
5
10
15
20
25
Processos

Eficiência


10.000.000
100.000.00
1,00
0,75
0,50
0,25
0,00
Eficiência
5
10
15
20
25
Processos

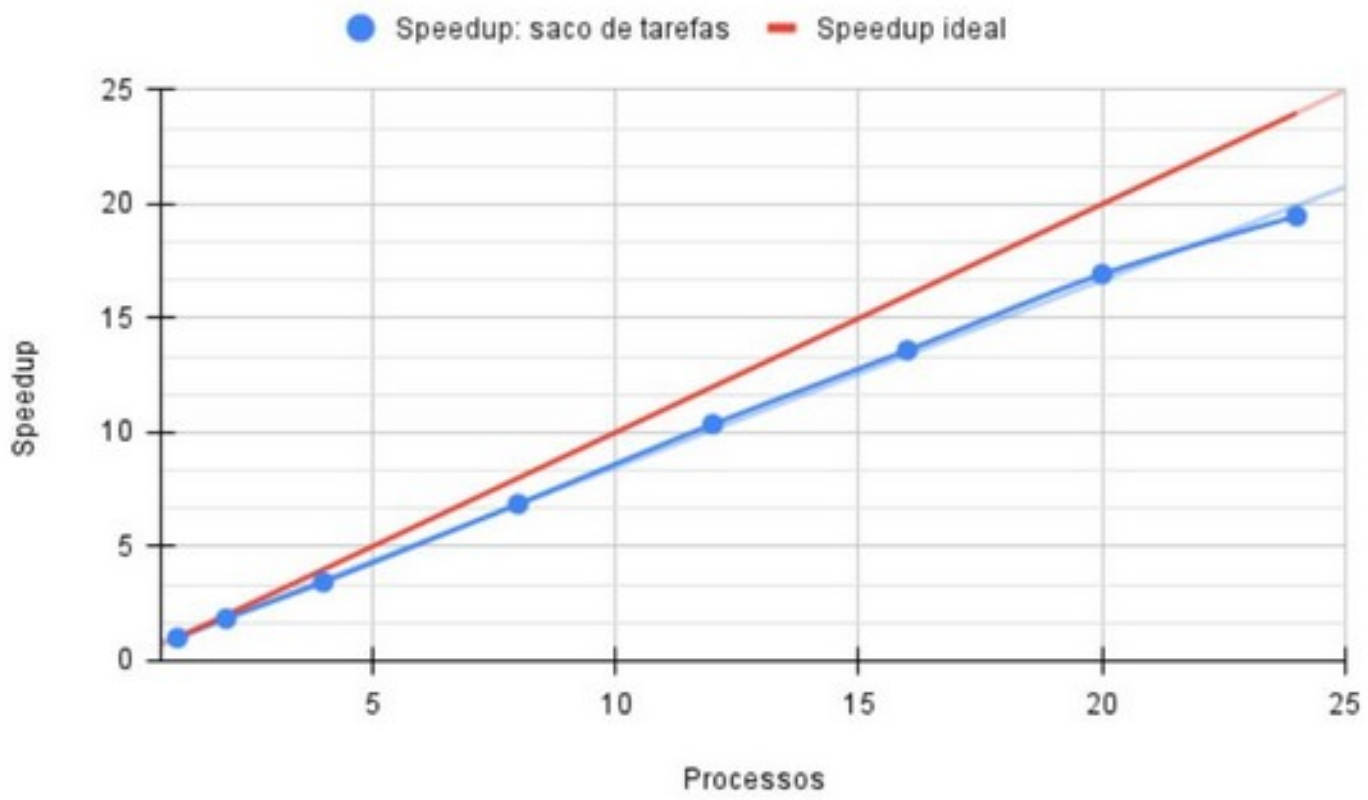
Números primos (saco de tarefas)

- Apesar dos cuidados que tomamos para a distribuição dos dados, há uma irregularidade muito grande no gráfico para valores do número de processadores iguais a 3, 5, 6, 7, 9, 10, 11, 12, 13 e 15.
- Curiosamente, nesses casos, existem processos que identificam apenas um ou nenhum número primo, terminando muito antes dos demais processos, que estão sobrecarregados calculando todos os demais números primos.
- Para vencer essa dificuldade é necessário empregar um outro método de paralelização chamado de “saco de tarefas”. Nesse método um processo (mestre) fica responsável por enviar as tarefas os demais processos (trabalhadores).

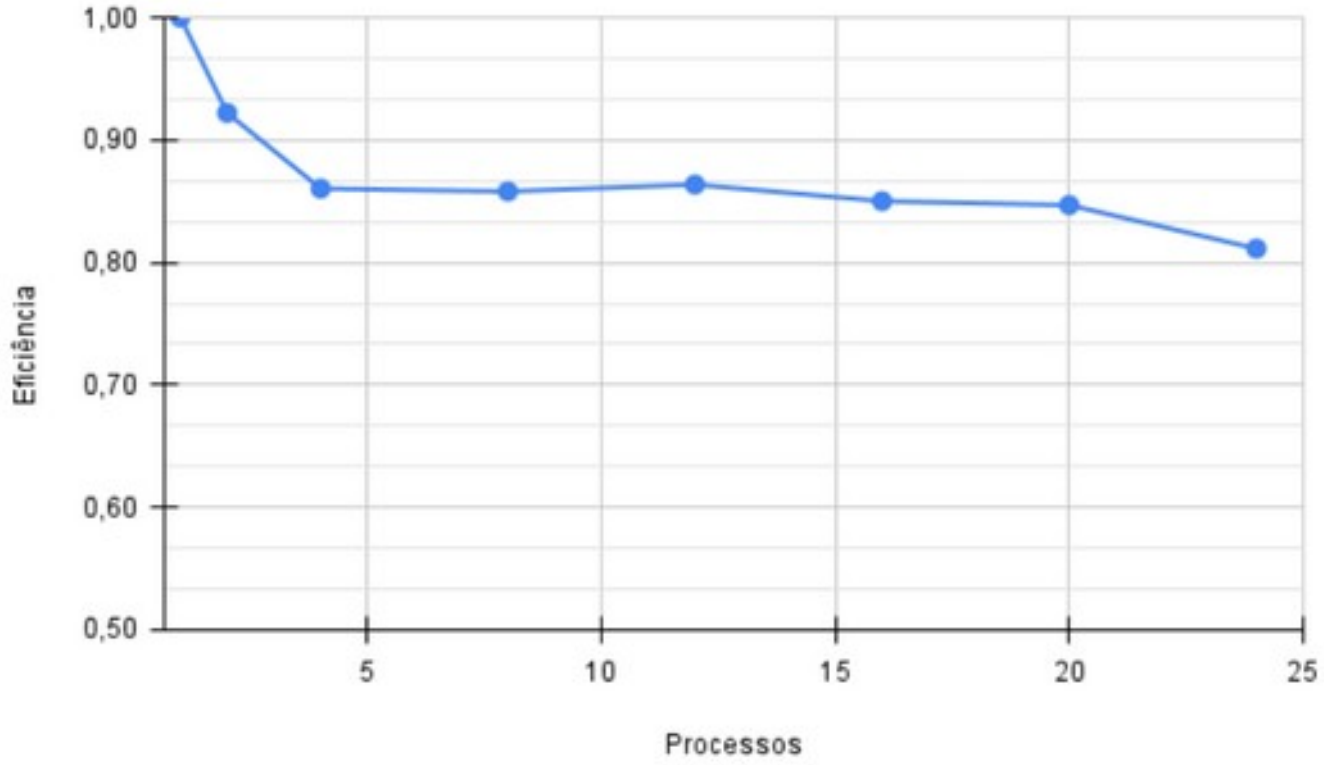
Números primos (paralelo)

- Assim que uma tarefa é terminada e o resultado enviado para o mestre, uma outra tarefa será alocada para o trabalhador e assim sucessivamente até que não haja mais tarefas para serem executadas.
- Maiores detalhes sobre esse tipo de implementação podem ser encontradas no livro texto.
- Apresentamos a seguir os gráficos com os resultados de speedup e eficiência deste método, que mostrou-se muito melhor.
- Isso mostra que a programação paralela não possui regras fixas, sendo que as soluções devem ser encontradas caso a caso.

Speedup


Speedup: saco de tarefas
Speedup ideal
Speedup
25
20
15
10
5
0
5
10
15
20
25
Processos

Eficiência



Multiplicação Matriz Vetor

Multiplicação Matriz-Vetor

- Um outro estudo de caso que faremos é a multiplicação de uma matriz por um vetor, que tem como resultado um vetor.

$$A_{m,n} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \quad b_n = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

$$c_m = \text{Ab} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{m-1} \end{bmatrix}$$

$$c_i = a_{i,0}.b_0 + a_{i,1}.b_1 + a_{i,2}.b_2 + \cdots + a_{i,n-1}.b_{n-1}$$

Multiplicação Matriz-Vetor

- Ao lado apresentamos um exemplo da operação.
- Note que cada elemento do vetor resultado é o produto escalar de uma linha da matriz com o vetor de entrada.

$$A \times b = c$$

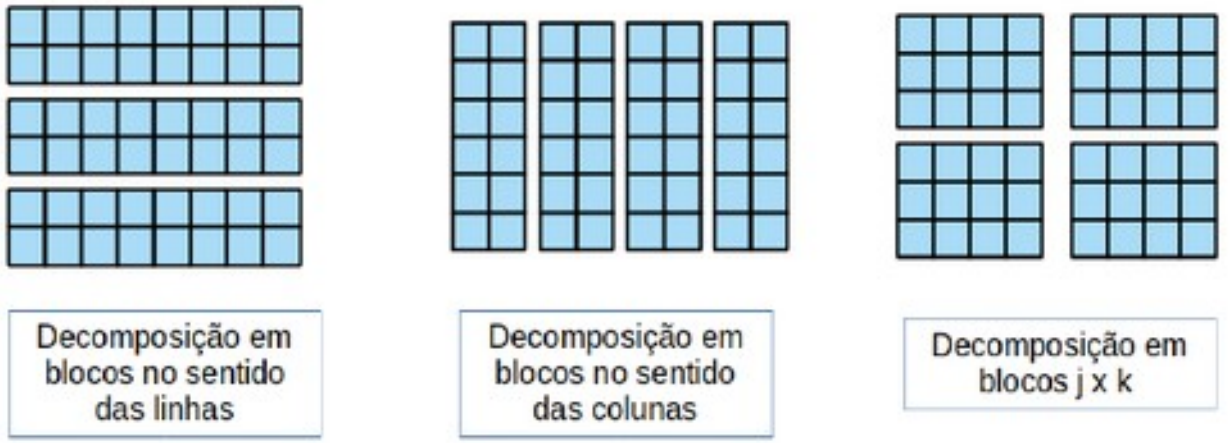
$$\begin{bmatrix} 2 & 1 & 3 & 4 & 0 \\ 5 & -1 & 2 & -2 & 4 \\ 0 & 3 & 4 & 1 & 2 \\ 2 & 3 & 1 & -3 & 0 \end{bmatrix} \times \begin{bmatrix} 3 \\ 1 \\ 4 \\ 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 19 \\ 34 \\ 25 \\ 13 \end{bmatrix}$$

Código Sequencial

```
#include <stdio.h>
#include <stdlib.h>
void mxv(int m, int n, double* A, double* b, double* c)
{
int i, j;
   for (i = 0; i < m; i++) {
       c[i] = 0.0;
       for (j = 0; j < n; j++)
            c[i] += A[i*n + j]*b[j];
   }
}
int main(int argc, char *argv[]) { /* mxv_seq.c  */
double *A,*b,*c;
int i, j, m, n;
    printf("Por favor entre com m e n: ");
    scanf("%d %d",&m,&n);
    printf("\n");
    A=(double *)malloc(m*n*sizeof(double));
    b=(double *)malloc(n*sizeof(double));
    c=(double *)malloc(m*sizeof(double));
   printf("Atribuindo valor inicial à matriz A e ao vetor b\n");
   for (j = 0; j < n; j++)
        b[j] = 2.0;
   for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
             A[i*n + j] = i;
   printf("Mutiplicando a matriz A com o vetor b\n");
   (void) mxv(m, n, A, b, c);
   free(A);
   free(b);
   free(c);
   return(0);
}
```

Particionamento dos dados

- Há várias maneiras de decompor o problema e distribuir os dados entre os diversos processos.



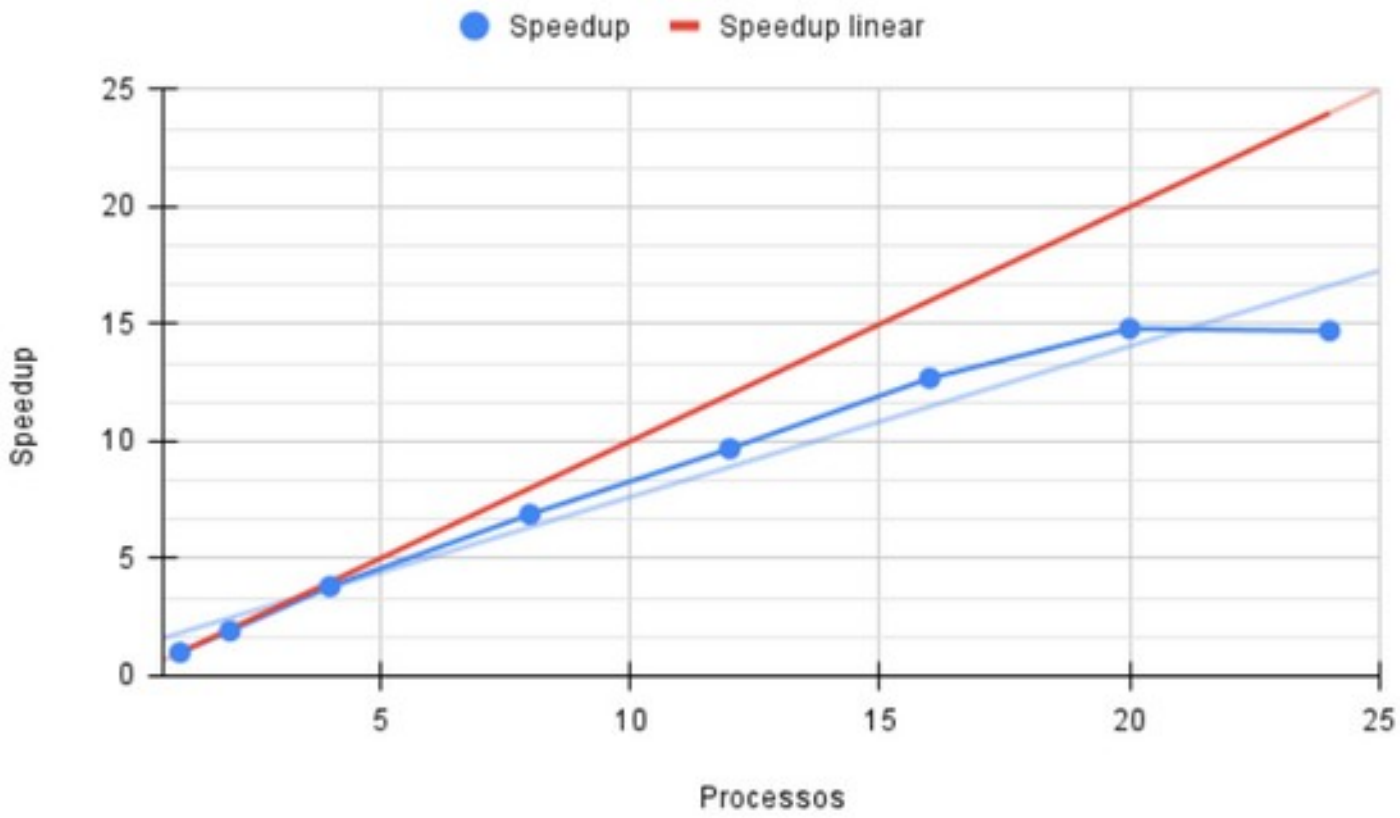
Particionamento dos dados

- Cada uma dessas formas de decomposição tem as suas vantagens e desvantagens, além de complexidades distintas.
- Por simplicidade, vamos assumir que utilizaremos a primeira alternativa de distribuição de dados, com cada processo possuindo um bloco de linhas da matriz **A** e os vetores **b** e **c** replicados em cada processo.
- Uma análise simples de complexidade, supondo-se $m = n$, indica uma complexidade computacional sequencial de $O(n^2)$. Quando **p** processos são utilizados, a complexidade computacional por processo, sem custos de comunicação, igual a $O(n^2/p)$.

Código Paralelo

```
inicio = MPI_Wtime();
    /* Difunde o vetor b para todos os processos */
MPI_Bcast(&b[0], n, MPI_DOUBLE, raiz, MPI_COMM_WORLD);
    /* Distribui as linhas da matriz A entre todos os processos */
MPI_Scatter(A, n, MPI_DOUBLE, Aloc, n, MPI_DOUBLE, raiz, MPI_COMM_WORLD);
/* Cada processo faz o cálculo parcial de 'c'  */
(void) mxv(n, Aloc, b, cloc);
  /* O processo raiz coleta o vetor 'c' dos demais processos */
MPI_Gather(cloc, 1, MPI_DOUBLE, c, 1, MPI_DOUBLE, raiz, MPI_COMM_WORLD);
fim = MPI_Wtime();
```

Speedup


Speedup
Speedup linear
25
20
15
10
5
0
Speedup
5
10
15
20
25
Processos

Eficiência